

ISIDRO RAMOS

PUBLICACIONES 1968-2013

VOLUMEN 1:
MADRID, BILBAO, VALENCIA

VALENCIA, SEPTIEMBRE DE 2014



Isidro recogiendo el título de Director Emérito del grupo ISSI (18/12/2009)

PRÓLOGO

A primeros de octubre de 1982, comencé a cursar el segundo ciclo de la Licenciatura en Ciencias Físicas en la *Universitat de València* (UV). Escogí la especialidad de Electrónica e Informática por diversas razones, la más poderosa de las cuales fue lo atractivo —y casi misterioso— que resultaba entonces el trabajo con ordenadores. En esos momentos, la presencia de la Informática en el plan de estudios era bastante testimonial, limitándose a una asignatura llamada «Programación» en 4º curso, y otra llamada «Calculadoras Electrónicas» en el curso siguiente. Como estudiantes, esperábamos con interés las primeras clases de programación, y llegó el día del inicio del curso. Esa fue la primera vez que me encontré con Isidro.

Tras la presentación de rigor —«soy Isidro Ramos, Catedrático de…»—, comenzó la primera clase del curso. Primer tema: «Estudio de una memoria direccionable». Bibliografía: Apuntes de Claude Pair. Me hubiese gustado tener una cámara para filmar, al final de la clase, la cara que se nos quedó a todos los que esperábamos otra cosa totalmente diferente. Yo diría que nos costó un tiempo recuperarnos del shock. Cierto es, para ser justos, que las clases de problemas se centraron en el lenguaje Pascal, lo que mitigó notablemente el impacto de la teoría. Como bien apunta José Ramón Garitagoitia en este libro, en las clases de Isidro se recogían ideas que se plasmarían, años más tarde, en el concepto de Tipo Abstracto de Datos que tan relevante ha sido en el desarrollo de lenguajes formales de especificación, a los que Isidro ha dedicado un largo periodo de su carrera investigadora.

Además de su labor como catedrático, Isidro ostentaba los cargos de Director del Centro de Cálculo de la UV, y también del Centro de Cálculo de la Diputació de València. Al co-

menzar el 5º curso, de nuevo Isidro Ramos como profesor. «¿Qué nos contará este año?», pensamos. Tras la primera clase, se despejó la incógnita: el contenido fundamental del curso sería la construcción de compiladores, propuesta que agradó a la audiencia. Sin embargo, a las pocas semanas de iniciado el curso, Isidro se despidió de nosotros, pues había sido nombrado Rector Fundador de la Universidad de Castilla—La Mancha (UCLM) por el Ministro de Educación.

Volví a encontrar a Isidro en 1987. A los meses de ser contratado en el Departamento de Sistemas Informáticos y Computación (DSIC), llegó la noticia de que un nuevo catedrático iba a dirigir el Departamento. Su nombre era Isidro Ramos. Venía con ímpetu renovado tras su paréntesis de gestión en la UCLM —durante el cual, sin embargo, no dejó completamente de lado su actividad investigadora, como podrá comprobar el lector—. Desde entonces desarrolló una intensa labor de formación investigadora de los jóvenes que, al amparo del auge de los estudios de Informática, habían entrado a formar parte del DSIC. En pocos años, las primeras tesis doctorales dirigidas por Isidro en el DSIC comenzaron a defenderse, y el grupo inicial fue creciendo hasta convertirse en uno de los más numerosos del departamento.

Con el tiempo, algunos de los miembros del grupo crearon sus propios grupos, aunque casi siempre se mantuvo, de algún modo, la ligazón ente ellos derivada de la historia de sus miembros. Yo, junto a algunos compañeros más, he permanecido cerca de Isidro desde entonces hasta la actualidad. Hemos compartido proyectos de investigación, publicaciones, y muchos ratos de conversaciones, ya sea en la sala de café, o en su casa, a la que siempre nos hemos sentido invitados. A lo largo de los años, y de la mano de Isidro, hemos conocido a investigadores de gran nivel nacional e internacional, especialmente en la comunidad de Ingeniería del Software, que tanto contribuyó a crear. Su voluntad integradora la ha demostrado con frecuencia acogiendo en el grupo a estudiantes de diversas procedencias, a grupos emergentes que necesitaban el primer empujón, incluso su dedicación a tareas de gestión académica a nivel nacional e internacional.

No he conocido, en mi ya larga trayectoria, a nadie con el entusiasmo por su trabajo que ha derrochado Isidro. Pese a no tener que demostrar nada a nadie, ha mantenido una curiosidad intelectual que le ha llevado a abrirse a nuevas y prometedoras ideas en Ingeniería del Software, y a comunicarlas a los miembros de su grupo. Isidro pertenece a una generación que ha dedicado su vida académica a la investigación y la docencia por pura vocación, poniendo siempre en valor al grupo frente a las individualidades. Convencido de que la responsabilidad ha de ejercerse, frente al gobierno cuantitativo imperante hoy en la Academia, ha asumido sus responsabilidades con espíritu de servicio a los demás. Nunca le he oído rechazar a una persona sin antes darle una o más oportunidades. Su capacidad de análisis y su intenso bagaje cultural hacen de él una persona intelectualmente muy atractiva, con quien nunca se puede mantener una conversación aburrida.

En 2013, Isidro cumplió 70 años, edad que conlleva la jubilación como profesor universitario tras una trayectoria académica de más de 40 años, a lo largo de la cual ha desarrollado una ingente tarea de investigación, docencia y gestión académica sin parangón en el campo de la Informática en España. Con tal motivo, a primeros de 2014 comencé una tarea que llevaba en mente desde hacía algún tiempo para homenajear a quien ha sido mi maestro y compañero de tantos años: recopilar toda su producción científica y compilarla en un libro que le regalaría, junto con mis compañeros, en alguna cena de grupo al llegar el día de su jubilación, que no su retiro. El desafío fue de aúpa.

Durante meses, el curriculum vitae de Isidro ha estado sobre mi mesa, a modo de *checklist* en la que iba marcando los artículos que iba consiguiendo. Conforme retrocedía en el tiempo, la cosa se iba poniendo más difícil, por lo que decidí pedir ayuda a sus coautores de

todas las épocas. He de decir, en este momento, que todo aquél a quien expliqué mi propósito se mostró entusiasmado con la idea, y mostró su disposición a facilitarme todo el material al que tuviese acceso. No solo eso, sino que muchos de ellos manifestaron su deseo de unirse a cualquier homenaje que se hiciese a Isidro.

Es por ello que, finalmente, lo que comenzó como una recopilación de trabajos se ha convertido en algo mucho más grande: la muestra del afecto y admiración que la mayoría de las personas que han contribuido profesan a Isidro. De la misma forma, se convirtió en una obra colectiva en la que muchos han aportado tiempo y esfuerzo. Quisiera agradecer especialmente a Mª Carmen Penadés las horas pasadas en la Hemeroteca de la Universidad Complutense de Madrid (UCM) buscando los ejemplares de la Revista de Automática, el Boletín del Centro de Cálculo de la UCM, etc. Ella misma nos lo cuenta con detalle en este preámbulo. Abel Gómez, con sus proverbiales e impagables paciencia y dedicación, recogió una primera «versión final» hecha artesanalmente y la transformó en el documento que está en sus manos. Quiero también mencionar a las personas que han elaborado los prólogos de las diferentes partes en las que está organizado este libro: Ernesto García Camarero, Roberto Moya, José Ramón Garitagoitia, José Francisco Quiles, María Alpuente, Maria Dolores Lozano, José Ángel Carsí, Patricio Letelier, Silvia Abrahão y Emilio Insfrán. Todos ellos ofrecen su visión personal de una parte de la carrera de Isidro, desde el punto de vista científico y humano. Ricardo Baeza y Marcos Borges han colaborado de manera especial en recuperar vivencias de la época CYTED. El silencio cómplice de Elisa Soler, así como su localización de fotos valiosas, ha sido igualmente importante.

Esto no hubiese sido posible sin la colaboración de un gran número de coautores que me han facilitado versiones electrónicas de los trabajos. La lista es larga, y corro el riesgo de olvidarme de algunos de ellos, por lo que me van a permitir que no los mencione uno por uno aquí. Agradezco igualmente a todos los amigos, de todas las épocas, que han querido manifestar su afecto a Isidro escribiendo unos párrafos que están recogidos al final del libro. Ha sido realmente emocionante leer las cálidas respuestas recibidas a mi petición de colaboración en este homenaje.

Finalmente, agradezco profundamente a mi amigo Isidro todo lo que me ha aportado en más de 25 años. Todavía hoy, mientras disfruta de una apacible vida de investigador senior, sigue siendo el referente en muchas de las actividades en las que estoy inmerso. Y espero que así continúe mientras el cuerpo aguante.

José Hilario Canós Cerdá
Septiembre de 2014

DÍAS DE HEMEROTECA

Justo antes de las vacaciones de Semana Santa, Hilario me envía un correo con unos enlaces y me dice que tenemos que hablar. En aquel momento no podía imaginar que en las siguientes semanas mi actividad docente/investigadora iba a descubrir una nueva faceta, hoy en día casi olvidada. Isidro Ramos inició su actividad en la Universidad de Madrid (ahora UCM) y allí fueron publicados la mayoría de sus primeros trabajos e investigaciones. La labor que me propuso Hilario, y que acepté encantada, fue la de acercarme a la Hemeroteca de la Facultad de Físicas de la UCM y solicitar algunos trabajos de Isidro; en el catálogo *Cisne* constaba que estaban allí. La tarea parecía fácil y sencilla, pero al llegar me encontré con un primer obstáculo: por reformas, todas las publicaciones más antiguas de la hemeroteca, y éstas lo eran, estaban en cajas en un sótano de la Facultad. La misma bibliotecaria, muy amablemente, me dijo: «Sabía que esto pasaría, pero ahora mismo y en los 6-10 meses próximos, mientras duren las reformas, es imposible acceder a ese material». Yo creo que vio mi cara de decepción, y rápidamente buscó en su ordenador y me dijo que también estaban localizadas en la Biblioteca de la Facultad de Informática, del mismo campus. Nos despedimos, no sin antes darme detalladamente todas las indicaciones para llegar.

Tras un paseo agradable bajo la lluvia, llegué a la Facultad de Informática de la UCM, localicé la biblioteca y pregunté por una las publicaciones que venía a buscar: «Revista de Automática. Año 1971». La bibliotecaria miró en su PC y me dijo: «Sí está, P59 – Vitrina del tercer piso; tengo que llamar a una compañera para que abra las vitrinas y busque el material. Es antiguo y no se puede dejar en préstamo». Al llegar al tercer piso, me di cuenta que la tarea no iba a ser tan rápida cómo pensaba: había al menos 5-6 tomos P59, bastante gruesos. Todas las ediciones que tenían de la Revista de Automática, que posteriormente pasó a ser Revista de Automática e Informática, estaban encuadernados en esos tomos. La bibliotecaria me dio uno al azar y, justo al abrirlo, como una premonición, me aparece el nombre de Isidro Ramos, «Lenguaje para la escritura de traductores». Yo emocionada, le dije: «Si, sí está, es éste». Fue entonces cuando les explique con detalle el motivo de mi visita: recopilar las publicaciones de Isidro Ramos. Me sorprendió la amabilidad de las bibliotecarias, Araceli y Ángeles, puesto que durante toda la mañana me estuvieron bajando tomos P-59 entre los que yo buscaba publicaciones de Isidro, dándome permiso para escanear con mi móvil, página a página. Esta tarea se prolongó un par

de mañanas y mi pequeña amistad con ellas fue en aumento, facilitándome todo el material. En total, conseguí 12 publicaciones en dicha revista entre los años 1970 y 1981, incluido el trabajo «Sistemas de creación y mantenimiento de bancos de información» publicado en tres partes y que recibió el Premio Nacional CCS de Informática.

Para finalizar mi labor, sólo me faltaban tres trabajos publicados en el Boletín del Centro de Cálculo de la Universidad de Madrid (CCUM). Al preguntar por ellos, me dijeron que mejor que volviera por la tarde, ya que las bibliotecarias de ese turno estaban catalogando parte de ese material, y seguro me podrían ayudar mejor. Así lo hice, y sorprendentemente, nada más llegar y decir mi nombre, ya estaban perfectamente informadas de que venía a consultar el Boletín del CCUM. Pero mi sorpresa fue en aumento cuando al decir que buscaba unas publicaciones de «Isidr…» me cortaron y dijeron las dos casi al unísono: «Isidro Ramos Salavert». Yo dije: «Sí, justo». Fueron ellas mismas las que me explicaron con todo nivel de detalle que desde hacía algunos meses estaban catalogando todas las publicaciones, seminarios, charlas, etc. que formaban parte del Boletín del CCUM. Toda esta información la estaban publicando en «Dialnet», y aunque no estaba el material digitalizado, sí se podía buscar y localizar fácilmente, pues ellas ya lo habían procesado. Estaban emocionadas de que alguien fuera a preguntar por aquellos materiales, sentían que sus muchas horas de catalogación no habían sido en balde. Pero lo más divertido fue que ellas mismas me preguntaron por Isidro, si lo conocía, si seguía en activo, cómo era…, ya que formaba parte de los nombres que más se repetían en los boletines del CCUM durante esos primeros años. Incluso me comentaron que les costó un tiempo asociar la firma «I. Ramos» de algunos trabajos, con el nombre completo, y si yo creía que era así. Les ratifiqué que seguro, «I. Ramos» era Isidro Ramos Salavert. Finalmente, y gracias a las bibliotecarias Rosa y Lucía, las 3 publicaciones que me faltaban se convirtieron en 14 ya catalogadas (… igual aparece alguna más). Ellas mismas, con una amabilidad extraordinaria, me acompañaron a hablar con el responsable de la Hemeroteca y con el responsable de Préstamos Interbibliotecarios para facilitarme todo el material ya escaneado y en archivos PDF que enviarían a mi correo electrónico a través de la Biblioteca de la UPV. Aprovecho estas líneas para agradecer a Araceli, Ángeles, Rosa y Lucía, las bibliotecarias de la Facultad de Informática de la UPM, su amabilidad y ayuda.

Reconozco que he disfrutado con la labor de recopilación y búsqueda de estas primeras publicaciones de Isidro, de ojear los contenidos que se publicaban por aquellos años —en el nacimiento de la Informática—, los congresos que se celebraban, documentados con fotos de los ponentes y los comités, el formato de los mismos, etc. Pero como la actividad de Isidro ha sido extraordinaria, la búsqueda de sus publicaciones se ha extendido también a otros organismos, como por ejemplo la Federación Española de Municipios y Provincias (FEMP), donde igualmente la bibliotecaria Montse me atendió amablemente y nos facilitó las publicaciones existentes de las Jornadas de Informática de la Administración Local (JIAL-83). Finalmente, otras publicaciones de sus primeros años, y ante la imposibilidad de encontrarlas, nos llevaron a Hilario y a mí a agudizar nuestro ingenio para localizarlas (pero esto ya se lo contaremos en otro momento).

No quiero terminar estas palabras sin antes reconocer el papel de Isidro durante toda mi vida académica, pues mi relación con él no se ha limitado a la búsqueda de sus primeros trabajos, sino que desde que acabe mis estudios de 5º curso de la Licenciatura en Informática en 1993, he formado parte su grupo de investigación. Recuerdo con cariño el proyecto PASO, las reuniones de grupo, y sus constantes aportaciones a todos los que éramos más jóvenes y teníamos que empezar con nuestras clases y nuestra tesis doctoral. En mis primeros años como profesora asociada compartí asignatura con Isidro (“Ingeniería de Requerimientos”), y siempre

he sentido su apoyo, tanto en lo académico como en lo personal. Y a pesar de los años ya pasados, Isidro sigue estando ahí, disponible y accesible, para cualquier cosa que quieras comentar con él. Gracias por todo, Isidro.

Mª Carmen Penadés Gramaje

… Y NOCHES DE ORDENADOR

Hace tantos meses que comenzó la preparación de estos volúmenes que lo cierto es que ya he olvidado el momento preciso en el que Hilario me comentó la tarea que estaba a punto de comenzar: preparar un libro con todas las publicaciones de Isidro a lo largo de su carrera. Lo que sí recuerdo es que pensé que iba a ser un trabajo enorme, casi titánico, y que, al menos, le ayudaría a encontrar todas aquellas publicaciones para las que solicitase mi ayuda.

Si bien en ese momento contaba con refrescar los trabajos de Isidro de los últimos años —pues los conocía de primera mano desde que fui felizmente acogido en el seno del grupo ISSI hace ya diez años siendo aún estudiante de Ingeniería Informática—, lo que no podía imaginar es que terminaría por repasar página a página (literalmente) buena parte de sus publicaciones durante los últimos 45 años. He de decir que, a pesar del trabajo que ello ha supuesto, ha sido algo realmente fascinante. 45 años en una disciplina como la nuestra dan para mucho, muchísimo.

Dada la oportunidad que se me ha dado para escribir unas líneas aquí —muy generosamente, ya que mi contribución a esta compilación ha sido de lejos la menos épica y desafiante de todas—, me gustaría contar qué he visto desde la pantalla de mi ordenador durante la creación de este «libro»: dedicación, reconocimiento y afecto.

A lo largo de los últimos meses, y especialmente de las últimas semanas, Hilario me iba comentando de vez en cuando cómo el número de artículos —y el consecuente número de páginas— iba creciendo y creciendo, hasta que un día me dijo —no sé si con alarma, sorpresa, o con qué otra emoción ;-)— que el libro iba camino de convertirse en una colección de cuatro o cinco volúmenes. Aún tenía que decidir bajo qué criterios organizar los volúmenes. Pobre de él pensé, ¡qué trabajo se está pegando!

Yo aún no había visto ni una línea del famoso libro, pero finalmente el deseado día llegó. Era 2 de agosto. Recibí un correo desde *Dropbox*: *HJ CC quiere compartir «_libroIsidro» contigo.* «Hasta ha empezado el nombre de la carpeta con un guión bajo para tener el acceso siempre a la vista» —pensé—. Un escueto mensaje decía:

Hola,

ya tengo acabado lo de Isidro. Quería pediros un favor: sin prisas, ¿podéis echar una mirada a ver si todo está bien?

Quisiera volver a él la última semana de agosto, así que ya veis que no hay ninguna prisa.

¡Gracias!

Hilario había terminado, con casi un mes y medio de antelación, el proceso de búsqueda, organización y consolidación de artículos y colaboraciones de todos. Tras meses oyendo hablar del libro de Isidro obviamente no podía esperar más para pegar esa «mirada a ver si todo está bien». Eso hice, y he de decir que el primer repaso al «libro» —cinco PDFs de 500 páginas cada uno aproximadamente— me resultó especialmente emocionante. Impresiona poder ver tantos hitos alcanzados, vivencias, trabajos, recuerdos, colaboraciones y muestras de afecto condensados en unas «pocas» páginas. Toda una trayectoria, toda una vida.

En el documento, efectivamente, todo estaba bien. El contenido estaba listo, formateado y organizado en sus respectivos volúmenes. Hilario se merecía unas vacaciones. «No te preocupes —le dije— yo me encargo de revisarlo todo y de hacer las correcciones necesarias». ¡Qué acababa de decir!

Finalmente, estas «correcciones necesarias» se han llevado —siendo comedidos— más de una noche de verano. Estando tan cerca de concluir esta tarea que ha llevado meses, Hilario y yo acordamos que hiciera una «reedición» completa que mejorase la primera versión e intentara poner en valor la admiración, afecto y gratitud hacia Isidro que sentimos todos los que hemos participado en este trabajo. Por mi parte, he hecho de simple amanuense intentando darle buena forma a estos volúmenes, procurando revisar todo el contenido, y prestando especial atención a publicaciones antiguas, donde para facilitar la lectura ha sido necesario procesar las imágenes digitalizadas desde los originales página a página. El resultado no ha sido siempre óptimo, pero esperamos que entienda el lector la dificultad que implica y la dedicación que requiere.

No quisiera terminar sin dedicarle desde estas páginas privilegiadas unas breves palabras a Isidro, con quien he tenido la suerte de encontrarme en la vida. Me siento honrado de ser parte de la larga lista de doctorandos que hemos obtenido nuestro título de doctor gracias a su guía. También me gustaría hacer notar que, si bien es cierto que cierro esta lista por ahora, tampoco me extrañaría en absoluto que dicha lista continúe creciendo conociendo el carácter de Isidro. La dedicación, generosidad, pasión y nivel investigador de Isidro son, como estos volúmenes demuestran, formidables, además de un ejemplo a seguir.

Por último quisiera añadir que siento un sincero agradecimiento por haber podido trabajar como parte del grupo ISSI durante estos últimos años. Isidro es el gran nexo de unión de la familia que es ISSI. Este grupo, que él fundó y dirigió en su etapa en la Universidad Politécnica de Valencia es una segunda casa para muchos de nosotros, el lugar donde hemos aprendido infinidad de cosas (incluso más allá de la investigación), y el lugar donde hemos encontrado a grandes amigos a lo largo y ancho del mundo. Supongo que como fundador imbuyó buena parte de su ser en el espíritu del grupo y la gente que lo ha formado.

Espero que el trabajo y cariño puestos en este sencillo homenaje sirvan para demostrar mi agradecimiento por todo ello. Nos vemos pronto Isidro.

Abel Gómez Llana

ESTRUCTURA DEL LIBRO

El libro abarca trabajos que se inician a finales de los años 60 del siglo XX, y terminan en 2013. Ante la gran cantidad de material recopilado, se optó por descartar algunos de ellos, en concreto los correspondientes a pósteres y todos aquellos presentados en talleres asociados a proyectos de investigación. Así, la mayoría de los trabajos aquí recogidos son artículos publicados en revistas o en actas de congresos y workshops, tanto nacionales como internacionales. Se ha incluido, además, transcripciones de algunas conferencias impartidas por Isidro en actos académicos.

Los artículos recopilados ocupan más de 2100 páginas. Eso ha obligado a realizar una edición en papel estructurada en cinco volúmenes, cada uno de ellos conteniendo una o más partes. Para facilitar la búsqueda, cada uno de estos volúmenes contiene un índice completo que abarca toda la compilación. Estructurar todo en partes fue un gran desafío y causó más de un dolor de cabeza. ¿Por lugares? ¿Por fechas? ¿Por temas? ¿Por tesis? Finalmente se decidió una aproximación mixta: la primera parte de la obra se estructura en torno a tres etapas fundamentales: Madrid, Bilbao, y Valencia / Castilla—La Mancha. Estas tres etapas se suceden en el tiempo, y corresponden básicamente con la afiliación de Isidro a una institución de cada uno de esos lugares. El resto —una vez establecido Isidro en la UPV— se organizó atendiendo a los grandes temas de investigación en los que trabajó a lo largo del tiempo: Métodos Formales en Ingeniería de Requisitos, Arquitecturas Software, Diseño de Interfaces de Usuario, Gestión de Modelos, Ingeniería del Software Dirigida por modelos, e Ingeniería de Líneas de Producto Software. Entre esas partes hay solapamiento temporal, pues se ha trabajado y publicado en diversos campos simultáneamente. El libro se cierra con las aportaciones personales de los amigos que, siguiendo nuestra invitación, han querido manifestar su estima hacia Isidro.

Observará el lector que las páginas conservan en su mayoría la numeración original de las fuentes de donde se extrajeron. Dada la heterogeneidad en el formato de los trabajos aquí recogidos, se ha optado por un diseño poco intrusivo en busca de la simplicidad visual. No obstante, con el fin de ayudar a la localización de los artículos mediante la tabla de contenidos, se ha superpuesto un número de página propio de esta compilación. Igualmente, se ha procurado incluir la cita bibliográfica —no siempre completa, por lo cual pedimos disculpas— en la prime-

ra página de cada uno de los trabajos indicando, entre otros datos, el lugar y año de publicación.

A modo de descargo: la búsqueda ha sido dirigida por el contenido del CV de Isidro y la lista de publicaciones del grupo ISSI. A pesar de que en las búsquedas en la hemeroteca de la UCM se ha encontrado algún material no listado, no tenemos la absoluta seguridad de que en el libro estén todas las que son. Con respecto a la presentación, comprenderá el lector que no ha sido fácil realizar un proceso de digitalización plenamente satisfactorio, por lo que pedimos sinceras disculpas.

EL SABER Y EL SABER HACER EN INFORMÁTICA: UNA PERSPECTIVA HISTÓRICA

Prof. Dr. Isidro Ramos Salavert

Magníficos Sres. Rectores, autoridades, Profesores y alumnos, Sras. y Sres.

No he sabido resistir la tentación de traer al ánimo de todos nosotros, al recinto de saber que es el aula y que hoy se simboliza en este Salón, en este acto inaugural y solemne, las palabras de un Ingeniero ilustre, de un Artista reconocido, del hombre universal por excelencia: Leonardo da Vinci, cuando en su «Libro de Notas» en el «Apartado sobre Mecánica» escribe:

«La Ciencia es el Capitán, la Práctica los soldados»

O cuando en el Apartado sobre los errores de los que dependen de la práctica sin Ciencia, podemos leer:

«Los que se enamoran de la Práctica sin Ciencia son como el marino que se embarca sin Carta de Navegación o Brújula, nunca estará seguro de adónde va…»

Con el Corolario obvio:

«No existe mejor Práctica que una buena Teoría…»

Estas palabras fueron escritas en el Siglo XV, cuando el concepto de Teoría Formal, o la preocupación por la formalización del Lenguaje no habían aparecido en el horizonte de la

Ciencia y la Tecnología —y de hecho tardarían cuatro siglos en aparecer.

Nuestra lección se va a situar en un horizonte mucho más próximo históricamente (apenas cincuenta años), en una disciplina de todos conocida: la *Informática*.

Socialmente es una realidad el impacto de la Informática en la sociedad a través de la Ciencia, la Tecnología, la Técnica, y la Cultura ya que todos los aspectos están implicados. Para los Informáticos es claro el papel catalítico que está jugando en otras disciplinas científicas como la Lógica, el Algebra… o tecnológicas como la Electrónica.

El tema ha sido ya tratado exhaustivamente a nivel de la calle, periódicos, televisión, revistas generales y específicas se han hecho eco de lo que ha venido en llamarse «La Tercera Revolución Industrial», «la tercera ola» (siguiendo el sugerente término de G. Dörfles). Quizás se ha dicho ya todo, quizás sólo podamos hacer ya una síntesis de lo presentado y un resumen de lo publicado, pero todo ello con una idea en la mente: transmitir en qué consiste la Informática y cuál ha sido su evolución desde la perspectiva del saber y del saber hacer.

Hay una idea que por su globalidad me gustaría comentar: hablábamos antes de la Tercera Revolución Industrial, y dicho calificativo exige una explicación. En todo proceso de producción participan básicamente tres componentes: materias primas, energía y saber hacer. La Segunda Revolución Industrial supuso el que la máquina de vapor sustituyera al operador humano como fuente de energía versátil y controlable; y aunque el uso de la energía mecánica era ya una adquisición histórica: molinos de viento, norias, etc., este hecho en apariencia trivial tuvo efectos decisivos (incidencias sociales que tal concentración de energía supuso: la aparición de fábricas, de ciudades alrededor de ellas, medios de comunicación, etc.) que configuraron el devenir social.

La Informática incide en el tercer aspecto: el saber hacer, sustituyendo al hombre en tareas que antes le eran propias (que eran consideradas inteligentes), y que van convirtiéndose en rutinarias, exteriorizadas y soportadas en máquinas controladas por ordenadores, talleres flexibles, robots, etc.

Diremos siguiendo a Jean François Lyotard:

> *«El saber se ha convertido en los últimos decenios en la principal, fuente de producción»*

La *prótesis intelectual* que constituye para el hombre el ordenador va a incidir en un aspectos considerado hasta el momento como patrimonio específicamente humano: la creatividad, que va a ser modificada en su concepción, en el sentido de que la frontera difusa entre lo creativo y lo rutinario va a ir desplazándose paulatinamente hacia el primer aspecto convirtiéndose, cada vez más, en rutinarias y por lo tanto realizables por un ordenador, actividades inteligentes.

Para ello ha sido necesario comunicarse con el ordenador a través de lenguajes y soportar en él el conjunto de conocimientos al que antes aludíamos.

Citemos nuevamente al filósofo de la Postmodernidad Jean François Lyotard:

> *"[…] el saber científico es una clase de discurso. […] las ciencias y las técnicas llamadas de punta se apoyan en el lenguaje: […] las teorías lingüísticas, los problemas de comunicación, la Cibernética, las Álgebras Modernas y la Informática, los ordenadores y sus lenguajes, los problemas de traducción de los*

lenguajes y la búsqueda de compatibilidades entre lenguajes […], los problemas de memorización y las Bases de Datos, la Telemática y la puesta a punto de terminales inteligentes […]"

Sobre los aspectos: *lenguajes y conocimientos* versará nuestra lección.

La historia de la Informática es la historia de una gran equivocación, la historia de un gran meandro del pensamiento humano en el que se pierden treinta años en su avance, en su evolución… y en el que se llenan miles de papeleras con millones de líneas de programas en los que nadie confía ni puede utilizar y que han costado ingentes cantidades de dinero de producir. Centrémonos en este aspecto, el *Software*.

¿Pero qué es el Software? Una primera definición sería: la parte no material de un ordenador, es decir, el conjunto de programas. Luego veremos que muchas actividades lógicas se plasman en el hardware con máquinas orientadas a lenguajes, etc. Con ello la frontera entre *material* y *logical* aparece cada vez más difusa. El lenguaje de una máquina, el lenguaje máquina puede ser de alto nivel, podemos tener máquinas ALGOL, PASCAL, LISP, etc. Pero por el momento llamaremos *Logical* al conjunto de facilidades no materiales de un ordenador: útiles diversos, programas y desde la perspectiva de usuario: el conjunto de lenguajes, sistema operativo, y entorno del trabajo que encuentra el trabajar con él.

Para entender el meandro al que antes me refería y adquirir perspectiva histórica, vamos a recorrerlo brevemente. Su origen se sitúa a finales de los años cuarenta. Se construye el primer ordenador, desnudo, sin Software. La interfaz con el usuario, el lenguaje de programación, es el código binario. Quizás cargadores elementales realizan además, la decodificación de código octal o hexadecimal a binario. La construcción de un léxico mnemotécnico y fijo para las operaciones y la introducción de las variables (como se hacía en Matemáticas) obligará al diseño de ensambladores y con ello al comienzo del diseño de traductores, que sirvió de motor en los años sesenta a la teoría de Lenguajes, Estructuras de Información, Gestión de la Memoria, e Ingeniería del Software (puesto que hacer un compilador es un gran proyecto de Ingeniería). Pero si por este camino podríamos ya recorrer el meandro de la evolución del *Logical*, con los sucesivos lenguajes de programación y sus traductores asociados, vamos a abandonarlo momentáneamente para analizar el modelo de cálculo subyacente en los programas o frases construidas con dichos lenguajes para plantear problemas al ordenador.

El modelo usado es el de la máquina de estados, el autómata de estados finitos, o en general la *Máquina de Turing*: el ordenador pasa de un estado a otro por la ejecución de una acción descrita por una instrucción, un cálculo será una sucesión de acciones y un programa la secuencia de instrucciones que lo describen. Las acciones son elementales, las que por el material podían ser realizadas en aquellos momentos: intercambios memoria-registros, sumas, restas, y algunas instrucciones condicionales que permitían romper la pura secuenciación de dichas acciones dando lugar a bucles… Estamos en el origen de toda la programación imperativa con GOTO.

Con este modelo de cálculo (imperativo, secuencial), se aborda el problema de la Informática, que consiste en: dado un enunciado de un problema (luego veremos que obtener el enunciado de un problema es tenerlo prácticamente resuelto) y entendiendo el problema como un conjunto de tratamientos que queremos realizar sobre unos objetos, partir de los datos para obtener los resultados. Una primera etapa de análisis conduce este enunciado a un algoritmo en el que aparecen tanto las acciones (en número finito y ejecutables en tiempo finito), como los objetos formalizados sobre los cuales se realizan. Un paso posterior de representación, llamado programación, nos va a convertir dicho algoritmo y dichos objetos formalizados en un programa y en objetos del lenguaje de programación, (aportados por la parte declarativa del lenguaje)

para finalmente en una última etapa conocida como traducción o compilación (si hablamos de un lenguaje de programación), convertir el programa, el conjunto de instrucciones a instrucciones máquina (en lenguaje máquina) y el conjunto de objetos del lenguaje de programación a objetos en lenguaje máquina.

Es sintomático que hasta finales de los años sesenta con el lenguaje SIMULA, los objetos a tratar deban representarse directamente en las casillas de memoria o variables poco estructuradas de los primeros lenguajes de programación de alto nivel: el FORTRAN, el ALGOL, etc. Con ello el aspecto algoritmo es fuertemente dependiente de este bajo nivel de representación de los objetos. El *Logical* de la época es un continuo y confuso amasijo de representación de los objetos, de los algoritmos y de las deficiencias de los lenguajes de programación, limitados en sus sintaxis y semántica por los pocos conocimientos teóricos que en ellos se vertían. Se dice que el FORTRAN (o aún más el hoy tan difundido BASIC) es el primer lenguaje ensamblador independiente de la máquina.

El inclinar la balanza sobre el hacer sin contar con el saber va a conducir a la llamada crisis del Software. Pongamos como mecanismo exculpatorio la enorme presión social ejercida sobre la Informática, que ninguna tecnología había sufrido tanto y de forma tan apremiante hasta ese momento.

El *Logical* emprende el camino de los lenguajes imperativos, FORTRAN, ALGOL, COBOL, que definen muy bien los años sesenta, PL 1 define por su parte los años setenta. El ALGOL es el primer lenguaje que incorpora mecanismos constructivos eficientes de estructuración de algoritmos: recursividad, estructura repetitivas, etc. pero permanece al mismo nivel que el FORTRAN en cuanto a la representación de los objetos. Es el lenguaje SIMULA, (con el concepto de clase), el primero en el que se trata convenientemente el aspecto: «representación de objetos», aunque su difusión casi exclusivamente académica y limitada al ámbito de la simulación hacen que su impacto sea pequeño inicialmente.

Con la crisis del Software y los trabajos del Prof. Dijkstra (en particular: Notas sobre Programación estructurada), de Dahl sobre estructuras de datos, y de Wirth sobre diseño de un lenguaje específicamente orientado a enseñar a programar: el PASCAL, se plantea, en los años setenta, una primera reflexión sobre el camino equivocado en el que se estaba. El problema no era el lenguaje (solamente) sino los mecanismos de vencer la complejidad de que se disponía. El concepto de módulo que esconde la representación concreta va a ser aplicado tanto a los datos (definiéndose los tipos abstractos de datos) como a los algoritmos (módulos y su representación) que con el método descendete de D. E. Knuth y de refinamientos sucesivos, va a abrir el camino a la década de las Metodologías (los años setenta).

El saber hacer en *Logical* se va almacenando en bibliotecas de programas cada vez más difundidas y utilizadas, paquetes estadísticos, como el BMD, el IMSL, matemáticos como el Mathpack de simulación como el GPSS, gráficos, etc.

Por motivos aparentemente dispares (y en el marco de la Inteligencia Artificial) aparecen dos lenguajes que van a replantear esta tendencia y que van a conducir a una segunda reflexión, nos referimos al LISP de MacCarty y al PROLOG de Alain Colmerauer. El primero rompe con el concepto de semántica operacional basada en la máquina de estados, haciéndola apoyarse en los esquemas funcionales recursivos y el segundo sitúa el concepto de cálculo como la deducción dentro de una teoría formal de primer orden, las cláusulas de HORN definidas, con la metarregla de considerar la negación como el fracaso en una demostración finita. ¿En qué sentido afecta esto al *Logical*? ¿En qué sentido afecta esto al usuario de la Informática?

En PROLOG o en general en los lenguajes de la Programación Declarativa, el acento se va a situar en el QUÉ queremos resolver, (en el enunciado del problema) más que en el CÓMO queremos resolverlo. Sitúa el problema del *Logical*, de la comunicación con el ordenador, en el marco de la Lógica, del Algebra… (dejadas de lado en estos aspectos aunque usada asombrosamente en la verificación a posteriori de la corrección de los programas o en la definición axiomática de su semántica —pensemos en los trabajos de C.A.R. Hoare—) y en el diseño de circuitería dentro de la Teoría de la Conmutación. En este último caso se usa la Lógica como elemento de Análisis y Diseño de Circuitos desde un principio: no se Analiza y diseña un circuito, un ordenador, de cualquier manera y luego a posteriori se intenta describir su corrección, su funcionalidad usando herramientas formales… Las consecuencias de esta forma de proceder, en apariencia trivial, son enormes: el material es cada vez más potente, fiable, barato… mientras que el Software tiene exactamente las propiedades contrarias.

La segunda reflexión a la que antes aludíamos (y por supuesto los continuos fracasos técnicos y tecnológicos) hace que se vuelven los ojos hacia algo ya perfectamente conocido en los años cuarenta: la Lógica y el Algebra y se reconduce el diseño del *Logical* en este sentido. ¡Han pasado treinta años!

El modelo de cálculo va a consistir ahora en la demostración en el marco de una teoría formal y las máquinas empiezan a medir sus capacidades no ya en los millones de instrucciones por segundo que son capaces de ejecutar (los MIPS) sino en los millones de inferencias por segundo que realizan.

Pero una demostración en una teoría formal es un proceso altamente no determinista que obliga a la vuelta atrás cuando se ha tomado el camino equivocado. La búsqueda de demostradores automáticos de teoremas (que es el nuevo concepto de programa, es decir, teoría y demostración de un teorema equivalen a programa y ejecución de un programa), como decía: la búsqueda de demostradores automáticos de teoremas eficaces, lleva a simplificar el cálculo de predicados de primer orden eliminando conectivas. Con ello nos situamos en la Programación Lógica y sus derivados.

Pero en la programación funcional, en el LISP, un predicado tiene que representarse incorrectamente como una función y en programación lógica una función tiene que representarse, también incorrectamente, como un predicado. Esto invita a una aproximación mixta: el paso a un modelo de cálculo basado en el cálculo de predicados de primer orden con igualdad restringido, lo que permite la definición axiomática de funciones y una demostración eficaz mediante los sistemas de reescritura de términos, (en los que los problemas de finitud y unicidad se encuentran entre los campos de investigación teóricos del momento), y en los que el proceso de demostración se realiza vía algoritmos análogos a los de resolución de Alain Robinson. Tenemos un modelo mixto, con funciones definidas axiomáticamente a través de la igualdad y predicados en los que el usuario puede representar sus enunciados. Los objetos aparecen representados mediante tipos abstractos algebraicos con axiomas ecuacionales y los tratamientos mediante términos y ecuaciones del algebra heterogénea que es el tipo abstracto, y todo ello dentro de un marco formal.

Nos estamos refiriendo a los lenguajes de especificación Lógico-algebraicos actuales como OBJ, RAP, etc... que con su evolución están permitiendo además modificar el ciclo clásico (secuencial) de producción del *Logical*, de producción de programas introduciendo una posibilidad nueva en este proceso: la prototipación del software que permite validar el análisis desde las primeras fases del proceso. Esto que es una actividad normal en cualquier ingeniería, en Informática aparece como una revolución hace sólo unos pocos años… El software además puede ser generado automáticamente.

Toda esta crisis del modelo imperativo, basado en la máquina de Estados, va a incidir en el diseño del material con criterios software, y las máquinas que realizan demostraciones, motores de inferencia que reescriben, rompen además la secuenciación temporal en su operación. Aparecen nuevas arquitecturas: el paralelismo explotado totalmente (buscando la eficacia) huyendo del modelo elemental del autómata de estados finitos que yacía en la máquina de Von Neuman. La aparición de las máquinas de flujo de datos, etc.

Estamos en nuestro recorrido en los años ochenta. La Lógica ha acercado al lenguaje humano, para cuya formalización se creó, la comunicación con el ordenador, la sintaxis se ha suavizado y nos encontramos en un contexto semántico conocido: la Lógica (Teoría de Modelos) y el Algebra.

La pragmática se ha suavizado, también, mediante entornos amigables de interacción con el ordenador (vía ventanas, ratón, menús,...). El proceso del lenguaje natural permite interrogar bases de datos en lengua natural, el reconocimiento de la palabra y su generación automática consiguen logros restringidos pero espectaculares, el tratamiento de imágenes sigue un camino paralelo.

La interacción del hombre con el ordenador ha replanteado, y ha obligado a profundizar en el concepto de lenguaje, en su sintaxis, semántica, y pragmática y ha llevado a tratar formalmente cada uno de estos aspectos, dado que la comprensión de las frases (programación o especificación) la ha de realizar inicialmente el hombre al pensarlas y redactarlas en un lenguaje de programación imperativo o de especificación descriptiva, después la máquina al tener que entenderlas a través de un proceso de compilación o de interpretación y finalmente la máquina al tener que contestarlas en el contexto de ese lenguaje, y todo ello huyendo del efecto lateral de lo que es (era) un ordenador, que implicó la forma de hacer y que determinó las condiciones de entorno del problema de la comunicación y el gran meandro al que antes nos referíamos.

El uso interactivo del ordenador ha modificado también la forma de comunicar con él, el tiempo de respuesta real ha influido cualitativamente en el diálogo posible.

Si los lenguajes han evolucionado hacia entornos de especificación y hacen uso de la Lógica y el Algebra como herramientas naturales, la representación del conocimiento, de los hechos, que modelizan una situación real (el universo del discurso) ha sufrido también una evolución paralela. De los ficheros de datos con estructura de acceso determinadas por el hardware de los años setenta, pasamos a las bases de datos estáticas, independientes del material y del tipo de la aplicación. Calificamos de Bases de Datos Estáticas aquellas que sólo permiten recuperar la información que en ellas se almacena explícitamente. Las Bases de Datos deductivas almacenan además de hechos, reglas de deducción que permiten obtener nuevos hechos deducidos de los explícitos, además de los almacenados.

La representación del conocimiento mediante Teorías o Metateorías Lógicas está abriendo el paso a Sistemas Expertos que son algo más que buenas argucias de programador.

La evolución del Software, lenguajes y conocimientos y sus tratamientos y bases asociados, tienden a incorporar de forma inmediata resultados teóricos de los sistemas de reescritura, de sistemas deductivos más eficaces, en un entorno de multiplicación de la potencia, de la velocidad, de la capacidad de memoria y de una pragmática de los lenguajes más fácil, usando entornos integrados de programación o especificación.

Aspectos tecnológicos como el tratamiento de gráficos, del color, de las pantallas de alta resolución que posibilitan una interacción a través del ratón, de las ventanas, de los menús, el reconocimiento y generación de la palabra, los editores orientados a sintaxis, los decompiladores… hacen todavía más amigable la pragmática del *Logical*.

Y todo esto ¿qué supone desde nuestra perspectiva tecnológica, de ingenieros de la Informática? Supone abordar en forma precisa y eficiente la producción de productos hasta ahora de calidad escasa: los programas (cuando deberían ser productos con una calidad de producción controlada). Y ello siguiendo un proceso análogo al seguido por el Arquitecto que para construir una casa elabora minuciosamente los planos antes de poner ningún ladrillo o del ingeniero de CCP cuando calcula un puente antes de remover el terreno.

Citemos, en este sentido, a dos de los más grandes informáticos mundiales, B. Burstall y J. Goguen (1981):

> *"If you want someone to build a house for you, it is wise to employ an architect to make sure that what gets built is what you want. The architect will prepare a detailed specification of the house which you can discuss and amend before anyone starts laying bricks. In the same way before anyone invests time and emotional energy in writing a program it is as well to have a specification to make sure that what is produced is what is required."*

La Informática reivindica un hacer serio, tecnológico, basado en la teoría (el saber) y en la experiencia (el saber hacer), reivindica su *status* como una Ingeniería, quizás más moderna que las demás pero no por ello menos exigente en sus presupuestos teóricos, metodológicos… y no con menos impacto económico y social. E igual que cuando se habla del fleje de una viga (¡de una simple viga!) se hace uso del cálculo diferencial sin que se acuse de teoricista al que realiza el cálculo de estructuras, el informático cuando hable de una cuenta corriente en un banco, de una nómina tendrá que decir, usando la Matemática de la Informática que se modelizan adecuadamente mediante una teoría algebraica de un cierto tipo (una presentación) sin que por ello sea acusado de teoricista…

Resumiendo, pues, la evolución del *Logical* se ha producido en el sentido de facilitar una sintaxis más cómodo y próxima al lenguaje humano, una semántica no imperativa (preocupada por el QUÉ y no por el CÓMO) y una pragmática mucho más fácil mediante entornos de programación y especificación, y un mayor equilibrio entre el saber y el saber-hacer.

La evolución de la Informática ha replanteado el concepto de lenguaje: su sintaxis, su semántica y su pragmática y también el concepto de conocimiento a través de las Bases de Datos deductivas, de las Redes Semánticas y ello sumergido en nuevas técnicas de resolución de problemas, en nuevas teorías y mayores facilidades tecnológicas que sistematizan y facilitan el uso del ordenador y sobre todo ha replanteado el status del saber.

La sistematización de la creatividad es el objetivo de la Informática y marcará en el futuro su evolución.

El nivel informático de una comunidad no se mide por la cantidad de divisas que se gasta importando grandes ordenadores (normalmente subutilizados cuantitativa y cualitativamente) sino por la cantidad de divisas que gana exportando productos de calidad fiable: por ejemplo programas… y esto sólo se consigue con un buen-hacer tecnológico basado en el saber y en el saber-hacer, tema con el que la UCLM tiene un compromiso irrenunciable.

Y ahora en los 90 ¿qué ocurre?, estamos en la tercera reflexión: la Red, el Web vuelven a cuestionar todo lo dicho con anterioridad. Todo producto informático no basado en la red, distribuido, con estructura cliente-servidor, accesible amigablemente mediante páginas Web… es obsoleto. Las bibliotecas digitales, el comercio electrónico, la enseñanza cooperativa, la Tele-

medicina… abren un futuro apasionante. Dicho futuro es el presente de los jóvenes estudiantes que se agolpan en nuestras aulas, a las que acuden en búsqueda de unos conocimientos cuya carencia los dejaría huérfanos de la Realidad. Definitivamente descabalgados del Progreso. Esta es nuestra labor.

HE DICHO.

ÍNDICE GENERAL

VOLUMEN 1: MADRID, BILBAO, VALENCIA

Prólogo i

Días de hemeroteca v

… y noches de ordenador ix

Estructura del libro xi

El saber y el saber hacer en informática: una perspectiva histórica xiii

Índice general xxi

PARTE I: MADRID (1968-1974)

Prólogo de Roberto Moya Quiles 3

Isidro Ramos y el Centro de Cálculo de la Universidad de Madrid, por Ernesto García Camarero ex-director del CCUM 5

Seminarios del Centro De Cálculo De la Universidad Complutense de Madrid. 1968. 11

Gil Gayarre, M., Ramos, I., et al., Posibilidades de la Automatización en la corrección de exámenes. Revista de Automática. Año III, nº 7. 1970. 109

Fernández Flores, I. Ramos I., Traductor 70/13. Boletín CCUM. Volumen 10. 1970. 123

Moya R., Ramos, I. Aplicación de Hash-Coding al tratamiento automático de Gramáticas". Boletín CCUM. nº 17. 1971 137

Fernández Flórez, I., García Camarero, E., Ramos, I. A Basic Language Oriented to secondary School. Information Processing 71. 1972. 147

García Camarero, E., Ramos, I. Ensayo sobre la definición automática de tratamientos clínicos. Boletín CCUM. 1971. 151

Ramos, I. Lenguajes para la escritura de traductores. Revista de Automática. Enero-Marzo 1973. 163

Alfonseca, M., Ramos I. Un analizador sintáctico para un subconjunto de PL/I. Actas del Congreso Nacional de Automática. 1972. 169

Ramos, I. Un ejemplo de recursividad: HANOI. Boletín CCUM. nº 24. 1974. 205

Ramos, I., Lozano, C. Muñoz, A. Sistema de Creación y Mantenimiento de Bancos de Información. Revista de Automática, 1974 (Parte I), 1975 (Partes II y III). 211

PARTE II: BILBAO (1975-1983)

Prólogo de José Ramón Gariagoitia 255

Ramos, I. Système pour la création et gestion des banques de données. Seminaires Institut de Programmation (Université de Paris VI). 1974. 257

Moya R., Ramos I. Metaprogramación Estructurada. 3er Congreso Nacional de Informática y Automática. 1975. 263

Ramos, I., Moya, R., Martín, P. Evaluador Sintáctico del Lenguaje de Control de Comandos del Sistema Operativo EXEC-8. 3er Congreso Nacional de Informática y Automática. 1975. 289

Ramos, I. Aramberri, J. Cross-Macroassembler for Micro MCS-80 in a HP 2116-B Minicomputer. Euromicro. 1976. 301

Ramos, I. Informátique de Base: These, Antithèse, Synthèse. Convention Informatique. 1976. 309

Garitagoitia, J.R., Ramos, I. Relocating Direct Linking Loader for MCS-8 Microcomputer Series. Proc. of COMPCON 77. 1977. 313

Ramos, I. Sistema Editor / Perforador de textos. Revista de Automática e Informática. Volumen: nº 31. 1977. 319

Moya, R. Ramos, I. LPE: Un lenguaje en español para la programación estructurada. Proceso de Datos, no. 70, 1977. 325

Moya, R., Ramos I. SPL (Structured Programming Language). A language for structured programming. Proc. Of the 5th Annual III Conference. 1977. 331

Ramos, I. Aproximación prospectiva al software de base. Actas de la Convención Informática Latina. 1977. 339

Ramos, I., Aramberri, J. y Mellado, M. Modelo de realización de primitivas de sincronización para sistemas multimicroprocesador. Actas del 75 Aniversario Real Sociedad Española de Física y Química. 1978. 345

Aramberri, J., Ramos, I., Garitagoitia, J. R. Diseño y desarrollo de un sistema microprocesador. Actas I Jornadas sobre microprocesadores. 1978. 347

Moya, R., Ramos, I. SP Gramars (Context Free Gramars for Structured Programming). ACM SIGPLAN. Vol.4 Nº1. 1979. 357

Aramberri, J, Ramos, I., Mellado, M., Garitagoitia, J. R. Modelo de Sistema Microprocesador. Actas del 4º Congreso de Informática y Automática. 1979. 365

Sopeña, L., Ramos, I. Sistema de control en tiempo real de una red viaria. Actas del 4º Congreso de Informática y Automática. 1979. 375

Aguirre, J., Ramos, I. Sistema de programación estructurada con vuelta atrás. Actas del 4º Congreso de Informática y Automática, Madrid, 1979. 385

Ramos, I. Garitagoitia, J.R., Aramberri, J. Algunas soluciones al problema de interbloqueo entre procesos. 4º Congreso de Informática y Automática. 1979. 391

Guaresti, J.M., Garitagoitia, J.R. y Ramos I.A software chronological state controller. Actas del 4º Congreso de Informática y Automática. 1979. 403

Ramos, I. Calculadora de mesa. Un proyecto de clase sobre teoría de Lenguajes y Autómatas. Revista de Informática y Automática. 1979. 415

Zuloaga, J.L., Santamaria, R., Emaldi, R., Ramos, I. Sistema de Control lógico programable sobre microordenador. Revista de Informática y Automática. 1979. 421

Garitagoitia, J.R., Martinez,C., Ramos, I. MIKBUS: un bus para placas de microcomputadores cooperantes. Actas del II Simposio Español de Informática Repartida. 1982. 429

Garitagoitia, J.R., Ramos, I. Algoritmo de prevención para un sistema de gestión de ficheros. Actas de la Convención Informática Latina. 1983. 439

PARTE III: UNIVERSITAT DE VALÈNCIA - UNIVERSIDAD DE CASTILLA-LA MANCHA (1978-1987)

Prólogo de Francisco José Quiles Flor 455

Ramos, I. Semántica para SPL. Questió. Vol. 4. nº 1.1980. 459

Ramos, I. Un lenguaje de Programación con tipos Abstractos interpretado en PROLOG. Otra forma de compilar. Revista de Informática y Automática. 1984. 467

Ramos, I. Especificación de Pantallas. Revista de Informática y Automática, 1984. 471

Marín, J, Ramos, I. Metodología para la adquisición de material informático. Actas de las Jornadas de Informática en la Administración Local. 1984. 475

Ramos, I. The impact of new tecnologies on University organization: the example of Castilla-La Mancha. C.R.E. INFORMATION. Volumen: nº 70. 1985. 491

VOLUMEN 2: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

PARTE IV: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

Prólogo de María Alpuente 527

Celma, M., Moreno, L., Pastor, O., Ramos, I. Implementación de un sistema de ges-

tión de bases de datos deductivas. Revista de Informática y Automática, Vol. XX, Nº 3. 1987. 529

Alpuente, M., Botti, V., Galiano, I., Sanchis, E., Vila, F., Ramos, I. Entorno de programación LALI implementado en PROLOG. Revista de Informática y Automática . Vol. XX, Nº 4. 1987. 537

González del Río, A., Alpuente, M., Casamayor, J.C., Pastor, M.A., Ramírez, M.J., Ramos, I. Protolog : A conceptual Schema Facility for Automated Prototype Generation. 7th. IASTED International Symposium. 1989. 543

Ramos I., Canós, J.H., Forradellas, R., Oliver, J. A conceptual schema specification system for rapid prototyping. 8th IASTED International Symposium. Austria. Feb. 1990. 547

Ramos, I. Logics and OO-Data Bases: a declarative approach. Proc. International Conference on Database and Expert Systems Applications (DEXA 90), 1990. 551

Ramos, I.; Pastor, O.; Casado, V. OO and Active Formal Information System Specification. Proc. International Conference on Database and Expert System Applications (DEXA 91) Vol. II. 1991. 557

Pastor, O. Ramos, I. Canós, J.H. Object Oriented and Relational Specification of Information Systems. Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS. 1991. 563

Ramos, I., Cuevas, J., Devesa, J., Pastor, O., Canós, J.H. On the use of Algebras as Semantic domains of Object Societies. Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS. 1992. 579

Alpuente, M., Ramírez, M. J. An Equational Constraint Logic approach to Conceptual Modelling. In Ramos, I. and Tjoa, A. (eds), Proc. of the International Conference on Database and Expert Systems Applications (DEXA 92), 1992. 589

Ramos, I., Pastor, O., Cuevas J., Devesa , J. Objects as Observable Processes. Proc. of the 4th International Workshop of the Deductive Approach for DB and IS.1993. 595

Pastor, O., Ramos, I., Canós, J.H. From Analysis to Design in an OO and Deductive Environment. Proc of the International Conference on Database and Expert System Applications (DEXA´93). 1993. 611

Toval, A., Ramos, I. , Pastor, O. Prototyping Object Oriented Specifications in an Algebraic Environment. Proc. Of the International Conference on Database and Expert System Applications (DEXA'94), 1994. 617

Devesa, J., Cuevas, J., Ramos, I. Implementation of a Term Rewriting System for solving process queries in an object-oriented environment". Proc. Of the GULP-PRODE 94. 1994. 627

Canós, J. H., Penadés, M.C., Ramos, I., Pastor, O. A Knowledge-Base Architecture for Object Societies. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 643

Pastor, O., Ramos, I., Canós J.H. OASIS v2: A Class Definition Language. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 649

Ramos, I., Toval, A., García-Molina, J., Flores, M., González, M., Fernández, J. L., Nicolás, J., Pastor, O., Canós, J. H., Pelechano, V., Penadés, M.C., Bonet, B. Entorno General de Produccion automática de prototipos de software orientado a objetos: OOAP (Object-Oriented Assistant Prototyper). Actas de las I Jornadas de Informática. Asociación Española de Informática y Automática (AEIA). 1995. 659

Pastor O., Pelechano V., Bonet B., Ramos I. An object Oriented Approach for making Automated Prototyping Feasible. Proc. of the International Conference on Database and Expert System Applications (DEXA 96). 1996. 673

Carsí, J.A., Ramos, I., Penadés, M.C., Pelechano, V., Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic. Actas de las I Jornadas de Trabajo en Ingeniería del Software, 1996. 683

Penadés, M.C., Ramos, I., Canós, J.H., Tolsá, J., Ferrer, J. Análisis, diseño e implementación de un Entorno de Prototipación Automática. Actas de las I Jornadas de Trabajo en Ingeniería del Software,1996. 695

Sánchez, P. Letelier, P., Ramos, I. Representación de Objetos OASIS como Teorias y Traducción a un Modelo de Concurrencia. Computación y Sistemas, Vol. I, Nº. 2. 1997. 707

Lozano, M.D., Ramos, I. Evolución y perspectivas en el desarrollo de software: nuevas tendencias orientadas a objetos. Ensayos: revista de la Escuela Universitaria de Magisterio de Albacete, Nº. 12. 1997. 719

Sánchez, P., Letelier, P., Ramos, I. Constructs for Prototyping Information Systems using Object Petri Nets. Proc. of the IEEE International Conference on System, Man and Cybernetics.1997. 733

Lozano, M.D., Ramos, I., Cuenca, P. Application of an Object-Oriented Methodology for Automatic Software Production : OO-Method. Proc. Of the IEEE PACRIM 97. 1997. 739

Letelier, P., Sánchez, P., Ramos, I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitios. Proc. of the Argentine Symposium on Object Orientation. 1997. 743

Letelier, P. Sánchez, P., Ramos. I. Animation of Systems Specifications Using Concurrent Logic Programming. Proc. of the Symposium in Logical Approaches to Agent and Modelling and Design (ESSLLI´97). 1997. 751

Carsí, J.A., Canós, J.H., Ramos, I. Un intérprete de Transaction Frame Logic. Actas de las II Jornadas de Trabajo en Ingeniería del Software. 1997. 759

Tolsá, J.M., Ferrer, J., Penadés, M.C., Canós, J.H., Ramos, I. Meta-KAOS. Un metaentorno de prototipación automática. Actas de las II Jornadas de Trabajo en

Ingeniería del Software. 1997. 773

Ramos, I.,Canós, J.H., Pastor,O.+ investigadores proyecto MENHIR. MENHIR: Modelos, Entornos y Nuevas Herramientas para la Ingeniería de Requisitos. Actas de las III Jornadas sobre Tecnología de Objetos. 1997. 787

Canós, J.H., Lozano, M.D., Ramos, I. KAOS: An Object-Oriented Software Tool for the Objects Definition, Updating, Querying and Programming in an Object-Oriented Environment. Proc. of the IEEE Canadian Conference on Electrical & Computer Engineering.1998. 799

Letelier, P., Sánchez, P., Ramos, I. Especificaciones de proceso para objetos y su representación en Lógica Dinámica. Actas de las III Jornadas de Ingeniería de Software. 1998. 803

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto. Actas de las III Jornadas de Ingeniería del Software. 1998. 813

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a las pautas de representación de OASIS en un formalismo gráfico. Actas de las III Jornadas de Ingeniería del Software. 1998. 827

Carsí, J. A., Camilleri, S., Canós, J.H., Ramos, I. Propuesta de Interfaz Gráfica de Usuario homogénea para el diseño y la explotación de Sistemas de Información. Actas de las III Jornadas de Ingeniería del Software. 1998. 839

Letelier, P., Sánchez, P., Ramos, I. Un ambiente para especificación incremental y validación de modelos conceptuales. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99).1999. 853

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99). 1999. 867

Letelier, P., Sánchez, P., Ramos, I. Prototyping a requirements specification through an automatically generated concurrent logic program. Proc. of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99). 1999. 879

Pastor,O., Canós,J.H., Ramos,I. From CASE to CARE (Computer-Aided Requirements Engineering). Proc. of the International Conference on Conceptual Modeling (ER'99). 1999. 895

Lozano, M., Ramos, I. Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. Workshop on Integrating Human Factors into Use Cases and OO Methods at ECOOP'99. 1999. 911

Letelier, P., Sánchez, P., Ramos, I. Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales". Actas del VI Congreso Internacional de Investigación en Ciencias Computacionales (CIICC'99). 1999. 917

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. Modelado conceptual con un lenguaje

formal y orientado a objeto. Actas de la XXV Conferencia Latinoamericana de Informática (CLEI'99). 1999. 929

Letelier, P. Sánchez, P., Ramos, I. Animation of Conceptual Models using two Concurrent Environments: An overview. Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers. 1999. 943

Parets, J., Carsí, J., Canós, J.H., Anaya, A., Hurtado, M.V., Penadés, M.C., Paderewski, P., Ramos, I., Rodriguez, M.J., La evolución de modelos en el desarrollo de software: los enfoques MEDES y OASIS. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos. 1999. 951

Sánchez, P., Letelier, P., Ramos, I. Una Interpretación de la Herencia de OASIS en Lógica Dinámica. Computación y Sistemas, vol. 4, núm.4. 2000. 963

Sánchez, P., Letelier, P., Ramos, I. Animating formal specifications with Inheritance in a DL-based framework. Requirements Engineering Journal, vol.4, no.4. 2000. 973

Anaya, R., Ramos, I. Un lenguaje formal para especificación y refinamiento de Objetos de Negocio. Ingeniería del software y bases de datos: tendencias actuales. Universidad de Castilla-La Mancha, 2000. 995

Anaya, R., Ramos, I. AR2CA: Una Herramienta para la Construcción de componentes reutilizables a través de niveles de refinamiento. Actas de 3as Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software (IDEAS2000). 2000. 1011

Carsí, J. A., Ramos, I., Canós, J.H. Reflection in the OASIS object model to solve Schema Evolution Problems. Proc. of the International Symposium on the Principles of Software Evolution (ISPSE2000). 2000. 1025

Sánchez, P., Letelier, P., Ramos, I. Validation of Conceptual Models by Animation in an Scenario based approach. Workshop 'Scenario-based round-trip engineering' at OOPSLA 2000. 2000. 1031

Giret, A., Ramos, I.Full software agents as OASIS 3.0 objects. Proc. of the 2nd Constraint) Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2001. 2001. 1037

Letelier, P., Ramos, I. Trazabilidad de Requisitos y Calidad del Software: Una Aproximación basada en UML. En Piattini y García (Eds.), Calidad en el Desarrollo y Mantenimiento del Software. Ra-MA, 2002. 1053

Pérez, J., Carsí, J. A., Ramos, I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR) at ICSM'02. 2002. 1075

VOLUMEN 3: ARQUITECTURAS SOFTWARE (2002-2010)

PARTE V: ARQUITECTURAS SOFTWARE (2002-2010)

Prólogo de Patricio Letelier 1109

Ferrer, J., Lorenzo, M. A., Ramos, I.,Carsí, J. A., Pérez, J. Modeling Dynamic Aspects in Architectures and Multiagent Systems. Proc. of the Constraint Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2002. 2002. 1111

Pérez, J., Ramos, I., Lorenzo, M. A., Letelier, P., Jaén, J. PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos. Actas de las VII Jornadas de Ingeniería del Software y Bases de Datos. 2002. 1125

Ferrer, J., Lorenzo, M. A., Ramos, I., Carsí, J. A. Dinamismo en arquitecturas y sistemas multiagente. Actas de las V Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2002). 2002. 1137

Jaén, J., Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures. Proc. of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03). 2003. 1147

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. Proc. of the 3rd IEEE International Conference on Quality Software (QSIC 2003). 2003. 1153

Pérez, J., Ali, N., Ramos, I., Carsí, J. A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes. Workshop Desarrollo de Software Orientado a Aspectos (DSOA). Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1161

Pérez, J., Ali, N., Ramos, I., Pastor, J. A., Sánchez, P., Álvarez, B. Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1167

Lorenzo, M. A., Peñarrubia, A., Carsí, J.A., Ramos, I. Modelando con UML Arquitecturas Software Reflexivas desde la Aproximación Orientada a Aspectos: Un Caso de Estudio. Actas de las VI Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2003). 2003. 1177

Navarro, E., Ramos, I. Requirements and Architecture: a marriage for Quality Assurance. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1189

Silva, J., Ali, N., Carsí, J. A.,Ramos, I. El aspecto de distribución de PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1199

Ali, N., Silva, J., Jaén, J., Ramos, I., Carsí, J. A., Pérez, J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proc. of the 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2003). 2003. 1209

Pérez J., Ali, N. Carsí, J. A., Ramos, I. Navarro, E. Designing Software Architectures with an Aspect-Oriented Language. ADVICE: Journal of Aspect Orientation, Vol. 1, No. 1, 2004. 1217

Navarro, E., Letelier, P., Ramos, I. Goals and Quality Characteristics: Separating

Concerns. Proc. of Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004. 2004. 1239

Navarro, E., Letelier, P., Ramos, I. UML Visualization for an aspect and goal oriented approach. Proc. of the 5th Aspect-Oriented Modeling Workshop (AOM'04) at UML 2004 Conference. 2004. 1247

Navarro, E., Ramos, I., Pérez, J. Goals Model Driving Software Architecture. 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004). 2004. 1255

Ali, N., Pérez, J., Ramos, I. High Level Specification of Distributed and Mobile Information Systems. Proc. of the Second International Symposium on Innovation in Information & Communication Technology (ISSICT2004). 2004. 1263

Ali, N., Carsí, J. A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2004. 1277

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004). 2004. 1285

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Orientación a aspectos y Orientación a objetivos: una propuesta para su integración. Taller de Desarrollo de Software Orientado a Aspectos, junto a JISBD'2004. 2004. 1293

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Dynamic Evolution in Aspect-Oriented Architectural Models. European Workshop on Software Architecture. 2005. 1299

Pérez, J., Ali, N., Costa, C., Carsí, J. A., Ramos, I. Executing Aspect-Oriented Component-Based Software Architectures on . NET Technology. Proc. of the 3rd International Conference on .NET Technologies. 2005. 1317

Ali, N., Ramos, I., Carsí, J. A. A Conceptual Model for Distributed Aspect-Oriented Software Architectures. Proc. of the International Conference on Information Technology Coding and Computing, ITCC 2005. 2005. 1329

Limón, R., Torres, J., Ramos, I. Separación de "Crosscutting Concerns" en el Desarrollo Orientado a Aspectos a Nivel de Diseño Arquitectónico. Actas del 6º Simposium Iberoamericano de Computación e Informática (SICI'05). 2005. 1335

Navarro, E., Letelier, P., Ramos, I. Integrating Expressiveness of Modern Requirements Modeling Approaches. Proc. of the 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005). 2005. 1345

Costa, C., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1355

Cabedo, R., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Implementación Orientada a Aspectos en C# de un sistema robótico tele-operado. III Taller de Desarrollo de Software Orientado a Aspectos de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2005. 1363

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1371

Navarro, E., Letelier, P., Mocholí, J. A.,Ramos, I. A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems, XLVI(5), Special Issue on Systems Analysis and Design. 2006. 1379

Cabello, M. E., Costa, C., Ramos, I.,Carsí, J. A. Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System. Journal of the WSEAS Transactions on Information Science and Applications, Issue 10, Volume 3. 2006. 1391

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. Proc. of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006). 2006. 1399

Pérez, J., Navarro, E., Letelier, P., Ramos, I. A Modelling Proposal for Aspect-Oriented Software Architectures. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). 2006. 1415

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. A Goal-Oriented Approach for Safety Requirements Specification. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06). 2006. 1425

Limón, R., Ramos, I., Torres, J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development. Proc. of the International Conference the Computational Science and Its Applications (ICCSA 2006) . 2006. 1433

Limón, R., Ramos, I. Using Styles to Improve the Architectural Views Design. Proc. Of the International Conference on Software Engineering Advances (ICSEA 2006). 2006. 1443

Cabello, M. E., Costa, C., Ramos, I., Carsí, J. A. Generic Architecture of an Expert System for Diagnosis. Proc. of the 10th WSEAS International Conference on Computers. 2006. 1449

Cabello, M. E., Ramos, I., Costa, C. Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico. Actas del XIX Congreso Nacional y V Congreso Internacional de Informática y Computación del ANIEI-CIICC´07. 2006. 1455

Ali, N., Millan, C., Ramos, C. Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model. Proc. of the 8th International Sym-

posium on Distributed Objects and Applications (DOA). 2006. 1463

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes. IEEE Latin America Transactions, Special Edition JISBD'06, vol. 5, issue 4. 2007. 1481

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Mobile Ambients in Aspect-Oriented Software Architectures. Proc. of the IFIP Working Conference on Software Engineering Techniques - SET 2006. 2006. 1489

Navarro, E., Letelier, P., Reolid, D., Ramos, I. Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques. Proc. of the 15th International Conference on Information System Development. 2006. 1501

Navarro, E., Letelier, P., Ramos, I. Requirements and Scenarios: running Aspect Oriented Software Architectures. Proc. of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). 2007. 1515

Limón, R., Ramos, I. Analyzing Styles of the Modular Software Architecture View. Proc. of the First European Conference on Software Architecture (ECSA 2007). 2007. 1521

Limón, R., Ramos, I. Relating Software Architecture Views by Using MDA. Proc. of the ICCSA 2007 International Conference on Computational Science and its Applications. 2007. 1525

Limón, R., Ramos, I., Morales, M. Zárate, J. Analyzing and Designing Software Architecture Views driven by their Relationships. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 1537

Pérez, J., Costa, C., Carsí, J. A.,Ramos, I. Verificación de Modelos Arquitectónicos Orientados a Aspectos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos. 2007. 1547

Costa, C., Ali, N., Pérez, J., Carsí, J. A., Ramos, I. Dynamic Reconfiguration of Software Architectures Through Aspects. Proc. of the First European Conference on Software Architecture (ECSA'07). 2007. 1557

Limón, R., Cabello, M. E., Ramos, I. Establishing Relations among Software Architecture Views through MDA for SPL. Proc. of the 14th International Congress on Computer Science Research (CIICC'07). 2007. 1563

Pérez, J., Ali, N., Carsí, J. A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J. A. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Journal of Information and Software Technology 50 (2008) . 2008. 1575

Ali, N., Solís, C., Ramos, I. Comparing architecture description languages for mobile software systems. Proc. of the First International Workshop on Software Architectures and Mobility at ICSE'08. 2008. 1597

Navarro, E., Gómez, A., Letelier, P., Ramos, I. MORPHEUS: a supporting tool for MDD. Proc. of the18th International Conference on Information Systems

Development (ISD2009). 2009. 1603

Ali, N., Ramos, I., Solís, C. Ambient-PRISMA: Ambients in Mobile Aspect-Oriented Software Architecture. Journal of Systems and Software 83. 2010. 1615

Pérez, J., Ramos, I., Carsí, J.A., Costa, C.: Model-Driven Development of Aspect-Oriented Software Architectures. Journal of Universal Computer Science, vol. 19, no. 10. 2013. 1637

VOLUMEN 4: INTERFACES DE USUARIO (2000-2002) - GESTIÓN DE MODELOS (2002-2006)

PARTE VI: INTERFACES DE USUARIO - IDEAS (2000-2002)

Prólogo de María Dolores Lozano 1701

Lozano, M.D., Ramos, I. An Object Oriented Approach for the Specification of User Interfaces. Proceedings of PACRIM'99 Conference. 2000. 1703

Lozano, M. D., Ramos, I., González, P. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development". Proc. of the IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000). 2000. 1707

Lozano, M. D, González, P., Ramos, I. Models and Ontologies for the Specification of User Interfaces. Proc. of the Workshop on Model Engineering at ECOOP'00, 2000. 1717

Lozano, M. D., Ramos, I. Aproximación Basada en Modelos en la especificación de Interfaces de Usuario. Actas de las I Jornadas de Interacción Persona-Ordenador (INTERACCIÓN-20), 2000. 1725

Lozano, M. D., González, P., Ramos, I. User Interface Generation. Current Trends. INFORMATIK /INFORMATIQUE. Journal of the Swiss informatics societies. Vol. 2. 2001. 1733

Lozano, M. D., González, P., Ramos, I. IDEAS: An Object-Oriented Environment for developing User Interfaces". Proc. of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01). 2001. 1739

Lozano, M. D., González, P., Ramos, I., Montero, F., Molina, J. P. Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.16.2002. 1749

Lozano, M., González, P., Ramos, I. A Graphical User Interface Development Tool. 16th British Computer Society Conference on Human-Computer Interaction (BHCI'02). 1759

Lozano, M., González, P., Montero, F. Molina, J. P., Ramos, I. Integrating Usability within the User Interface Development Process of Web Applications. Proc. of the 2nd International Workshop on Web Oriented Software Technology at ECOOP'02. 2002. 1763

Montero, F. Lozano, M. D., González, P., Ramos, I. Designing Web Sites by Using Design Patterns. Proc. of SugarLoafPLoP 2002. 2002. 1779

PARTE VII: GESTIÓN DE MODELOS

Prólogo de José Ángel Carsí 1805

Carsí J.A, Ramos I., Silva J., Pérez J., Anaya V. Un Generador Automático de Planes de Migración de Datos. I+D Computación, Vol. 1, No. 1. 2002. 1809

Silva J.F., Carsí J.A., Ramos I. Comparación automática de esquemas conceptuales orientados a objeto, Revista Ingeniería Informática - Revista electrónica del Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción - Chile. 2002. 1821

Pérez J., Ramos I., Anaya V., Cubel J.M, Domínguez F., Boronat A., Carsí J.A. Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. Proc. of the Workshop on Software Evolution Through Transformations (SET'02) at ICGT2002. 2002. 1833

Silva, J., Carsí, J. A., Ramos, I. An algorithm to compare OO-Conceptual Schemas. Proc. of the International Conference on Software Maintenance (ICSM'2002). 2002. 1847

Pérez J., Carsí J.A, Ramos I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR'2002) at the ICSM'02. 2002. 1857

Pérez, J., Carsí, J. A., Ramos, I. ADML: A Language for Automatic Generation of Migration Plans. Proc. of the First Eurasian Conference on Advances in Information and Communication Technology. 2002. 1869

Boronat, A., Carsí, J. A., Ramos, I. Una plataforma semántica para la gestión de modelos. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos. 2003. 1879

Boronat, A., Pérez, J., Carsí, J. A.,Ramos, I. Two experiencies in software dynamics. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. 2004. 1889

Boronat, A., Carsí, J. A., Ramos, I. An Algebraic Baseline for Automatic Transformations in MDA. Proc. of the Workshop on Software Evolution Through Transformations at ICGT2004. 2004. 1919

Boronat, A., Pedrós, J., Carsí, J. A., Ramos, I. Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004). 2004. 1935

Boronat, A., Carsí, J. A., Ramos, I., Pedrós, J. An Approach for Cross-Model Semantic Transformation on the . NET Framework. Proc. of the .NET Technologies'2005 Conference. 2005. 1947

Boronat, A., Boronat, A., Carsí, J. A.,Ramos, I. Una Arquitectura para la Definición

de Metáforas Gráficas para Metamodelos. Revista IEEE América Latina, Special Edition - JISBD'2004. 2005. 1957

Silva, J., Carsí, J. A., Ramos, I. Análisis teórico experimental de criterios de comparación de esquemas conceptuales orientados a objeto. Computación y Sistemas, Vol. IX no.1. 2005. 1967

Boronat, A., Carsí, J. A., Ramos, I. Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. Proc. of the 9th European Conference on Software Maintenance and Reengineering. 2005. 1983

Boronat, A., Carsí, J. A., Ramos, I. Automatic Support for Traceability in a Generic Model Management Framework. Proc. of the European Conference on Model Driven Architecture - Foundations and Applications. 2005. 1995

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). 2005. 2011

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos - JISBD'05. 2005. 2019

Queralt, P., Hoyos, L., Boronat, A., Carsí, J. A., Ramos, I. Un motor de transformación de modelos con soporte para el lenguaje QVT Relations. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). 2006. 2025

Boronat, A., Carsí, J. A., Ramos, I. Exogenous Model Merging by means of Model Management Operators. Proc. of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change at ICGT 2006. 2006. 2035

Boronat, A., Carsí, J. A., Ramos, I.,Letelier, P. Formal Model Merging Applied to Class Diagram Integration. Proc. of the International ERCIM Workshop on Software Evolution (2006). 2006. 2055

Boronat, A., Carsí, J. A., Ramos, I. Algebraic Specification of a Model Transformation Engine. Proc. of the Fundamental Approaches to Software Engineering Workshop (FASE'06) at ETAPS'06. 2006. 2061

Boronat, A., Ramos, I., Carsí, J. A. Definition of OCL 2. 0 Operational Semantics by means of a Parameterized Algebraic Specification. Proc. of the 1st International Workshop on Algebraic Foundations for OCL and Applications. 2006. 2077

Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J. A. An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework. Proc. of the Second European Conference on Model Driven Architecture. 2006. 2093

Gómez, A., Boronat, A., Hoyos, L., Carsí, J. A., Ramos, I. Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos.

Actas de las XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006). 2006. 2109

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I., Täubner, C., Eckstein, S. Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2119

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2129

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2135

Lloret, E., Escobar, S., Palomar, M., Ramos, I. Incremental and adaptive Software Systems Development of Natural Language Applications. Proc. of the 22nd International Conference on Information Systems Development (ISD 2013). 2013. 2143

VOLUMEN 5: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

PARTE VIII: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

Prólogo de Emilio Insfrán y Silvia Abrahão 2177

Carsí, J. A., Insfrán, E., Abrahao, S., Genero, M., Piattini, M., Ramos, I. ¿Cómo utilizar experimentos controlados en la selección de transformaciones QVT para obtener diagramas de clases UML más fáciles de entender? Proc. of the 8th Argentine Symposium on Software Engineering (ASSE 2007). 2007. 2179

Limón, R. Ramos, I. Aragón, R. Transformación de Vistas Arquitectónicas Orientada por Modelos. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 2189

Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Marco de Trabajo basado en MDA para la Medición Genérica del Software. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2199

Mora, B., García, F., Francisco Ruíz, Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Software Measurement by Using QVT Transformations in an MDA Context (ICEIS 2008). Proc. of the Tenth International Conference on Enterprise Information Systems, Volume DISI. 2008. 2209

Abrahao, S., Genero, M., Insfrán, E., Carsí, J. A., Ramos, I., Piattini, M. Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In Rech, J. and Bunse, C. (Eds.), Model-Driven Software Development: Integrating Quality Assurance. 2008. 2217

Insfrán, E., Carsí, J. A., Abrahao, S., Genero, M., Piattini, M., Ramos, I. Towards Quality-Driven Model Transformations: A Replication Study. Proc. of the International Workshop on Empirical Studies of Model-Driven Engineering (ESMDE 2008) at MODELS 2008. 2008. 2247

Cabello, M. E., Ramos, I. A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines. Proc. of the International Conference of Health Informatics- HEALTHINF´08. 2008. 2257

Cabello, M. E., Ramos, I. Variability Management in Product Lines for Decision Support Systems. Proc. of the 10th International Conference on Enterprise Information Systems (ICEIS´08). 2008. 2267

Cabello, M. E., Ramos, I. The Baseline: the milestone of Software Product Lines for Expert Systems automatic development. Proc. of the 9th Mexican International Conference on Computer Science. 2008. 2277

Gómez, M., Gómez, A., Cabello, M. E.,Ramos, I. BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. Actas del VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). 2009. 2287

Cabello, M. E., Ramos, I., Gómez, A., Limón, R. Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for Expert Systems development. Proc. of the 1st Asia Conference on Intelligent Information and Database Systems (ACIIDS´09). 2009. 2297

Cabello, M. E., Ramos, I. Expert Systems development through Software Product Lines techniques. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2303

Cabello, M. E., Ramos, I. Model-Driven Development of Decision Support Systems: tackling the variability problem. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2315

Gómez, A., Ramos, I. Cardinality-based feature models and Model-Driven Engineering: fitting them together. Proc. of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. 2010. 2327

Gómez, A., Cabello, M. E., Ramos, I. BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. Proc. of the 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010) at SPLC 2010. 2010. 2335

Gómez, A., Ramos, I. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. Proc. of the 19th International Conference on Information Systems Development (ISD'10) 2010. 2343

Abrahao, S., Javier González, Insfrán, E.,Ramos, I. Software Evolution in Model-Driven Product Line Engineering. ERCIM News 88, Special Theme: Evolving Software. 2012. 2355

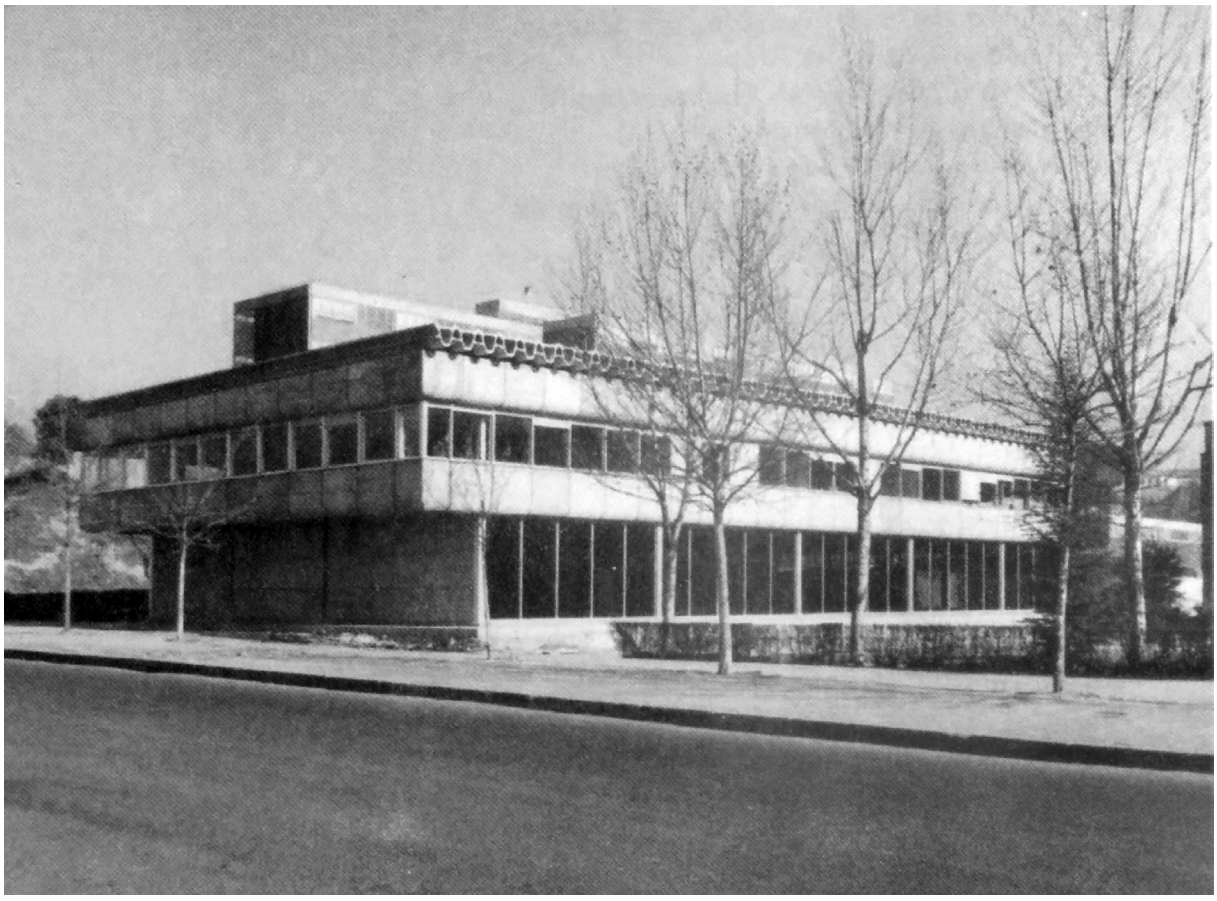
Cabello, M. E., Ramos, I., Gutiérrez, J. R., Gómez, A., Limón, R. SPL Variability Management, Cardinality and Types: an MDA Approach. International Journal of Intelligent Information and Database Systems (IJIIDS), Vol. 6, No. 2. 2012. 2357

Insfrán, E., Abrahao, S., Javier González, McGregor, J. D., Ramos, I. A Multimodeling Approach for Quality-Driven Architecture Derivation. Proc. of the 21st International Conference on Information Systems Development (ISD2012). 2012. 2383

PARTE IX: LOS AMIGOS

Los amigos 2397

Apéndice 2441



PARTE I

MADRID (1968-1974)

PRÓLOGO DE ROBERTO MOYA QUILES

ISIDRO RAMOS Y EL CENTRO DE CÁLCULO DE LA UNIVERSIDAD DE MADRID, POR ERNESTO GARCÍA CAMARERO

MADRID (1968-1974)

Parte I

El progreso en la ciencia depende de
nuevas técnicas, nuevos descubrimientos y
nuevas ideas, probablemente en ese orden.
(Sidney Brenner,premio Nobel
de Fisiología o Medicina en 2002).

Pero ese progreso no sería posible sin la concurrencia indispensable de las personas que tienen ideas y a través de su perseverancia logran aportar nuevas técnicas y, por ende, nuevos descubrimientos.

La obra de Isidro ha ido en esta línea, y la relevancia de su obra queda patente, entre otros testimonios, a través del presente volumen y, en variadas publicaciones. A través de éstas existen huellas de textos histórico-técnicos de la época en la que coincidimos y en alguna medida, mi aportación a esta edición pretende ser más desde una percepción de espectador y testigo de los logros de la persona que posibilitaron la obra.

Creo que la etapa que compartí con Isidro se la puede «etiquetar» como años ilusionantes de formalización del trabajo de «los pioneros». Es importante tener en cuenta que, en esa época, la Industria proporcionaba el conocimiento para un uso básico de los productos, dejando a la intuición y al libre albedrío su aplicación para producir algún resultado, frecuentemente por intercambio de «recetas» («El Practicón»).

Isidro fue portador a España de las corrientes de pensamiento y del «Estado del Arte» que circulaban a nivel internacional.

En aquellos «primeros tiempos», se trabajaba e investigaba con mucha ilusión y, entre los incentivos, estaba la esperanza de que, en esa ocasión, España cambiase el curso de su trayectoria hacia «otra» historia, en la cual no perdiéramos «el tren».

Las grandes ideas se conciben simultáneamente por varias personas que han respondido de una forma similar al clima del pensamiento. (...) La única cosa que distingue lo bueno de lo mejor es cómo se ejecuta – cómo se hace realidad. (J. Craig Venter «A Life Decoded. My Genome: My Life», traducción Espasa «La vida descodificada»).

Entre los dones naturales de Isidro para el trabajo de investigación y técnico, sobresalía su meritoria tenacidad para afrontar los desafíos inherentes a los objetivos que se proponía.

Finalmente, solamente agradecer en la persona de Isidro, su constancia para mantener vivo, a pesar de las dificultades que cualquier carrera profesional implica, ese objetivo inicial y lograr desarrollar nuevas ideas para los nuevos descubrimientos.

Es mi deseo no terminar esta introducción sin agradecer al Ph. D. José Hilario Canós Cerdá la posibilidad que me ha brindado de estar presente en esta ingente recopilación de publicaciones del Prof. Isidro Ramos Salavert, como testigo de los albores de su obra.

Roberto Moya Quiles
(1er. Doctor de Isidro Ramos, Universidad del País Vasco, 26 marzo 1977)

ISIDRO RAMOS Y EL CENTRO DE CÁLCULO DE LA UNIVERSIDAD DE MADRID

por Ernesto García Camarero
ex-director del CCUM

Escribir sobre Isidro Ramos es escribir sobre una persona notable, pero sobre todo sobre un viejo y querido amigo cuando ha entrado en el apacible periodo de su jubilación. Le conocí cuando entró como analista de sistemas en el Centro de Cálculo de la Universidad Complutense (CCUM). Él era un joven físico que venía de haber pasado una temporada en el CERN de Ginebra. Estaba lleno de energía y de entusiasmo. Venía a un Centro universitario, en aquella época único e innovador, dedicado a la informática.

En este Centro, Isidro Ramos cambio su vocación de físico por la de informático, y yo fui uno de los culpables de que esta metamorfosis se realizara, especialmente cuando le hice observar que la física llevaba siglos de desarrollo y la informática estaba naciendo en un mundo prometedor. Él inició con éxito ese cambio. Sus primeros años los pasó en el Centro de Cálculo, y desde allí inició su fecunda carrera.

En estas líneas recordaremos algunas de las actividades de Isidro en el CCUM, pero antes permítaseme que me extienda un poco en recordar algunas de las cosas que se hacían en ese Centro y cuál era su espíritu y ámbito de trabajo en los años en los que Isidro estuvo inmerso.

Creado por un acuerdo entre IBM y la universidad de Madrid, funcionó el CCUM desde finales de los años 60 hasta el comienzo de los años 80. La Universidad aportaba el edificio[1] y el personal para su funcionamiento e IBM aportaba un ordenador 7090 y los periféricos correspondientes. La condición puesta por IBM fue que la actividad del Centro se orientase exclusivamente a la docencia y a la investigación, así como la difusión de estas técnicas en las universidades españolas.

En los años 60 del siglo XX, se estaba produciendo un cambio de paradigma en relación a lo que eran las computadoras. Fue un gran salto mundial con el que se pasó de considerar a las computadoras como máquinas puramente aritméticas al servicio del cálculo numérico, a ser consideradas como maquinas lógicas aplicables al tratamiento de cualquier tipo de información y, por tanto, abriendo universalmente su dominio de aplicaciones. Este cambio significó, por una parte, una especial atención a la lógica simbólica, a los lenguajes formales, a la inteligencia artificial, a la lingüística, a la representación del conocimiento…Y, por otra, a la búsqueda de aplicaciones avanzadas en las que estuviesen presentes los métodos heurísticos, la creatividad, la simulación de modelos, la representación del conocimiento, la conversacionalidad, las redes de comunicación, … Todos estos temas se convirtieron en un gran campo de investigación al que, dada su novedad, se podrían incorporar grupos de científicos de todo el mundo con altas probabilidades de aportar innovaciones y la oportunidad de las consecuencias derivadas de ellas.

Dentro del ambiente sociopolítico español, en los años 60, se estaba iniciando un cambio que afectaba principalmente a un cierto desarrollo económico, y con él la necesidad de un desarrollo tecnológico y científico. Todo esto hacía necesario cambios que facilitaran la creación de un ambiente cultural y una apertura internacional para dejar entrar un flujo de ideas que había estado cerrado durante mucho tiempo.

Visto desde la perspectiva actual, la fecunda actividad del CCUM logró abrirse al cambio de paradigma y a que se elevara el nivel de atención a la informática universitaria en España, tanto en educación (universitaria y secundaria) como en investigación, y que se abordaran temas de informática teórica y creatividad, necesarios para participar en la creación de la nueva ciencia y tecnología.

Desde el CCUM se impulsaron:

El servicio de cálculo en las universidades. Ya antes había algunos ordenadores asignados a algunas cátedras, pero fue el CCUM el primer centro que dio servicio regular de cálculo a las universidades españolas, tanto a los profesores en sus trabajos de investigación como a los estudiantes para su formación.

La enseñanza de la informática en los diversos niveles educativos, desde el entrenamiento técnico (principalmente en lenguajes de programación), al nivel universitario y en la enseñanza secundaria.

En la enseñanza universitaria se elaboraron planes de estudio universitarios (como el iniciado en la especialidad de Computación en la Facultad de Matemáticas de la Universidad de Madrid y el apoyo dado en la creación del Instituto de Informática, que unos años después se convertiría en la Facultad de Informática de la UPM). No solo se elaboró el plan de estudios que fue aprobado por la Junta de Gobierno de la universidad, sino que sus clases se impartieron en las propias aulas del CCUM, por su propio personal titulado. El espíritu de las enseñanzas en la Facultad de Matemáticas era doble: por una parte, considerar a la informática teórica como parte de la nueva matemática y, por otra, considerar a los problemas que surgían de la nueva ciencia informática como inspiradores del desarrollo de la matemática, de forma análoga al papel que

1 El edificio de nueva planta diseñado especialmente para Centro de Cálculo por el arquitecto Miguel Fisac, sirvió como ejemplo de este tipo de recintos.

tradicionalmente había jugado la física. Además de la informática teórica se incluyeron también temas más directamente vinculados con el software o con las aplicaciones, especialmente las no numéricas.

En la enseñanza secundaria se desarrolló la primera experiencia nacional, con la colaboración de las universidades de Barcelona, Zaragoza y Alcalá de Henares. Con esta experiencia se observó, en tres distritos universitarios distintos y en varios institutos y colegios de secundaria, el grado de receptividad del uso de ordenadores en alumnos de 11-13 años. Se publicaron los resultados de la experiencia y se redactó un manual de informática para niños y se promovió, en diversos ámbitos nacionales, este nivel de la enseñanza de la informática con la colaboración de FUNDESCO, CITEMA, INCIE, e internacionales siendo miembro de varias asociaciones (en particular WG.3.1 de la IFIP) o participando en varias reuniones y congresos (como en los Congresos de la IFIP, en los seminarios de la OCDE en Paris o en el seminario de la OCDE en Frascati)

Los estudios de informática teórica (especialmente la lógica y lingüística computacionales y sus aplicaciones), que se desarrollaron mediante diferentes cursos, cursillos y conferencias en los que colaboraron, un núcleo de lógicos españoles formados en universidades europeas (Fernandez Prida, García Camarero, Martínez,…), junto a especialistas extranjeros. Sus trabajos han sido recogidos en diversas publicaciones monográficas y su actividad ha ejercido influencia en los estudios de estas disciplinas en diversas universidades españolas.

La aparición y desarrollo de las computadoras ha venido necesariamente precedida por los estudios de lógica formal, y su crecimiento hacia el futuro necesitará también el crecimiento de esta ciencia. El ábaco chino se considera como el germen de máquinas aritméticas como las de Pascal o Leibniz del siglo XVII que, más tarde, se desarrollaron como máquinas de sobremesa en el siglo XIX. Pero las computadoras solo pudieron aparecer cuando se difundieron los estudios teóricos de Boole recogidos en su lógica o álgebra. También fueron indispensables los estudios de la lógica simbólica o formal realizados por Frege, Russell, Hilbert, Gödel, von Neumann, Turing, Church, … gracias a los cuales se concibió la estructura y arquitectura de las computadoras, y se abrió el camino para el desarrollo de los lenguajes artificiales y de programación, de la teoría de algoritmos y de la inteligencia artificial sin lo cual no se habría desarrollado la informática con tanta seguridad y rapidez.

En el CCUM se pensó que si se quería que España participara de forma positiva en el desarrollo de la informática mundial, sus técnicos no debían comportarse como simples dispositivas periféricos, y para ello era imprescindible que se estudiara, cultivara y difundiera la informática teórica.

Por eso, desde el comienzo de sus actividades, incluyó entre ellas la realización de conferencias y de cursos breves (20 horas), dadas por profesores de diversas universidades europeas y americanas que se dedicaban a esta especialidad. Entre los profesor extranjeros que colaboraron en esta tarea podemos citar a: Bernard Vauquois, Director del Centro de Estudios de Traducción Automática de la Universidad de Grenoble, Rene Moreau, Director del Desarrollo Científico de IBM Francia, Marco Schützenberger , del MIT y de Universidad de Paris VII, Salomon Klasczko de la Universidad de Frankfurt, Claud Pair, del Instituto Universitario de Tecnología, de la Universidad de Nancy, Maurice Nivat, de la Universidad de París VII, Jacques. Pitrat, de la Universidad París VI y Maître de récherche en el C.N.R.S, Michel A Melkanoff, Director del Computer Science Department de la Universidad de California en los Ángeles (UCLA), M. Galinier, de la Universidad de Toulouse, J. L Lauriere. de la Universidad de PARIS VI, Dominique Perrin, de la Facultad de Matemáticas de la Universidad de Ruan, J. L Lauriere, del Institut de Programmation, Universidad PARIS VI, A. Colmerauer, de la Universidad d'Aix-Marseille,. D. Pastre, de la Universidad París VII, B. Robinet, de la Universidad París VI, R. Demolombe, (del ONERA

— CERT, Toulouse), Christian Choffrut, de la Université París VII, Ph. Deschamps. del Laboratoire de Langages et traducteurs del INRIA (París), Robert CORI, de la Universidad de Burdeos, M. Penttonen, de la Facultad de Matemáticas, Universidad de TURKU (Finlandia). También colaboraron, entre otros, en este tipo de cursos los profesores españoles: Fernando. Orejas, de la Facultad de Matemáticas, Universidad de Madrid, José Díaz, de la Facultad de Informática. Universidad Politécnica de Barcelona, Francisco J. Garijo, del Centro de Informática. Universidad del País Vasco, Josu Aramberri, de la Facultad de Ciencias. Universidad del País Vasco,

De forma más regular se inició la introducción de estas ideas en la especialidad de la Facultad de Matemáticas de la Universidad de Madrid, especialmente con las asignaturas de Teoría de autómatas y lenguajes formales impartida por Ernesto García Camarero y de Teoría de la Computabilidad, por José Fernández Prida[2].

La importancia dada por el CCUM a los temas de informática teórica y el reconocimiento internacional a esta actividad quedó recogida en varias reuniones internacionales que organizó el CCUM, solo o en colaboración con otras instituciones. Como el Coloquio Sobre Lógica Simbólica, celebrado en el CCUM, entre los días : 17-21de Febrero 1975 (en este Simposio colaboraron con ponencias junto a los alemanes H.D. Ebbinghaus, H. Hermes, E. Borger, J. Flum, los españoles J.F. Prida, M. Martínez: J. Meseguer, I. Sols y se editaron las actas)

O el Seminario Internacional Teoría de Lenguajes y de la Programación, organizado por el CCUM en colaboración con el Centro de Investigación UAM - IBM Universidad Autónoma de Madrid durante los días 23 a 25 abril 1975. En este seminario actuaron como ponentes: Dr. A. D. Falkoff, del IBM Philadelphia Scientific Center; Prof. S. Ginsburg, Computer Science Program University of Southern California; Prof. Klaus Indermark, Institut für Angewandte Mathematik und Inforinatik Universität Bonn; Prof. Ingbert Kupka, Institut für Informatik Universität Hamburg; Prof. M.A. Melkanoff, Computer Science Department University of California. Los Angeles ; Prof. Maurice Nivat, U.E.R. de Mathématiques Université París VII; Prof. Claude Pair, Instituto Universitaire de Calcul Automatique Université de Nancy; Prof. Arto Salomaa, Inslitute of Mathematics University of Aarhus; Prof. Marcel Schützenberger. U.E.R. de Mathérnatiqucs Université Paris VII; Prof. A. van Wijngaarden, Stichting Mathematisch Centrum Amsterdam

También se colaboró en varias ediciones del International Colloquium on FORMALIZATION OF PROGRAMMING CONCEPTS, como el organizado Peñiscola, en abril de 1981 por el Centro Cálculo Universidad Literaria Valencia, Centro Cálculo Universidad Complutense Madrid, Facultat d'Informática Universitat Politécnica Barcelona.

Búsquedas de nuevas aplicaciones no numéricas de la informática, especialmente en las que se incluye de algún modo la creatividad. Esta actividad se desarrolló en torno de unos seminarios de investigación entre los que figuran los dedicados a: autómatas adaptativos, ordenación de la construcción, valoración automática del aprendizaje, generación automática de formas plásticas, composición automática de espacios arquitectónicos, lingüística matemática, ordenadores en la música, enseñanza de ordenadores en secundaria, a los modelos para la simulación de sistemas educativos, a la enseñanza programada asistida por ordenador, a los planes de estudio universitarios en informática, y algunos más. Los resultados detallados de estos seminarios se recogieron en el Boletín del Centro de Cálculo (del que se publicaron tres o cuatro números al año durante todo el tiempo de existencia del Centro. También los resultados se incluyeron en diversas publicaciones y exposiciones monográficas, y en Coloquios o Simposios nacionales e internacionales obtuvieron una buena acogida.

2 Pero además el CCUM fue un foco para el desarrollo de esas disciplinas especialmente gracias a la figura de José Fernández Prida, quien se había doctorado en estos temas en la Universidad de Friburgo, con el eminente lógico alemán Hans Hermes. Los trabajos de Prida fueron esenciales en los aspectos más abstractos de la teoría de computabilidad, y de las funciones recursivas, y formó a un grupo de colaboradores…. También colaboró en esta tarea Mariano Martínez, que había trabajado con el lógico polaco Mostowski en la universidad de Varsovia.

Nos hemos entretenido en dar esta visión global, aunque incompleta, del Centro de Cálculo de la Universidad Complutense para describir el entorno en el que Isidro Ramos se movía en los primeros años de su actividad informática. Él colaboró con el Centro, como analista de sistemas, de una forma muy activa y eficaz. Sus primeras publicaciones también se hicieron en el Boletín o en monografías del CCUM.

Uno de los primeros trabajos de Isidro Ramos, fue fruto de su participación en el Seminario de valoración automática del aprendizaje, al que asistía con regularidad, del que fue motor el catedrático de terapéutica física Miguel Gil Gayarre y en el que colaboró muy activamente el lingüista Víctor Sánchez de Zavala. Este Seminario tenía por objeto el estudio de los procedimientos de realizar exámenes que fueran valorados automáticamente por ordenador. Era un método que se hacía necesario cuando crecía de forma espectacular el número de estudiantes en las universidades y en otras instituciones. En este contexto Isidro desarrollo un programa para la valoración automática del aprendizaje, que funcionó efectivamente y que publicó en el nº 1 del Boletín (1968). También como resultado de los trabajos de este Seminario, Isidro publicó en colaboración con Miguel Gil Gayarre, Mª Jesús Manzanas y Carlos Gil Gayarre, en la Revista de Automática (1970) un largo artículo sobre Posibilidades derivadas de la automatización en la corrección de exámenes.

También Isidro Ramos escribió, en colaboración con García Camarero, una comunicación presentada en un Coloquio sobre aplicaciones de los ordenadores en Medicina, que se organizó en el CCUM (1971), en colaboración con IBM. Se trataba de un Ensayo sobre la definición automática de tratamientos clínicos, en el que aplicaban conceptos de la teoría de formal de lenguajes y de autómatas adaptativos.

También colaboró de forma continuada en el Seminario sobre enseñanza de ordenadores en la escuela secundaria. Como ya hemos dicho, a partir de este seminario se inició un proyecto en colaboración de las universidades de Barcelona, Zaragoza y Alcalá de Henares. Su finalidad era observar y valorar el comportamiento de los escolares, de 11 a 13 años, en relación con el uso de los ordenadores y el aprendizaje de los conceptos fundamentales de la informática. Entre diversas herramientas de estudio se definió un ordenador virtual al que se llamó 70/13, que se programaba en el leguaje de máquina. Hubo que redactar un lenguaje traductor (Traductor 70/13) de las secuencias de instrucciones en el lenguaje de la máquina 70/13 de que estaban compuestos los programas redactados por los niños para que pudieran ser ejecutados en la computadora IBM 7090 del CCUM. La elaboración de este traductor estuvo a cargo de Isidro Ramos (con la colaboración de Irene Fernández Flórez, también analista del CCUM). De este trabajo resultaron varias publicaciones. Las dos primeras en el Bol. CCUM (1970) y en el libro Ordenadores en la Escuela Secundaria, informa sobre una experiencia (CCUM, 1971), en ambos casos en colaboración con Irene Fernández Florez. La tercera (Irene Fernández Florez, Ernesto García Camarero, Isidro Ramos) A Basic language oriented to secondary school, fue una ponencia presentada en el Congreso Internacional de la IFIP en Lubljana, en 1971 (al parecer fue la primera ponencia española aceptada en un congreso de la IFIP).

La participación de Isidro Ramos en el Seminario sobre generación automática de formas computables fue muy activa desde el mismo momento de su creación. No solamente porque a través suyo se incorporaron Aguilera Cerní y José María Yturralde, que ayudaron a definir el espíritu del seminario, sino también por su colaboración en tareas técnicas necesarias para materializar, mediante programas de ordenador, las propuestas artísticas que se hacían en el mismo. En particular recuerdo que a Isidro se debieron los programas para construir los cuadros compuestos por los módulos del pintor Barbadillo y la utilización del plotter para realizar los bocetos de algunas de las figuras imposibles del pintor Yturralde. Hemos de recordar que las



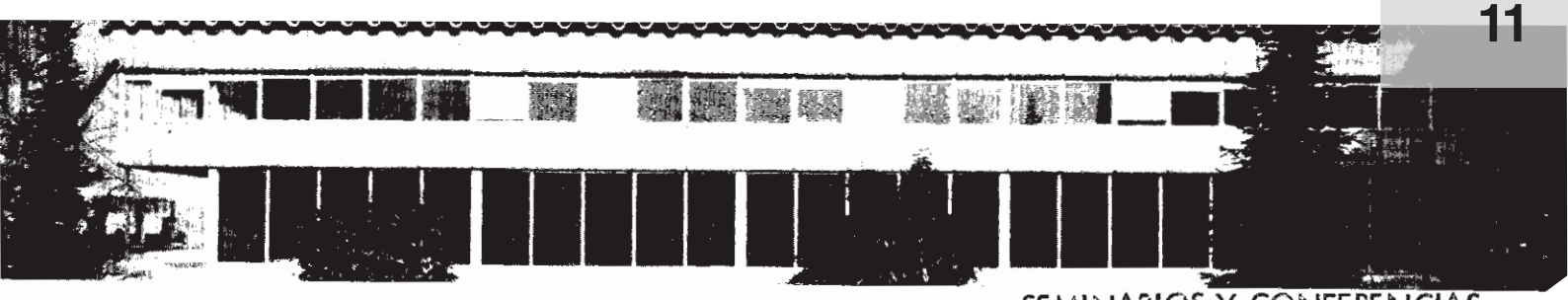
Primera reunión del Seminario de Generación Automática de Formas Plásticas del CCUM el 18 de diciembre de 1968, a las 13.00 h (Pintores, arquitectos e informáticos). Desde la izquierda: E. Delgado, Irene Fernández Florez, Abel Martín, Eusebio Sempere, Manuel Barbadillo, Josér María Yturralde, Aguilera Cerni, Guillermo Searle, Ernesto García Camarero, Fernando Álvarez Cienfiegos, Mario Fernández Barberá, A. García Quijada, Isidro Ramos, Soledad Sevilla Portillo, José Miguel de la Prada Poole, Manuel Casas Gómez y Javier Seguí de la Riva.

obras de estos pintores fueron las que más atención merecieron de entre las obras del Computer Art que se desarrollaron en el Centro. También estas obras tuvieron una gran difusión a través de las exposiciones organizadas en el CCUM y numerosas otras retrospectivas, así como reproducidos en revistas y libros dedicados al arte cibernético, en los que no es habitual citar a los informáticos que ayudan a realizar las obras.

Después de hacer el Traductor 70/13, orientaría Isidro su atención a temas vinculados con las Bases de Datos, la Ingeniería de Software, o los Lenguajes de programación a los que principalmente se dedicaría en adelante. Sobre estos temas publicó durante su periodo en el CCUM: Aplicación del hash-coding al tratamiento automático de gramáticas (en colaboración con Moya, (Bol. CCUM, 1970), Lenguajes para la escritura de traductores (Revista de Automatica, 1971), Un analizador sintáctico para un subconjunto de PL/I. (en colaboración con M. Alfonseca, presentado en el Congreso Nacional Automática, Barcelona, 1972, en que se incluye el listado del programa), Un ejemplo de recursividad: Hanoi (Bol. CCUM, 1974) en que agradece a Claude Pair haberle familiarizado con este tema, Sistema de creación y mantenimiento de bancos de información. (en colaboración con Lozano y Muñóz Herrero) que se publicó en tres partes en la Revista de Automática.

Después de su paso por el CCUM, Isidro Ramos continuó su brillante carrera. Claude Pair, matemático graduado en la École normale supérieure de Paris, (director y fundador del Centro de Investigación en Informática de Nancy), fue invitado por el CCUM, en mayo de 1971, a dar un curso sobre Estructura de la Información, al que asistió Isidro Ramos. De ahí surgió la invitación que le hizo Pair a Isidro para integrarse en su grupo y preparar con él su doctorado.

A su doctorado siguió una larga carrera académica, con su paso por el rectorado de la universidad de Castilla La Mancha, y su fecunda actividad de docencia e investigación en la Universidad Politécnica de Valencia. Los resultados de sus investigaciones y de los muchos proyectos que dirigió fueron presentados en numerosos Simposios, Conferencias, Reuniones y Congresos nacionales e internacionales y recogidos en más un centenar de publicaciones realizadas cerca de setenta coautores y que han sido citadas por más de trescientos cincuenta autores.



CENTRO DE CALCULO DE LA UNIVERSIDAD DE MADRID

SEMINARIOS Y CONFERENCIAS
Nº 1 Diciembre 1968

Sumario

Seminarios

Presentación 1
Valoración del aprendizaje 2
Composición de espacios arquitectónicos con ordenadores 9
Lingüística matemática 12
Generación de formas plásticas 16
Ordenación de la Construcción 16

Conferencias

Cálculo numérico con ordenadores electrónicos en el tratamiento de vibraciones moleculares 17

SEMINARIOS

Presentación

Entre las tareas a desarrollar por un Centro Universitario de Cálculo, nos ha parecido de especial importancia impulsar el estudio de la posibilidad de automatizar procesos de investigación y análisis, en campos en los que hasta ahora esta automatización no ha penetrado.

Con ello no pretendemos reducir toda actividad intelectual, científica o artística, a puro mecanismo, pero sí desglosar esa actividad en un aspecto puramente creador, de otro más bien mecánico, y de esta forma aumentar la capacidad creadora liberándola de la servidumbre condicionada por lo reiterativo y mecánico.

Con estos fines generales y recogiendo opiniones y deseos de varias personas, hemos propiciado la creación y funcionamiento de diversos seminarios de estudio, cuya actividad ya se ha iniciado en el primer trimestre del presente curso académico.

Como se ha manifestado por los participantes a todos los seminarios la necesidad de vincularse con las personas nacionales o extranjeras que puedan aportar algo para mejorar la marcha de los estudios, hemos creído oportuna, independientemente de la correspondencia directa, la publicación de un Boletín que sirva para crear esa vinculación, al tiempo que proporcione un vehículo de información para que en las Universidades y otros núcleos de estudio tengan noticia de nuestras actividades, y los que así lo deseen puedan relacionarse con nosotros. La aparición de este Boletín estimamos que será mensual, y en cada número incluiremos resúmenes de los asuntos tratados y de la bibliografía usada o recomendada. Tambien incluiremos aquellas noticias, o sugerencias, que se nos hagan por correspondencia. En principio su distribución será gratuita, y restringida a las autoridades académicas y a aquellas personas que estando especialmente interesadas por estos temas nos lo soliciten.
E.G.C.

Valoración del aprendizaje

Participantes: E. García Camarero, M. Gil Gayarre, R. Moya Quiles, M. A. Oliva Portolés, I. Ramos, V. Sanchez de Zavala.

Reuniones: 14 y 28 de Novoembre y 12 de Diciembre
Hora: 9 de la mañana.

En la primera reunión se intercambiaron ideas sobre la problemática de la calificación automática de exámenes, en la que principalmente Gil Gayarre expuso su experiencia y planes prácticos para la calificación de las pruebas de su Cátedra de Terapéutica Física de la Universidad de Madrid.

En la segunda Victor Sanchez de Zavala propuso un plan general (provisional) de trabajo, que incluimos más abajo.

En la tercera Isidro Ramos expuso las lineas generales del programa sobre valoración del aprendizaje que está realizando para la computadora IBM 7090 del Centro de Cálculo de la Universidad de Madrid, sobre el cual se incluye una reseña en este Boletín.

Próximas reuniones: 9 y 23 de Enero y 6 y 20 de Febrero

Comunicaciones:

PLAN PROVISIONAL DE TRABAJO

Teniendo en cuenta lo que nos expuso el día pasado Gil Gayarre, creo que podría proponerse el siguiente plan provisional de trabajo:

I - Aportación de bibliografía y documentación. Aquí habrá que incluir los impresos hasta ahora utilizados en la cátedra de Terapéutica Física, etc.
II - Establecimiento de contacto con otras personas (posiblemente de fuera de Madrid) y organizaciones (posiblemente de fuera de España).
III - Especificación de los temas más importantes que podrían estudiarse.
IV - Asignación provisional del orden de ataque de los temas y su posible distribución entre los asistentes al seminario. (indudablemente, será más fructuoso que varias personas se ocupen de cada uno de los que comiencen por estudiarse).
V - Resolución (provisional) de algunas de las principales cuestiones de los temas elegidos.
VI - Experimentación y corrección de las soluciones.

Sobre el apartado I

Solo puedo indicar ahora que en 1965 se presentó una tesis en la Facultad de Filosofía y Letras de Valencia sobre "Pruebas objetivas en Geografía e Historia", escrita por José VICENTE ROMEU, que ha sido ayudante de Pinillos allí y que actualmente está fuera de España (de profesor). Pinillos dice que estudia distintos modos de examinar (con preguntas para contestar si o no, con elección entre varias respuestas, rellenando huecos en posibles respuestas, etc.) y señala cuales resultan más convenientes y cuales suscitan más desconfianza en los alumnos _ que en Valencia llegó al asalto del seminario en que se guardaban los exámenes y a su sustracción.

Tambien convendría, creo, adquirir las siguientes obras:

P.J. STONE, D.C. DUNPHY, M.S. SMITH y D.M. OGILVIE: "The General Inquirer: A Computer Approach to Content Analysis"

y

Cambridge Computer Associates of Cambridge, Massachusstts: "User's Manual for the General Inquirer"

ambas publicadas por M.I.T. Press, Cambridge de Mass., en 1967.

Sobre el apartado II

Creo que puede intentarse establecer contacto con el Instituto de estadística médico-biológica y documentación de la Universidad de Marburgo, en particular con su director, el Dr. Peter Ihm, que está organizando sistemas de efectuar pruebas objetivas para valoración con computadora (segun el Frankfurter Rundschau de 10 de Octu-

bre último).

Desgraciadamente, no he sido capaz de encontrar el número de Arbor en el que escribía un catedrático de Medicina de Barcelona, al parecer en contacto con otros franceses, sobre pruebas análogas que éstáutilizando.

Sobre el apartado III

A mi juicio, pueden resumirse los temas que más directamente nos interesan bajo los siguientes epígrafes:

1º - Qué queremos medir.
2º - Los instrumentos de medida.
3º - El proceso de medición.
4º - La valoración de las medidas,
5º - La preparación de los instrumentos de medida.

Vamos a ver qué podría abarcar cada uno de estos temas (que simplemente resumen las propuestas de Gil Gayarre).

1º - Qué queremos medir

Podemos querer medir

A) algo referente a los discentes
B) algo referente a los docentes

En cuanto a A), puede interesarnos

a) su situación (intelectual) actual
b) la base actual que nos permita predecir cierta situación futura (su eficacia profesional, o incluso su éxito académico).

La cuestión a) puede subdividirse del siguiente modo:

a1) los "conocimientos" adquiridos, que pueden considerarse, (en punto al contenido):
- conceptos básicos
- temas (del programa de la asignatura, o partes de ellos)
- bibliografía

y en cuanto al "modo" de presencia, se puede intentar averiguar
- su existencia
- la rapidez de acceso a ellos

a2) la capacidad solventatoria (que parece esencial, tanto dada la teoría más corriente hoy de la inteligencia como capacidad de resolver problemas, como teniendo en cuenta que en la mayoría de los ejercicios profesionales de lo que se tratará será de resolver un problema);

Entre sus aspectos más importantes podemos atender a
- la amplitud (= nivel de generalidad) y conectividad (con otros temas o disciplinas) del tratamiento del problema
- la precisión de los resultados
- la claridad de exposición
- la rapidez de resolución

- la soltura bibliográfica (en cuanto a referencias y a manejo durante la prueba)

La cuestión b) puede subdividirse así:

b1) la operatividad intelectual, o características formales que presente la resolución de problemas por el discente (que se encontrarán a medio camino entre los rasgos que midan la capacidad solventatoria de los problemas concretos presentados y las características que suelen llamarse psicológicas, esto es, que pretenden ser enteramente independientes de la disciplina, e incluso de la instrucción y cultura del sujeto).

A este respecto habrá que considerar, creo:

. el enfoque de los problemas:
  - generalizador
  - puntualizador
  - formalizador
  - metodológico, etc
. la categorización y sistematización del tema
. la estrategia empleada
. los recursos formales utilizados:
  - deductivos
  - analógicos, etc
. la receptividad mostrada ante:
  - los datos
  - los indicios
. los análisis críticos y refutaciones
. la inventiva mostrada (aunque acaso deba pasar al apartado b2)

b2) las "facultades" del sujeto, ya sean
. intelectuales
. del carácter y personalidad, entre las que habrá que distinguir entre las
  - generalmente importantes
  - especificamente importantes para la profesión que sea

Creemos que el valor que podrá darse, en cualquier caso, a pruebas (tests) a este respecto solo habrá de ser de eliminador, cuando no se alcance el umbral que se determine (por estudios sociológicos o de otra índole).

Por lo que se refiere a B), no puedo proponer nada específico actualmente.

Conviene advertir, de todos modos, que para tratar decorosamente este tema será preciso, según me parece, contar con las teorías del aprendizaje, de la inteligencia y de la memoria que hoy parezcan más satisfactorias, así como no descuidar los datos que puedan obtenerse de sociología de la enseñanza.

2º - Los instrumentos de medida

Habrá que estudiar

A) los requisitos computativos (o sea, qué límites nos imponen las máquinas computadoras actualmente existentes y la que tenemos a nuestra disposición, y el "coste" en horas de programación, de funcionamiento, etc. , que representaría llegar hasta cada uno de los distintos límites alcanzables).

B) las características del instrumento; para ello habría que determinar los posibles

. principios de su construcción, por ejemplo:

- de división en módulos
- de agrupación de los módulos en tipos
- de reiteración de módulos (vease el tema 3º, A)

. métodos generales de preparación de módulos, por ejemplo:

- el de "opacidad" de los módulos (que el discente no vea claramente qué es lo que se pretende averiguar a su propio respecto)
- el de las sustituciones dobles (dejar huecos en diversos esquemas de respuesta a elegir)
- el de los esquemas arborescentes (como esquemas explicativos)
- el de los sinónimos (sustituir las palabras empleadas por el discente por otras de un diccionario registrado en la computadora)

. modos de disponer y presentar los distintos tipos de módulos elegidos.

C) las limitaciones del instrumento que se elija

D) la adaptación del instrumento a los recursos computativos.

Es de temer que para presentar un proyecto de soluciones aceptables referentes a este tema será necesario, además de saber todo cuanto se haga ya al respecto en otros centros de enseñanza, estudiar cuidadosamente la teoría de la medida y algo de la teoría de la computabilidad; y, desde luego, realizar muchos ensayos.

3º - El proceso de medición

Aquí habrá que ocuparse, a mi juicio, de los siguientes puntos, por lo menos:

A) la técnica de la medición; cosa que incluirá el estudio de:

. las puntuaciones y su integración en cada prueba, integración que puede ser:

- total (calificar con una puntuación resumen)
- parcial (calificar con un conjunto de puntuaciones)

. las condiciones de ejecución de las pruebas y las instrucciones a dar al respecto.

. el posibles escalonamiento de las pruebas a lo largo del curso (cosa que depende de la forma de desarrollo de la disciplina,

si lineal o cíclicamente, si con clases repetitorias o no, si por grupos de trabajo, etc.)

B) las fuentes de error (incluyendo el "ruido" de los aciertos puramente aleatorios, ausencias, etc.)

C) los métodos de atenuación de los errores

D) la técnica de la preparación para la computación

Como es obvio, los resultados de este tema estarán intimamente trabados con los del tema anterior.

4º - La valoración de las medidas

Habrá que distinguir entre la valoración

A) del discente

B) del docente

Por lo que se refiere a A), es claro que se puede distinguir entre:

. una determinación de orientaciones

- seguidas por los discentes
- recomendadas a ellos en virtud del resultado de las pruebas

. una fijación de los niveles de "calificación" académica.

Respecto de B) no me es posible presentar, de momento, propuesta alguna.

5º - La preparación de los instrumentos de medida

Para penetrar en este último tema, de cuya solución satisfactoria depende que la automatización sea real o meramente aparente (o, a lo más, parcial), tal vez sea útil someter a un análisis de contenido (automatizado, a su vez, naturalmente, y de ahí la importancia de disponer de la obra de Stone et al.) los textos básicos de la disciplina _ esto es, ciertas partes seleccionadas en ellos _ apuntes de clase, etc. Y estos análisis, incidentalmente, tal vez ayudasen a resolver el problema de la valoración B) del tema anterior.

Sobre los apartados IV, V y VI no tengo por el momento ninguna sugerencia que ofrecer.

Indudablemente, el deseo de no olvidar posibles cuestiones importantes me ha llevado a una generalidad en ocasiones temo que algo vacía, pero espero que entre todos podemos llegar a concretarla.

V.S.Z.

Gil Gayarre aportó las siguientes citas bibliográficas:

= HUBBARD, J.P. and CLEMENS, W.C.
Multiple test Examinations in Medicine.
A Guide for Examiner and Examine
1961

HARRIS. T.C. & BUCKLEY - SHARP
Automation of Multiple Choice Examination Marking
1968 British J. Medicine Education 2/1 March

UN PROGRAMA PARA LA VALORACION AUTOMATICA DEL APRENDIZAJE

Como fruto de las conversaciones mantenidas entre el Profesor Gil Gayarre y el Analista del CCUM I. Ramos se pensó que la calificación de exámenes de determinadas materias susceptibles de plantearse en forma de tests, presentaba la posibilidad de una corrección automática utilizando un sistema de proceso de datos. Esto evitaría la tarea rutinaria de corrección y proporcionaría la seguridad de no cometer errores.

El problema fundamental consiste en la selección de un conjunto de preguntas que se plantean al alumno junto con cuatro soluciones por cada una de ellas, solamente una es correcta y el alumno marca la respuesta sobre un papel especial que se le da junto con el cuestionario.

Esto facilita enormemente la contestación (solo se debe poner una cruz en la solución que se crea correcta) y nos permite realizar exámenes de hasta 140 preguntas.

Estos papeles son pasados automaticamente a fichas, donde queda perforado el examen del alumno.

La comparación de estas fichas de examen con un examen patron (que tiene todas las preguntas correctas) nos proporciona el número de preguntas bien contestadas por el alumno y de aquí la calificación (se tiene en cuenta el factor de acertar al azar que resulta ser del 30% y que se resta al % contestado).

El hecho de que el resultado de un examen refleje por una parte el trabajo del alumno y por otra la efectividad de las explicaciones en clase, nos hizo pensar en que unas estadísticas sobre los resultados de los exámenes podrían incidir sobre las explicaciones segun un mecanismo de feed-back.

Por ello, en cada examen, además de la calificación, se hacen los siguientes estudios:

1º) - Dificultad de la pregunta:

La calculamos en función del % de alumnos que no la contestan.

2º) - Fatiga del alumno a través del examen:

El cansancio puede disminuir la rapidez de respuesta haciendo que las últimas preguntas le resulten más dificiles de contestar. Se hace un estudio del % medio contestado en función del número de la pregunta.

3º) - Las condiciones sociales del alumno pueden influir en el rendimiento en los exámenes; por ello se hace un estudio de:

a) - Edad: se ve para un determinado intervalo de edad el % medio que contestan.

b) - Estado: Se distingue entre: casados, solteros, religiosos, ..., y se estudia el porcentaje contestado por cada grupo.

c) - Sexo: se calcula el % medio contestado por los hombres y las mujeres.

4º) - Tiempo medio por pregunta:

Se calcula el tiempo medio necesario para contestar cada pregunta; esto nos informará sobre la duración de exámenes futuros.

Toda esta información puede influir "a posteriori" en la enseñanza.

Como ampliaciones, se tiene pensado la posibilidad de más de una solución correcta por pregunta y que la calificación venga dada posteriormente despues de haber estudiado la dificultad de la pregunta, lo que nos proporciona un peso a tener en cuenta al establecer la calificación.

I.R.

Hemos recibido del Prof. Smith-Agreda, catedrático de Anatomía de la Facultad de Medicina de la Universidad de Valencia, una amable carta en la que nos comunica en líneas generales, el método utilizado por su cátedra para realizar las calificaciones de los exámenes mediante ordenador. Transcribimos a continuación algunos de sus párrafos:

"Adjunto les remito una ficha de las utilizadas por la Cátedra. Al mismo tiempo se entrega a cada alumno un cuestionario con las preguntas en las cuales se dan cinco posibles respuestas, o ninguna puede ser válida. Hasta el momento actual no hemos utilizado sistemas en que haya más de una solución. De las treinta y cinco preguntas no todas poseen el mismo valor, las hay con mayor o menor e incluso preguntas eliminatorias. En el cuestionario se incluyen preguntas cuya contestación se limita a elegir entre una palabra, y preguntas cuyas contestaciones calibran todo el conocimiento del trayecto de un nervio, las diferentes integraciones de potenciales de un proceso nervioso, etc.; es decir que no solamente se examina cuantitativa sino cualitativamente y no sólo los conocimientos meramente informativos del alumno sino también conocimientos que exigen un cierto proceso razonado para la obtención del resultado correcto.

A parte de esta prueba de I.B.M., los alumnos que la superan, dada la índole eminentemente práctica de nuestra asignatura, realizan un examen sobre cadáver que permite objetivizar los conceptos teóricos, de una manera práctica.

Las fichas computadas, nos permiten al personal docente de la Cátedra poder realizar un examen estadístico de la utilidad de los métodos docentes utilizados, y un análisis de los diferentes factores que han influído en la obtención de unos resultados y otros, con lo cual variamos el método en relación a las exigencias de los diferentes factores, no siendo uno de los menos a tener en consideración, el número de alumnos y la psicología de los mismos."

Composición de espacios arquitectónicos con ordenadores

Participantes: A. Arranz, F. Briones, M. de las Casas Gómez, I. Fernández Florez, A. García Arringoa, E. García Camarero, A. García Quijada, A. Searle, J. Segui de la Riba.

SEMINARIOS

Valoración del Aprendizaje

Participantes: S. del Campo Urbano, E. García Camarero, M. Gil Gayarre, R. Moya Quiles, I. Ramos, V. Sánchez de Zavala.

Reuniones: 9 y 22 de enero.
Hora: 9 de la mañana

Comunicaciones:

En la reunión del 9 de enero V. Sánchez de Zavala presentó la siguiente comunicación:

Primer esbozo de un módulo de prueba automatizable y de su preparación automática (o semiautomática)

Parece necesario concretar las ideas generales que se han expuesto hasta ahora en un pequeño proyecto piloto. Siguiendo las indicaciones de nuestro compañero Moya, estoy tratando de preparar alguna prueba con la que se pudiera explorar el grado de "aprovechamiento" de los alumnos que siguieran un curso (imaginario) sobre un tema monográfico que fijemos; cuando esta tarea preliminar quede terminada (naturalmente, teniendo en cuenta las críticas e indicaciones de los demás asistentes al seminario), Moya se encargaría de estudiar con qué programa o programas podría ejecutarse en la máquina.

Por otra parte, carecería de todo interés preparar unas pruebas de esta índole mirando únicamente el tema concreto que se haya elegido: lo que queremos es encontrar métodos y técnicas generales. Por ello, partiendo de una decisión previa sobre lo que se quiere medir, y teniendo en cuenta, por más que de lejos, las carasterísticas previsibles y la posibilidad de ejecución real del proceso de medición, trato de preparar los "instrumentos de medida" correspondientes al ejemplo piloto elegido y, simultáneamente, intento obtener algunas reglas generales de construcción de pruebas análogas. (La presentación de los resultados obtenidos es casi inversa a la del proceso de obtención: formulo primeramente las reglas y principios generales y expongo después una "aplicación" al pequeño ejemplo piloto).

El estudio puede comenzarse, o bien esbozando unas características generales de los diversos tipos de pruebas ("instrumentos de medida") que deberían proponerse a lo largo del curso y estudiando las relaciones que debería haber entre ellas, o tomando una determinada y yendo hasta el detalle de su construcción y modo de utilizarla en un examen. He escogido esto último, temiendo perderme, si siguiese el primero, en vaguedades y, posiblemente, propuestas irrealizables. Parece lógico empezar por el primer tema o lección del imaginario curso previsto.

El contenido de éste será la teoría de la evolución biológica, para la que seguiré la a mi juicio excelente exposición de J. Maynard Smith en The Theory of Evolution, 2ª ed., Harmondsworth, Penguin, 1966.

Voy a suponer que se ha de realizar una prueba sobre el contenido del primer capítulo, cuyo título es "La adaptación". Como, verdaderamente, este concepto constituye íntegramente tal contenido, tenemos así un caso (que ha de ser frecuente a los comienzos de cualquier curso) en el que lo único que puede explorarse es la comprensión adquirida por los alumnos de un concepto fundamental. Podemos, pues, decir que vamos a preparar un módulo de prueba conceptual, y no un módulo de problema ni de ningún otro tipo.

Los instrumentos de medida y la medición

Los principios de la técnica de medición que parece necesario tener ahora en cuenta son esencialmente dos. El primero es el de la seriación de preguntas unida a la combinatoriedad irrestricta de las respuestas. Es decir, se ha de proponer una sucesión de preguntas encadenadas acerca de un problema, con objeto de ir penetrando en su comprensión por parte del alumno; y además, la respuesta que se dé a cada pregunta no ha de prejuzgar cuál haya de darse a la siguiente, con lo que las respuestas serán independientes, pese a versar sobre la misma cuestión, y la probabilidad de acertar por puro azar toda la sucesión valdrá aproximadamente el producto de las probabilidades correspondientes a cada pregunta, con lo que será poco menos que despreciable en cuanto el número de preguntas y de respuestas correspondientes a cada una de ellas sean un poquito elevados. (Cabe incluso la posibilidad, tal vez, de presentar varias series "en paralelo" y evaluarlas conjuntamente, con lo que la probabilidad de acierto aleatorio será todavía menor.) El otro principio que parece ineludible es el de las respuestas prefijadas: el alumno sólo tendrá que elegir entre cierto número de respuestas dadas a cada pregunta.

Con todo, es probable que sea muy conveniente dejar algún portillo por el que puedan "escapar" los alumnos más adelantados a la rígida sucesión de respuestas prefijadas; portillo que consistiría, simplemente, en una bifurcación en la sucesión a partir de una respuesta dubitativa, y a partir de la cual se plantearía una sucesión de preguntas de grado de dificultad más elevado.

Finalmente, me parece que es posible que sea útil concluir cada serie (o, al menos, las más importantes) con una definición que haya de hacer el alumno y con una pregunta sobre la metodología del concepto definido (que debería ser, desde luego, el que sirva de eje a toda la sucesión correspondiente). Así se podría tener, a la vez, una última comprobación de la no aleatoriedad del posible acierto y el examen mismo no carecería enteramente de eficacia pedagógica.

Veamos un método que parece prometedor para construir el instrumento de medida, en este caso un módulo de prueba conceptual. (Insisto en que este método no ha precedido a su ejemplificación con la teoría de la evolución, sino que ha rodado con ella). Tal método, como se verá, está pensado de modo que su aplicación sea lo más automática posible, a su vez, de forma que las ventajas de lo que en definitiva queremos obtener (corrección automática de pruebas de examen) no queden contrarrestadas por una labor excesiva para la puesta a punto de las pruebas mismas y, además y en consecuencia, por una gran rigidez en éstas, que prácticamente no podrían alterarse de un curso a otro (por lo cual - los alumnos podrían "aprenderse" las respuestas correctas poco menos que de memoria.)

El método ha de constar de dos fases (por lo menos): Una, A, de preparación, y - otra, B, de ejecución. Hoy no voy a tratar, prácticamente, más que la primera.

La fase A debería constar, creo, de cuatro partes:

la parte I, de planteamiento y análisis, consistiría en aislar y caracterizar el concepto eje sobre el que se centraría la sucesión de preguntas; habría que distinguir en él

1º, los rasgos básicos o fundamentales
2º, los conceptos contiguos (que lo limiten, que se caracterizarán por ciertos "no-rasgos" del c.-e.)
3º, los conceptos relacionados con él;

la parte II, de particularización, apuntaría a la formación de afirmaciones falsas relativas a la aplicación en casos concretos del c.e. (para montar sobre ellas un elenco completo de respuestas y, yendo hacia atrás, las preguntas correspondientes); constaría de tres operaciones:

1º, negar los rasgos básicos del c.e.
2º, igualar el c.e. a los conceptos contiguos
3º, desvincularlo de los conceptos relacionados con él;

la parte III, de definición, consistiría en plantear una serie de definiciones del c.e. a base de

a) los conceptos subsumentes (def. per genus prox. et diff. spec.)
b) los rasgos y (o) conceptos subsumidos (mera caracterización)
c) las características contextuales (def. "por abstracción")

(conviene advertir que parece necesario pedir al alumno que defina primero el - c.e. - en un espacio en blanco al efecto - y que luego "encaje" su definición - en uno de los varios esquemas que se le presenten, que se obtendrán de las definiciones por formalización parcial [= dejar huecos en los lugares ocupados por algunas de las palabras clave]);

la parte IV, metodológica, consistiría en formular una pregunta sobre las condiciones para aplicar científicamente el c.e. en un ejemplo concreto.

La fase B la dejo hoy en el siguiente esquema:

I, definición inicial del c.e. y su análisis
II, redacción de una o varias sucesiones exploratorias del c.e.
III, reducción a esquemas o "plantillas" de las def. propuestas al alumno
IV, redacción de la pregunta metodológica.

Ejemplo piloto de módulo de prueba conceptual

Siguiendo el capítulo citado del libro de J.M. Smith, se observa que el contenido correspondiente puede "analizarse" del siguiente modo:

1º equilibrio entre organismo y medio; adaptación de aquél a éste

2º la historia de la evolución viene a representar la historia de la adaptación de los organismos

3º cuándo puede decirse que una estructura es adaptativa?

a) por un análisis funcional "a priori"
b) por un análisis comprobativo, empírico

4º clases de adaptación

a) genética (muy lenta)
b) fisiológica (relativamente rápida)
c) comportamental (casi instantánea)

5º los cambios que no son adaptativos

6º diferencias entre las adaptaciones típicas de estructuras vivas distintas como son los animales y los vegetales.

A partir de este análisis se obtienen unas primeras contraposiciones, paralelizaciones y relaciones (reproduzco todos estos pasos que he ido dando al intentar la preparación de este instrumento en miniatura por si pudieran tener algún interés general):

adaptación	frente a	cambios padecidos (no útiles)
"	"	herencia
"	"	modificación activa del medio
"	y	equilibrio
"	"	evolución
"	"	estructura del organismo

adaptaciones instantáneas y a la larga

cuándo estamos justificados para calificar de adaptativo?

A I. Vamos a eliminar lo que no sea pertinente para obtener lo que queríamos distinguir en esta parte de la fase A. Rechazamos la relación entre adaptación y evolución (que se verá más adelante en el curso, y que ahora sólo podría confundir), así como la que haya entre adpt. y herencia, por análoga razón y porque, al ser ambiguo el concepto de adaptación (puede referirse lo mismo a individuos que a especies), no es posible especificar ahora tal relación.

Así nos queda lo siguiente:

1º, rasgos básicos de los cambios adaptativos:

a) mantienen a los organismos en equilibrio con su medio lo suficiente para hacer viable la especie

b) aumentan la viabilidad en el medio

2º, principales conceptos limitadores de tales cambios:

a) modificaciones que disminuyan la viabilidad

b) modificaciones transformadoras del medio (hay que considerar algunos casos especiales: castores, h.)

3º, conceptos relacionados más importantes:

a) tempo de las transformaciones

b) plasticidad morfológica de los organismos y su reactividad individual

A. II. Las particularizaciones contrapositorias son inmediatas; y es posible que se pudieran obtener automáticamente, por pura lógica de predicados, formulando explícitamente en ella lo obtenido en A I. - aunque tal vez de ese modo lo único que se consiguiese sería desplazar la dificultad (o sea, la intervención del profesor),

1º, "algunos cambios adaptativos no mantienen el equilibrio"

Las equiparaciones serían (con la misma obs. que antes):

2º, a) "hay cambios adaptativos que no aumentan la viabilidad"

b) "hay transformaciones útiles del medio por el organismo que son cambios adaptativos" (como he señalado antes, no puede rechazarse categóricamente este enunciado)

En cuanto a las desvinculaciones (id.), tendríamos:

3º, a) "sólo hay un tipo temporal de cambio adaptativo"

b) "todos los cambios adaptativos son del mismo tipo, independientemente de la estructura básica del organismo"

A III. Como definiciones podría proponerse las siguientes:

a) la adaptación es un cambio producido en las características de un ser vivo que le permite sobrevivir mejor

b) en la adaptación se modifica el ser vivo, éste tiene más posibilidades de supervivencia y el medio no queda transformado por su acción

b') la adaptación o es heredada, fija, o consiste en una alteración fisiológica, o se reduce a modificar una pauta del comportamiento (defectuosa, naturalmente, tal "defin."

c) se dice que hay adaptación a un medio cuando un ser vivo sobrevive mejor en él que en otros distintos

c') se dice que dos organismos están igualmente adaptados cuando sus tasas reproductivas en sus respectivos medios son iguales (parece ser una simple reducción de un concepto a otro, esta "def.").

A IV. Podría plantearse la cuestión de cómo determinar en caso concreto y determinado (un ejemplo) si cierta modificación - o estructura - observada es o no adaptativa.

En la próxima ocasión trataré de seguir, abordando la fase B.
V.S.Z.

Se decidió celebrar las reuniones los miércoles a las nueve de la mañana, en lugar de los jueves, como hasta ahora se venía realizando, manteniéndose sin embargo el período quincenal.

Composición de espacios arquitectónicos

Participantes: A. Arranz, M. de las Casas Gómez, I. Fernández Flórez, A. García Arangoa, E. García Camarero, J.M. de la Prada Poole, J. Seguí de la Riba.

Reuniones: Durante el mes de enero se celebró sólo la reunión del día 17. Las próximas reuniones serán el 14 y el 28 de febrero a las 9 de la mañana.

SEMINARIOS

Valoración del Aprendizaje

Participantes: S. del Campo Urbano, E. García Camarero, L. García-Ramos, M. Gil Gayarre, J.A. Martínez Carrillo, R. Moya Quiles, I. Ramos, V. Sánchez de Zavala.

Reuniones: 5 y 19 de febrero

Comunicaciones:

J.A. Martínez Carrillo expuso sus experiencias en problemas de valoración del aprendizaje desarrolladas en la Universidad de Valencia, y en particular desarrolló el método de Anderson sobre la estimación de parámetros en el análisis de estructuras latentes.

L.A. García-Ramos nos manda las siguientes notas sobre los programas de evaluación automática de calificaciones existentes en el Instituto Químico de Sarriá:

El reto que plantea la enseñanza debido a la masificación que actualmente se está produciendo exige que los métodos y modos de contacto del profesor con el alumno se adapten a las nuevas situaciones. La solución no es universal ni simple o al menos no es previsible que lo sea en un futuro inmediato, por lo que es necesario ensayar esquemas parciales que en un primer estadio cumplan la función primordial de una recogida de datos que permitan plantear sistemas de mayor amplitud.

En cuanto al aspecto de la evaluación automática de calificaciones, en el Instituto Químico de Sarriá se han realizado estudios, plasmados en programas de cálculo, restringidos por el momento a algunas asignaturas de tipo eminentemente práctico. Concretamente desde hace tres años se han utilizado programas de este tipo aplicados al Laboratorio de Análisis Cuantitativo, modificados según los resultados obtenidos en etapas previas. En esencia, el esquema para el procesamiento de los datos es el siguiente:

El número de prácticas es de unas 50-60 por curso, por lo que los alumnos suelen entregar los resultados a un ritmo medio de 2 semanales. Los resultados se entregan en fichas preperforadas, una por cada alumno, en las que aparecen los siguientes datos: Nº de alumno, Nº de práctica, Resultado (mg), Límite de confianza, Desviación Standard, Nº Observaciones y Error de Método. Los datos relativos a los errores estimados no se introducen en el cálculo por el momento estando en estudio su significación real. No obstante, se dispone de ellos para el momento oportuno de su utilización.

Como datos a aportar por el encargado de las prácticas están: el Nº máximo de alumnos, el Nº de práctica, el valor correcto del resultado (mg/ml) y la lista del volumen de problemas entregado a cada alumno.

La primera fase del programa hace un análisis de consistencia de los datos, eliminando las fichas que contengan algún error específico (nº de práctica diferente al analizado, duplicidad de nº de alumno, etc.) A continuación se asignan notas directamente a los alumnos que no han presentado resultados, pasándose seguidamente a la primera parte de la elaboración estadística de los resultados. Se hace una normalización pasándose los resultados en peso (mg) a una escala relativa (mg/ml). Se halla una primera media $\bar{x}_1$ y la desviación típica correspondiente σ_1. Considerando que si se supone una distribución gaussiana para una probabilidad del 95% la desviación para cada punto Xi es de 1,96 σ, se ha tomado como criterio para eliminar valores aberrantes el siguiente: $|\bar{X}_1 - Xi| > 2\sigma_1$. Una vez separados de la población los valores aberrantes, se aplica un baremo de coeficientes que dan la calificación en función del mayor o menor acercamiento al valor de $2\sigma_1$. Se calcula de nuevo una media $\bar{X}_1$ y una desviación típica σ'_1 corregidas y se calculan los coeficientes de desviación para cada resultado: $Zi = |\bar{X}'_1 - Xi| / \sigma$. En una versión inicial del programa a partir de los valores de Zi se hacía una distribución por clases y de ahí se establecían las calificaciones. En la versión actual en función de los resultados obtenidos en años anteriores se ha logrado establecer un baremo para asignar las calificaciones en función de Zi.

En la segunda parte de la elaboración estadística se introduce como media $\bar{X}_2$ el valor obtenido por el encargado de las prácticas, valor al que se somete a pruebas de garantía para asegurar un nivel de confianza adecuado. Se efectúa un ciclo análogo al de la primera parte, con la diferencia que no se reajusta la media $\bar{X}_2$. Una vez obtenidas las dos calificaciones para cada alumno se almacenan en un fichero de disco para su posterior utilización, y se editan los resultados. Todo el proceso dura, para un grupo de 70 alumnos, unos 10 minutos con lo que el alumno puede visualizar casi de inmediato sus resultados y extraer las conclusiones pertinentes. Se tiene en estudio incluir la información concerniente a nº de experiencias, límite de confianza y error promedio como factores de corrección. Es de notar que los resultados obtenidos según la media del curso y según el verdadero valor son bastante similares en la mayoría de casos.

Otro programa que está en una fase inicial de experimentación está orientado hacia exámenes rutinarios de frecuencia quincenal o mensual en la asignatura de Métodos Numéricos y Gráficos de Cálculo en los cuales interesa primordialmente la agilidad de cálculo del alumno en los tópicos de carácter standard. Al igual que en el caso anterior el alumno presenta los resultados sobre fichas preperforadas en el número que sea necesario para cubrir las respuestas del examen (un programa admite hasta 24 respuestas). En cada ficha se asignan datos de control tales como nº de alumno, grupo, número de examen y nº de ficha,

asímismo en cada ficha existe espacio para tres respuestas numéricas de formato análogo a la notación FORTRAN para cantidades en punto flotante normalizado. Como datos a aportar por el profesor están: número de respuestas, nº de examen, el nº de tolerancias y para cada pregunta, el valor correcto, la puntuación de la pregunta, el nº de cifras significativas y un factor de descuento por cada cifra dada de más o de menos, los márgenes de las tolerancias (hasta cuatro) y los coeficientes asignados para cada margen. Normalmente todo se establece para trabajar en porcentajes.

Los cálculos a efectuar son sencillos y se editan en forma de tabla. Al final se hallan las medias y desviaciones standard para cada pregunta y para el conjunto de la prueba.

Bibliografía.

D. D. BUSHNELL y D. W. ALLEN
The Computer in the American Education
John Wiley 1967

S.R. SMITH, R. Schor y P.C. DONOHUE
Computer Grading of Small Numbers of Laboratory Unknowns
Journal of Chemical Education 42, 224 (1965)

C.C. HINCKLEY y J.J. LAGOWSKI
A versatile Computer-Graded Examination
Journal of Chemical Education 43, 575 (1966)

G.A. CHAPPELL y R.M. Miller
Computer-Graded Qualitative Analysis
Journal of Chemical Education 44, 79 (1967)

N.A. FRIGERIO
Notes on Computer Grading of Examinations
Journal of Chemical Education 44, 413-4 (1967)

J.A. MANN JR, H. ZEITLIN y A.B. DELFINO
A Computer Centered Chemistry Records and Grading System
Journal of Chemical Education 44, 673-7 (1967)

N.D. YANEY
An Automated System for Testing and Record Keeping
Journal of Chemical Education 44, 667-8 (1967)

G.F. POLLNOW
SCORAN: A computerized Test Scorer and Analyzer
Journal Chemical Education 44, 679-80 (1967)

N. McKAIG JR
Grading Quantitative Analysis Laboratory Reports
Journal of Chemical Education 44, 169-72 (1967)

R.H. SCHWENDEMAN
Standardization of Grades in Multi-Section Courses
Journal of Chemical Education 45, 129-31 (1968)

J.F. ALTENBURG, L.A. KING y C. CAMPBELL
Computer Grading of General Chemistry Laboratory Reports
Journal of Chemical Education 45, 615-6 (1968)

L.A. G-R.

68 ALUMNOS NO. 6 Valoración de ácido acético (68-69)

Alumnos sin Resultados	Nota
36	0
49	0

RESULTADOS SOBRE LA MEDIA DEL CURSO

Alumnos con Resultados Aberrantes

Alumno	MGR.	Desv. Promedio	Nota
4	577.0	5.32	11.8
9	551.9	5.60	10.8
53	520.9	-3.18	26.4
54	782.0	6.04	10.0

MEDIA = 31.191654 $\frac{mg}{ml}$ Desv. Standard = 1.463923

ALUMNOS CON RESULTADOS DENTRO DE NORMA

Alumno	MGR.	COEFICIENTE	Nota
1	423.0	9.41	56.4
2	498.4	- 2.29	88.5
3	665.0	1.18	94.0
5	526.2	- 0. 18	99.0
6	626.0	3.72	81.3
7	427.8	2.89	85.5
8	328.6	3.67	81.6
10	519.0	-10.92	52.6
11	509.0	- 5.52	72.3

Alumno	MGR.	COEFICIENTE	Nota
12	600.4	3.39	83.0
13	510.6	- 1.47	92.6
14	401.6	- 0.86	95.6
15	775.0	- 2.42	87.8
16	569.6	1.76	91.1
17	573.8	2.42	87.8
18	538.0	5.55	72.2
19	562.0	-24.22	37.8
20	582.0	5.52	72.3
21	305.4	- 4.83	75.8
22	528.4	- 2.79	86.0
23	510.1	3.66	81.6
24	622.0	- 0.27	98.6
25	370.4	1.76	91.1
26	635.0	- 6.61	66.9
27	324.0	16.12	43.8
28	579.2	- 5.46	72.6
29	332.0	- 8.87	57.8
30	346.5	- 9.45	56.3
31	911.3	-21.94	39.0
32	646.0	18.67	41.3
33	558.0	-18.04	41.9
34	641.5	- 3.10	84.4
35	738.8	3.78	81.0
37	493.2	0.55	97.2
38	687.2	1.41	92.9
39	595.6	2.41	87.9
40	551.9	12.60	48.9
41	666.0	- 1.53	92.3
42	663.0	0.13	99.3
43	577.4	0.83	95.8
44	757.0	2.06	89.6
45	584.9	21.30	39.3
46	659.4	6.54	67.2
47	473.0	6.35	68.2
48	599.0	4.41	77.9
50	688.2	5.13	74.3
51	668.5	3.03	84.8
52	302.0	-12.04	49.9
55	658.8	1.19	94.0
56	625.5	5.21	73.9
57	511.0	2.13	89.3

Alumno	MGR.	COEFICIENTE	N o t a
58	598.0	- 5.17	74.1
59	623.1	- 1.38	93.0
60	637.6	- 1.88	90.5
61	590.3	- 0.71	96.4
62	699.6	- 9.17	57.0
63	583.9	4.77	76.1
64	749.2	- 1.55	92.2
65	672.0	- 0.01	99.9
66	668.0	1.13	94.3
67	683.0	- 2.29	88.5
68	716.0	0.29	98.5

MEDIA = 30.9692 Desv. Standard = 0.6985 Desv. Standard de la media= 0.0887

RESULTADOS SOBRE EL VALOR VERDADERO

Alumnos con Resultados Aberrantes

Alumno	MGR.	Desv. Promedio	N o t a
4	577.0	5.51	11.3
9	551.9	5.79	10.3
53	520.9	-2.99	29.4
54	782.0	6.23	10.0

MEDIA = 31.000003 Desv. Standard = 1.476616

ALUMNOS CON RESULTADOS DENTRO DE NORMA

Alumno	MGR.	COEFICIENTE	N o t a
1	423.0	9.64	55.8
2	498.4	- 2.81	85.9
3	665.0	0.89	95.5
5	526.2	- 0.56	97.1
6	626.0	3.59	82.0
7	427.8	2.71	86.4
8	328.6	3.54	82.2
10	519.0	-11.99	50.0
11	509.0	- 6.24	68.7
12	600.4	3.24	83.7
13	610.6	- 1.93	90.3
14	401.6	- 1.29	93.5
15	775.0	- 2.95	85.2

Alumno	MGR.	COEFICIENTE	N o t a
16	569.6	1.50	92.4
17	573.8	2.21	88.9
18	538.0	5.54	72.2
19	562.0	-26.14	36.9
20	582.0	5.51	72.4
21	305.4	- 5.51	72.4
22	528.4	- 3.34	83.2
23	510.1	3.53	82.3
24	622.0	- 0.65	96.7
25	370.4	1.51	92.4
26	635.0	- 7.40	62.9
27	324.0	16.79	43.2
28	579.2	- 6.18	69.0
29	332.0	- 9.81	55.4
30	346.5	-10.42	53.9
31	911.3	-23.71	38.1
32	646.0	19.50	40.4
33	558.0	-19.56	40.4
34	641.5	- 3.67	81.6
35	738.8	3.65	81.7
37	493.2	0.22	98.8
38	687.2	1.13	94.3
39	595.6	2.19	89.0
40	551.9	13.03	48.2
41	666.0	- 1.99	90.0
42	663.0	- 0.22	98.8
43	577.4	0.51	97.4
44	757.0	1.82	90.8
45	584.9	22.30	38.8
46	659.4	6.59	67.0
47	473.0	6.39	68.0
48	599.0	4.33	78.3
50	688.2	5.09	74.5
51	668.5	2.85	85.7
52	302.0	-13.18	48.0
55	658.8	0.90	95.4
56	625.5	5.18	74.0
57	511.0	1.90	90.4
58	598.0	- 5.87	70.6
59	623.1	- 1.84	90.7

Alumno	MGR.	COEFICIENTE	N o t a
60	637.6	- 2.37	88.1
61	590.3	- 1.13	94.3
62	699.6	-10.13	54.6
63	583.9	4.70	76.4
64	749.2	- 2.02	89.8
65	672.0	- 0.38	98.0
66	668.0	0.83	95.8
67	683.0	- 2.80	85.9
68	716.0	- 0.05	99.7

MEDIA = 31.0000 Desv. Standard = 0.6773 Desv. Standard de la Media = 0.0833

Composición de Espacios Arquitectónicos

Participantes: A. Arranz, M. de las Casas Gómez, I. Fernández Florez, A. García Arangoa, E. García Camarero, J.M. de la Prada Poole, J. Segui de la Riba, N. Spagnuolo.

Reuniones: Durante febrero se realizaron las reuniones los días 14 y 28.
Las próximas reuniones serán los días 14 y 28 a las 9 de la mañana.

Comunicaciones:

Seguí de la Riba envió las siguientes notas:

Confeccionada la lista de funciones primarias se procedió a su examen, precisión y corrección para una vez puntualizadas relacionarlas con los espacios mínimos necesarios.

Resumen del análisis de funciones.

Notas y consideraciones.

Funciones vegetativas.

Son las que reciben la incidencia de las transformaciones genéticas.

Están perfectamente autocontroladas.

- Las directamente ligadas con la homeostasis biológica y se satisfacen periódicamente con frecuencias medibles (alimentarse, exonerar, respirar, transpirar).

SEMINARIOS

Valoración del Aprendizaje

Participantes: S. del Campo Urbano, E. García Camarero, L. García-Ramos, M. Gil Gayarre, J.A. Martínez Carrillo, R. Moya Quiles, I. Ramos, V. Sánchez de Zabala.

Reuniones: 5 y 19 de marzo

Comunicaciones:

En la reunión del 5 de marzo el señor García Pérez, expuso un procedimiento analítico de evaluación de exámenes para proveer plazas de profesorado, teniendo en cuenta las técnicas de selección industrial de trabajadores.

En la reunión del 19, V. Sánchez de Zavala expuso su comunicación PRIMER ENSAYO DE EJECUCION Y FORMALIZACION DE UN MODULO DE PRUEBA CONCEPTUAL, centrando con ello el tema de la reunión alrededor de la necesidad de una formalización de un módulo de valoración y la forma de automatizarlo. El mismo realiza la crítica, que expone en su comunicación, con cuatro puntos. Se discuten los distintos item. Al primero se concluye que la investigación de la computabilidad del modelo lógico resolverá el problema de la complejidad del método práctico. Se discuten los puntos 2º, 3º y 4º. Se supone que una extensión y generalización del modelo que empieza a definirse con la comunicación de Sánchez de Zavala permita superar las dificultades que plantean.

El profesor Gil Gayarre insiste en la dificultad del objetivo final: la calificación positiva o negativa de un complejo de ejercicios realizados por un alumno y su standarización y homogeneización a distintos criterios y distintas disciplinas. Expone que los últimos ejercicios de los alumnos de Terapéutica Física van a ser enviados a 6 profesores de distintas Universidades para ver hasta qué punto coinciden los criterios tradicionales y los del análisis automático que se han realizado en el CCUM.

J.A. Martínez Carrillo propone hasta qué punto los problemas planteados están relacionados con las investigaciones de WOODGER (Lenguage and Biology, Oxford University) por un lado. Se discute la validez de métodos como el del "diferencial semántico" de Osgood y de todos aquellos que parten de la pura investigación estadística de los datos empíricos. Queda planteado el problema de como enfrentar un modelo lógico-matemático a los problemas prácticos de la calificación. Se insinúa la posibilidad de realizar un modelo probabilístico que se discutirá más adelante.

A continuación incluimos la comunicación de V. Sánchez de Zavala:

Primer ensayo de ejecución y formalización de un módulo de prueba conceptual

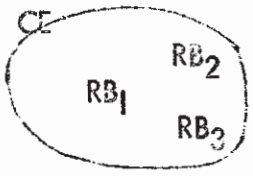
Voy a tratar de realizar la que el día anterior llamaba fase B en la preparación de un módulo de prueba conceptual, apoyándome para los ejemplos en el caso de la adaptación, tal y como había quedado analizado, particularizado (contrapositoriamente) y definido tal concepto (que llamaba "concepto eje" del capítulo 1º del libro de Maynard Smith sobre la teoría de la evolución).

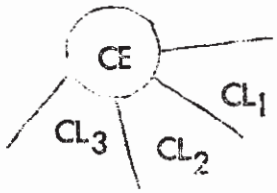
Preliminares

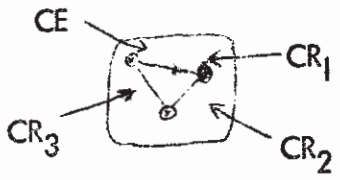
Teniendo en cuenta lo dicho el día anterior deberíamos partir de lo obtenido en la parte A II; sin embargo, como ya indicábamos, parece conveniente tratar de formalizar esta obtención, único modo de poder obtener en forma reproducible y rápida las particularizaciones que nos interesan, y sobre las que se han de montar las preguntas que se hagan al alumno y los elencos de respuestas que se le propongan. (Adviértase que una característica esencial de estas pruebas, para que no sean puramente abstractas y verbales, ha de ser la de ejemplificar, ésto es, proponer casos de aplicación al concepto que sea).

Intento de formalización

Como ya dije, vamos a dividir los conceptos que entren en juego con el "concepto eje" (CE) en tres grupos, rasgos básicos (RB), conceptos limitadores (CL) y conceptos relacionados (CR). Sus relaciones intensionales con el CE pueden simbolizarse con los esquemas siguientes:







(en los que he limitado arbitrariamente a tres en cada caso, por sencillez, los conceptos emparentados con CE). En el tercer esquema, los puntos con cruz simbolizan cualesquiera entidades en entren en una definición satisfactoria de CE.

A) - Rasgos básicos

Parece que la fórmula más sencilla representativa de las relaciones entre CE y los RB sería algo así como

$$\forall x \quad [CE\ (x) \longrightarrow RB_i\ (x)] \qquad\qquad i = 1, 2 \ldots n$$

(siendo n el número de rasgos básicos).

Pero hay que distinguir rasgos necesarios de rasgos suficientes. *Una forma de hacer esto consiste en agrupar todos los rasgos necesarios en la conyunción.*

$$\bigwedge_i RB_i\ (x) \equiv_{DEF} \bigwedge RB_1\ (x) \quad RB_2 \quad \bigwedge \ldots \bigwedge \quad RB_m\ (x)\ ; \quad \text{sii}$$

entonces tendremos como antes; *suponiendo que entre todos ellos formen un rasgo suficiente*

$$\forall x \quad [CE\ (x) \longrightarrow \bigwedge_i RB_i\ (x)]$$

y además, particularizando

$$\forall x \quad [CE\ (x) \longrightarrow RB_i\ (x)]$$

Esto último nos permitirá ejemplificar con el "objeto (= caso al que podría cuadrar el concepto eje) a:

$$CE\ (a) \longrightarrow RB_i\ (a)\ ,$$

y también

$$\neg RB_i\ (a) \longrightarrow \neg CE\ (a)\ ;$$

asimismo, si queremos obtener una proposición existencial (muy útil, sin duda, para los ejemplos), podemos poner la siguiente:

$$* \quad \exists\ x\ [CE\ (x) \wedge \neg RB_i\ (x)]$$

(que, obsérvese muy bien, es falsa si la proposición universal de partida es verdadera; por lo cual la he antepuesto un asterisco que la distinga).

Vamos a ocuparnos ahora de los rasgos suficientes. Con una definición correlativa a la anterior, o sea, con

$$\bigvee_i RB'_i\ (x) \equiv_{DEF} \quad RB'_1\ (x) \vee RB'_2\ (x) \vee \ldots \vee RB'_p\ (x)\ ,$$

podemos escribir

$$\forall x \; [CE\,(x) \longrightarrow \vee_i \, RB_i'\,(x)]$$

Pero ahora no es posible particularizar a un valor del subíndice $\underline{i}$. Lo que sí podemos hacer, por ejemplo, es contraponer:

$$\forall x \; [\neg \vee_i \, RB_i'\,(x) \longrightarrow \neg CE\,(x)] \; ,$$

y aplicar luego una ley de De Morgan, con lo que llegamos a

$$\forall x \; [\wedge_i \neg RB_i'\,(x) \longrightarrow \neg CE\,(x)] \; ,$$

fórmula que podemos "ejemplificar", es decir, aplicar a un objeto determinado:

$$\wedge_i \; \neg RB_i'\,(a) \longrightarrow \neg CE\,(a) \; ;$$

y, como antes, en forma existencial podremos escribir

$$* \quad \exists x \; [\wedge_i \neg RB_i'\,(x) \wedge CE\,(x)] \; .$$

(Como es natural, cada rasgo necesario puede componerse de varios rasgos cuasi--suficientes, que habrá que ligar mediante conyunción, como acabamos de ver; o también puede estar formado cada rasgo suficiente por varios rasgos cuasi-necesarios, que, según hemos visto antes, podremos sustituir en las fórmulas que nos interesan por uno cualquiera de ellos.)

Hemos visto que los rasgos básicos llamados suficientes ofrecen la desventaja de que hay que tomarlos siempre todos juntos (o, mejor dicho, sus negaciones); esta dificultad puede orillarse siempre que sea posible especificar en qué situación basta cada uno para caracterizar a CE, o, lo que es lo mismo, con tal de que pueda escindirse este último concepto en una serie de ellos, cada uno correspondiente a unas circunstancias o situación dados (por ejemplo, el concepto de adaptación puede desdoblarse en el de las adaptaciones cuando el medio no varíe y el de adaptaciones a un medio variable).

Llamemos S_k a la situación o condiciones que tenga que cumplir el "objeto para que podamos considerarle candidato al cumplimiento de un CE caracterizado por RB_k ; tendremos:

$$\forall x \left\{ [S_1\,(x) \rightarrow (CE\,(x) \rightarrow RB_1'\,(x))] \wedge [S_2\,(x) \rightarrow (CE\,(x) \rightarrow RB_2'\,(x))] \wedge \ldots \wedge [S_m\,(x) \rightarrow (CE\,(x) \rightarrow RB_m'\,(x))] \right\}$$

De aquí se obtiene inmediatamente, por particularización,

$$\forall x \left\{ S_k (x) \dashrightarrow \left[CE (x) \longrightarrow RB'_k (x) \right] \right\} \quad ,$$

de modo que cabe ejemplificar y existencializar sin la menor dificultad:

$$S_k (a) \longrightarrow \left[CE (a) \longrightarrow RB'_k (a) \right]$$

$$S_k (a) \longrightarrow \left[\neg RB'_k (a) \longrightarrow \neg CE (a) \right]$$

$$* \exists x \left\{ S_k (x) \wedge \neg \left[CE (x) \longrightarrow RB'_k (x) \right] \right\} \text{ , de donde}$$

$$* \quad \exists x \left[S_k (x) \wedge CE (x) \wedge \neg RB'_k (x) \right] .$$

B) - Conceptos limitadores

En este caso es evidente que podremos escribir

$$\forall x \left[CE (x) \longrightarrow \neg CL_l (x) \right]$$

$$\forall x \left[CL_l (x) \longrightarrow \neg CE (x) \right] ,$$

fórmulas de las que obtenemos, respectivamente

$$CE (a) \longrightarrow \neg CL_l (a)$$

$$* \quad \exists x \left[CE (x) \wedge CL_l (x) \right]$$

y

$$CL_l (a) \longrightarrow \neg CE (a)$$

$$* \quad \exists x \left[CL_l (x) \wedge CE (x) \right]$$

(las dos proposiciones existenciales son, naturalmente, idénticas, ya que ambas expresan que la intersección de los campos de aplicación de ambos conceptos es no-nula).

C) - Conceptos relacionados

(Vamos a dejar enteramente de lado estos conceptos, ya que su tratamiento formal presenta, a primera vista, grandes dificultades).

Ejemplos

En nuestro caso, CE será la noción de cambio adaptativo; vamos a limitarnos esta vez a los rasgos básicos, para simplificar (aunque véanse los apartados "Crítica" y "Difi--

cultades generales"). Teniendo en cuenta que los rasgos básicos de este CE que distinguimos el día anterior parecían componerse adyuntivamente para caracterizarlo (o sea, ser suficiente cada uno de ellos, no necesario), especificaremos lo siguiente:

S_1 cambio real (consecutivo a una alteración del medio)

S_2 cambio virtual (comparado con otras condiciones o lo que suceda en otras especies)

RB_1 conservación del equilibrio con el medio (lo suficiente para que sea viable la especie).

RB_2 aumento de la viabilidad de la especie.

Ejemplo 1º

Designaremos con a la estivación (= permanencia durante el verano del animal inmóvil o casi, con las funciones fisiológicas principales a ritmo muy lento y la temperatura corporal a escasos grados C por encima de la ambiente); y vamos a admitir que

S_2 (a)

La prueba podría rezar algo así como:

"En el desierto de Mohave, en California, los inviernos son relativamente templados y bastante secos, y los veranos tórridos y muy secos; se ha observado que ciertas especies de perritos de los desiertos estiva (aquí la explicación correspondiente), pero no sabemos si hiberna o no. En estas condiciones,

1º.- Llamaría Vd. adaptación al comportamiento del estivar?

a) - sí
b) - sí, si además hiberna; pero si no hiberna, no
c) - no, si además hiberna, pero si no hiberna, sí
d) - no

2º.- Por qué?

a) - Porque así, mayor número de animales vive con menor cantidad de alimento y agua
b) - Porque así los mecanismos de regulación de la temperatura corporal siguen las variaciones de la estación
c) - Porque la estivación y la hibernación son fenómenos contrarios en estaciones contrarias

d) - Porque así, por pasar los animales mucho tiempo inmóviles, los depredadores los capturan más fácilmente.

3º.- Supongamos que el enunciado de la cuestión hubiera sido distinto; en qué casos, precisamente, hubiera dado Vd. a la primera pregunta una respuesta contraria a la que haya dado?

Si el enunciado hubiese indicado

a) - que los abrigos donde los animales estivan estaban muy ocultos, prácticamente invisibles
b) - que durante el invierno el desierto de Mohave era húmedo y estaba lleno de vida
c) - que la estivación era muy breve
d) - (en ninguno de los casos anteriores).

Como se trata de explorar la comprensión del concepto de adaptación (en lo que se refiere a sus rasgos básicos), parece oportuno no valorar las respuestas con una mera dicotomía de "mal" y "bien", sino admitir la posibilidad de una comprensión imperfecta; y teniendo en cuenta que ni siquiera la conjunción de las respuestas correspondientes a las dos primeras preguntas indica siempre inequivocamente si se ha comprendido o no (por eso he introducido la tercera pregunta), introduciendo los símbolos valorativos.

2 - bien
1 - regular
0 - mal
- - (demora de la valoración),

pueden, tal vez, admitirse las siguientes tablas:

	a	b	c	d
A	2	-	0	0
B	0	0	0	0
C	0	0	0	0
D	0	0	0	-

	aA	bA	dD	todas las demás combinaciones
A	0	0	1	0
B	0	1	0	0
C	0	0	0	0
D	2	0	0	0

(Suponiendo que haya conseguido hacer las tres series de respuestas verdaderamente independientes, la probabilidad de obtener un 2 al azar es $1/4^3 = 1/64$ eq. 1,6%, la de obtener 1 al azar es 2/64 eq. 3,1%, y la de sacar al azar una nota no cero es - 1/21 eq. 4,75%).

Ejemplo 2º

Vamos a designar con $\underline{a}$ el aumento del tamaño de las branquias de los renacuajos en aguas contaminadas, y admitamos que

$$S_1 (\underline{a}) .$$

La prueba podría decir así:

"En una investigación sobre el ciclo vital de los anfibios de cuyos huevos, agrupados en masas gelatinosas adheridas a ramitas sumergidas, hacen rápidamente eclosión renacuajos, se observó que en las aguas contaminadas por desechos fabriles los renacuajos - de ciertas familias de ranas tenían branquias de tamaño mayor que el normal. A este respecto se han formulado diversas interpretaciones, que son, en lo esencial, las siguientes:

a) - este fenómeno debe estimarse adaptación, ya que así pueden subsistir los renacuajos pese a la pobreza de oxígeno.

b) - no debe estimarse adaptación, porque los renacuajos al llegar a adultos reabsorben mal las branquias, y mueren poco después de la puesta de los huevos

c) - sólo debería estimarse adaptación si se comprobase un aumento progresivo de la proporción de renacuajos con branquias grandes

d) - si fuese adaptación a ese medio contaminado la población de renacuajos de esos ríos debería aumentar con respecto a los de aguas limpias;

1º) - cuál respuesta le parece más acertada?.
2º) - por qué?.

a) - porque las branquias grandes responden mejor a las nuevas condiciones del medio

b) - porque la adaptación tiene que limitarse a esa fase del ciclo vital, que es cuando se respira oxígeno del agua (con las branquias grandes)

c) - porque en la adaptación lo que cuenta es la población hasta llegar a la madurez sexual

d) - porque son los renacuajos los que, si están adaptados, no han de morir más fácilmente, sino menos".

Parece que podría darse la siguiente tabla valorativa:

	a	b	c	d
A	1	0	1	0
B	0	0	0	0
C	1	0	2	0
D	0	0	0	0

(Las probabilidades de acertar aleatoriamente son ahora, naturalmente, mucho - mayores que antes. Pudieran combinarse dos ejemplos sobre el mismo CE, sin embargo, multiplicando entre sí los resultados valorativos, y con dos preguntas en cada uno y el supuesto de la independencia - ahora mucho más verosimil, a poco diferentes de aspecto que sean los dos ejemplos - podría llegarse a probabilidades inferiores al 1%).

Crítica

1º.- El procedimiento empleado es muy laborioso, por mucho que se guie uno de las fórmulas obtenidas en el apartado sobre formalización.

2º.- Intervienen nuevos conceptos, sin que se sepa bien cómo aparecen y qué papel desempeñan en las posibles respuestas

3º.- Es fácil que las preguntas (por efecto de lo anterior) sean oscuras y propensas a extraviar al alumno

4º.- Como consecuencia de lo anterior, no es seguro que las tablas de valoración - respondan a las relaciones existentes entre los conceptos que se hayan confundido o distinguido entre sí

Dificultades generales advertidas y su posible solución

1º.- Como se trata de ejemplificar un solo concepto (CE) en unos casos prácticos o ejemplos, las únicas preguntas que pueden plantearse de manera no muy retorcida son del tipo

es éste un caso del (CE) ?,

a las cuales las únicas respuestas que se ofrecen naturalmente al constructor del "instru-- mento" son

sí es un caso de (CE)
no es un caso de (CE),

en lugar del mínimo de cuatro que serían necesarias.

2º.- Habrá pues, que hacer entrar en juego otros conceptos emparentados con el CE (por ejemplo, en lugar de plantear primeramente preguntas relativas a los RB, y luego a los CL, podrían emplearse unos y otros en las mismas preguntas), que es lo que, de he-- cho, he tenido que hacer, pero a tientas, en los dos ejemplos presentados.

3º.- Parece conveniente averiguar cuál es el mínimo de conceptos emparentados que habría que manejar para obtener el número de respuestas posibles que se buscase (y, acaso, el de preguntas en serie con respuestas independientes entre sí), así como la mecánica combinatoria que habría que aplicar para sacar automáticamente de la definición del CE los esquemas correspondientes a todas esas preguntas y respuestas.
V.S. de Z.

SEMINARIOS

Valoración del Aprendizaje

Participantes: S. del Campo Urbano, E. García Camarero, L. García-Ramos, M. Gil Gayarre, J.A. Martínez Carrillo, R. Moya Quiles, I. Ramos, V. Sánchez de Zavala.

Reuniones: 16 y 30 de abril

Comunicaciones:

A continuación reproducimos la comunicación de Sánchez de Závala en la reunión del día 16:

Voy a tratar de resolver las dificultades planteadas el día anterior; o sea, a intentar presentar un método que permita obtener formalmente, dado un CE y varios conceptos emparentados con él, toda una serie de respuestas a preguntas que se puedan plantear respecto a pretendidos ejemplos de sistemas o individuos subsumidos en el CE; y si el método que ofrezco es satisfactorio, tendremos, ipso facto, información sobre el número de respuestas que puedan plantearse a los alumnos (para la elección, entre ellas, de una respuesta acertada a la pregunta en cada caso formulada), y, a la inversa, información sobre el número de conceptos con que habrá que jugar en cada caso para obtener una gama de respuestas en número prefijado.

FORMULAS GENERALES

1. Simbología

Con objeto de simplificar las fórmulas, vamos a partir de un solo rasgo necesario (que puede ser, por supuesto, un rasgo suficiente con tal de que hayamos aislado las circunstancias o situación, S_k, en que tal rasgo, RB_k', sea necesario para que un "objeto" satisfaga al CE: véase el trabajo anterior, apartado "Intento de formalización"). También con el mismo objeto vamos a eliminar la primera letra de cada una de las siglas empleadas el día anterior, con lo que

E será el concepto eje

B será el rasgo básico considerado (en la situación apropiada, si es necesario)

L_i será un rasgo limitador (es decir, uno de los ragos que caractericen a los conceptos limitadores del concepto eje; pero no habrá de consistir meramente en la negación de B)

r_i será un rasgo caracterizador de un concepto relacionado con el concepto eje, rasgo que, por presentar, en general, una magnitud variable (cuando un objeto satisfaga el concepto eje, y dependiendo del valor variable que presente otra magnitud característica), explicitaremos representándolo por

$r_i(k_i)$

Vamos a llamar también

C_i a las circunstancias o magnitudes características acabadas de mencionar, o sea, a las correspondientes al concepto relacionado con r_i; y explicitando también ahora la variabilidad, escribiremos

$C_i(k_i)$

Finalmente, representaremos con

G al conjunto de características comunes al rasgo básico considerado (B) y a los rasgos limitadores, L_i.

Por otra parte, representaremos con

$*$ la indicación de que la proposición simbólica situada a la derecha de este signo contradice los supuestos de partida, y con

$-$ la de que puede estar en contradicción o no con ellos, dependiendo de cuáles sean las demás circunstancias (no mencionadas en tal proposición);

no haremos explícito, en cambio, el signo usual de aserción que podría acompañar a las proposiciones simbólicas que estén de acuerdo con los supuestos de partida.

2. Gamas de respuestas

En todos los casos que siguen partiremos del supuesto de que se quieren conseguir, por lo menos, 4 respuestas por cada pregunta formulada en el módulo; y seguiremos admitiendo, como hasta ahora, que lo que se hará es plantear un posible caso práctico al alumno, que él habrá de resolver escogiendo una respuesta que lo clasifique (dentro de unos

conceptos determinados, atribuyéndole ciertos rasgos característicos, etc.).

Las fórmulas generales que vamos a presentar tendrán el esquema

$$I \longrightarrow II$$

que, desde luego, indicará que "I" implica "II"; la pregunta consitirá en formular que se cumple uno de los dos miembros en el ejemplo o caso presentado y ofrecer la posibilidad de que se cumpla el otro; y puesto que queremos tener cuatro respuestas posibles, los esquemas se presentarán así

$$\begin{array}{l} I \longrightarrow II \\ I \longrightarrow II' \\ I \longrightarrow II'' \\ I \longrightarrow II''' \end{array}$$

(se afirma el 1º y se presentan a elección las variantes del 2º)

o bien así

$$\begin{array}{l} I \longrightarrow II \\ I' \longrightarrow II \\ I'' \longrightarrow II \\ I''' \longrightarrow II \end{array}$$

(se afirma el 2º y se presentan a elección las variantes del 1º)

A) Con los conceptos relacionados (exclusivamente)

1º Suponiendo $i = 1, \ldots n$, tendremos

$$B + C_i(k_i) \longrightarrow r_i(k_i)$$

$$* \quad B + C_i(k_i) \longrightarrow r_i(k_i') \qquad k_i' \neq k_i$$

(al menos 3 valores distintos de k_i')

(podremos plantear n preguntas)

2º Suponiendo $i = 1, \ldots n$, tendremos

$$B + C_i(k_i) \longrightarrow r_i(k_i)$$

$$* \quad B + C_i(k_i) \longrightarrow r_i(k_i') \qquad k_i' \neq k_i$$

$$- \quad B + C_i(k_i) \longrightarrow r_{i'}(k_{i'}) \qquad i' \neq i$$

(al menos 2 valores distintos de k_i')

(podremos plantear n preguntas)

3º Suponiendo $i = 1, \ldots n$, tendremos

$B + C_i(k_i) \longrightarrow r_i(k_i)$

* $B + C_i(k_i') \longrightarrow r_i(k_i)$ $\quad k_i' \neq k_i$ (al menos 2 valores distintos de k_i')

– $B + C_{i'}(k_{i'}) \longrightarrow r_i(k_i)$ $\quad i' \neq i$ (1 solo valor de i' para cada uno de i)

Podremos plantear $\frac{n}{2}$ preguntas (si n es par), o

$\frac{n}{2} + 1$ (si n es impar)

B) Con los conceptos limitadores

1º Suponiendo $j = 1, \ldots m$, tendremos

$E \longrightarrow B$

* $E \longrightarrow L_j$ (al menos $m = 2$)

– $E \longrightarrow G$

2º Suponiendo $i = 1, \ldots n$, y $j = 1, \ldots m$, tendremos

$E \longrightarrow B$

$E \longrightarrow B + r_i(k_i)$

* $E \longrightarrow L_j$ (basta con $m = 1$ y $n = 1$)

$E \longrightarrow \neg L_j + r_i(k_i)$

Si queremos que todas las respuestas tengan idéntico número de componentes, tendremos que tener

o $m + 1 \geq 4$ (utiliz. un solo concepto relacionado)

o $(m + 1) \cdot n \geq 4$

4. Discusión

Tal como hemos formulado las preguntas y respuestas, en cada caso tendremos una respuesta sola contradictoria con lo supuesto, y las demás serán compatibles, en general, con ello; bastará negar constantemente uno de los miembros para tener lo contrario (o sea, una sola respuesta "acertada").

VSZ.

En la reunión del día 30, Sánchez de Zavala expuso lo siguiente:

Para poder presentar unos ejemplos de aplicación de la obtención formal de gamas de respuestas que esbocé el día anterior conviene primero modificar ligeramente la simbología para evitar ciertos defectos de claridad (¡de automatismo, por consiguiente!) que de otro modo se presentarían. Voy, pues, a indicar ahora las modificaciones con respecto al día anterior, manteniendo lo demás.

FORMULAS GENERALES (continuación)

1. Simbología

Emplearemos:

$r_i(k_{il})$ como símbolo de un rasgo caracterizador de un concepto relacionado con el concepto eje, siempre que queramos explicitar su carácter variable, dependiente de cierta magnitud característica;

$C_i(k_{il})$ como símbolo de dicha magnitud, siempre que queramos explicitar su variabilidad (de la que dependerá la de r_i),

y la tabla que sigue

	i		
	1	2	.....
$C_i(k_{il})$	$\langle C_i(k_{i1}) \rangle$	$\langle C_i(k_{i2}) \rangle$	.....
$r_i(k_{il})$	$\langle r_i(k_{i1}) \rangle$	$\langle r_i(k_{i2}) \rangle$	.....

(en la que los paréntesis angulares sirven para indicar que en el lugar que encuadren se encontrará el valor correspondiente que posea la magnitud cuyo símbolo se encuentre abarcado por ellos).

servirá para indicar la correspondencia de los valores de C_i y r_i

(Es claro que para cada valor de i el número de valores de l será, en general, distinto).

2. Gamas de respuestas

A) Con los conceptos relacionados (exclusivamente)

1º Suponiendo $i = 1, \ldots n$ y $l = 1, \ldots p$,

tendremos

$$\begin{array}{lll} & B + C_i(k_{il}) \longrightarrow r_i(k_{il}) & \\ * & B + C_i(k_{il}) \longrightarrow r_i(k_{il'}) & l' \neq l \quad (p \geq 4) \end{array}$$

(Podremos plantear n preguntas)

2º Suponiendo $i = 1, \ldots n$ y $l = 1, \ldots p$,

tendremos

$$\begin{array}{lll} & B + C_i(k_{il}) \longrightarrow r_i(k_{il}) & \\ * & B + C_i(k_{il}) \longrightarrow r_i(k_{il'}) & l' \neq l \\ - & B + C_i(k_{il}) \longrightarrow r_{i'}(k_{i\eta}) & i' \neq i \quad (p + n \geq 4) \end{array}$$

(Podremos plantear n preguntas; pero véase la Nota)

3º Suponiendo $i = 1, \ldots n$ y $l = 1, \ldots p$,

tendremos

$$\begin{array}{lll} & B + C_i(k_{il}) \longrightarrow r_i(k_{il}) & \\ * & B + C_i(k_{il'}) \longrightarrow r_i(k_{il}) & l' \neq l \\ - & B + C_{i'}(k_{i\eta}) \longrightarrow r_i(k_{il}) & i' \neq i \quad (p + n \geq 4) \end{array}$$

(Podremos plantear n preguntas; pero véase la Nota)

N o t a .- Si queremos que no haya ninguna respuesta repetida, y para cada pregunta utilizamos en su gama de respuestas h valores (distintos) de i'

una vez escogido el primer valor de i, quedarán "libres" para las próximas escogidas.

$$n - (1+h) \quad \text{valores de } i\ ;$$

una vez escogido el segundo valor de i, quedarán "libres", para las próximas escogidas,

$$n - 2(1+h) \quad \text{valores de } i;$$

.....

una vez escogido el t-ésimo valor de i, quedarán "libres", para las próximas escogidas,

$$n - t\ (1+h) \quad \text{valores de } i\ ;$$

pero para poder escoger este t-ésimo valor, habrá de cumplirse

$$n - t\ (1+h) \geq 0$$

de donde

$$t \leq \frac{n}{1+h} \quad ;$$

luego el número máximo de preguntas que podremos formular de modo que no haya ninguna respuesta repetida será

$$T = \doteq \frac{n}{1+h}$$

B) Con los conceptos limitadores

1º Suponiendo $i = 1, \ldots n$, $j = 1, \ldots m$ y $l = 1, \ldots p$,

tendremos

$$E \longrightarrow B$$

$$- \quad E \longrightarrow B + r_i(k_{il}) \qquad (pn + m \geq 3)$$

$$* \quad E \longrightarrow L_j$$

(obsérvese que en este caso no todas las respuestas de la gama harán intervenir el mismo número de conceptos: unas serán más "largas" que otras).

2º Suponiendo $i = 1, \ldots n$, $j = 1, \ldots m$ y $l = 1, \ldots p$,

tendremos

$$E \longrightarrow B$$
$$- \quad E \longrightarrow B + r_i (k_{ij}) \qquad (2pn + 1 \geq 3 \text{ , si } j \text{ toma un solo valor;}$$
$$* \quad E \longrightarrow L_j \qquad \text{en general}$$
$$- \cdot \quad E \longrightarrow \neg L_j + r_j (k_{jl}) \qquad (1+m)\ pn + m \geq 3)$$

(Obsérvese lo mismo que en el caso anterior)

3º Suponiendo $i = 1, \ldots n$, $j = 1, \ldots m$ y $l = 1, \ldots p$,

tendremos

$$E \longrightarrow B + r_i (k_{il}) \qquad ((1+m)\ pn \geq 4)$$
$$E \longrightarrow \neg L_j + r_j (k_{jl})$$

4. Discusión

Obsérvese que las condiciones (inecuaciones) que indicamos a la derecha de las gamas de respuestas se basan en la condición general de que el número de respuestas de cada gama sea

$$\geq 4.$$

Como es natural, pueden encontrarse otras gamas de respuesta, sin duda, pero las indicadas parecen ser bastante sencillas de obtener.

EJEMPLOS

Volvemos a tomar el ejemplo de la adaptación, que había presentado en el trabajo del 9 de enero, apartado "Ejemplo piloto de módulo de prueba conceptual" (parte inicial y epígrafe "A 1"). En este caso, la aplicación de las fórmulas generales obtenidas se hará en tres fases.

1. Interpretación de los símbolos

E = adaptación

B = aumento de la viabilidad (se entiende que en el ejemplo que se ponga en el examen se han de cumplir las circunstancias o situación, S, necesarias para que B sea un rasgo necesario para que un "objeto" cualquiera satisfaga E: véase el trabajo del 16 de abril, apartado "Fórmulas generales", epígrafe "1. Simbología", al comienzo)

L_1 = transformación del medio por la *actividad directa, instrumental* del ser vivo.

r_1 = plasticidad adaptativa

C_1 = tipo de constitución

tabla de correspondencia entre r_1 y C_1:

	1	
	1	2
$r_1(k_{1l})$	p. pequeña	p. grande
$C_1(k_{1l})$	c. animal	c. vegetal

r_2 = velocidad de adaptación

C_2 = nivel al que se produce la adaptación

tabla de correspondencia entre r_2 y C_2 :

	1		
	1	2	3
$r_2(k_{2l})$	v. pequeña	v. mediana	v. grande
$C_2(k_{2l})$	n. genético	n. fisiológico	n. comportamental

2. Obtención de esquemas posibles de gamas de respuestas

A) Con los conceptos relacionados

Como para $i = 1,\ p = 2$

$i = 2,\ p = 3$,

no podremos formar el esquema 1º

(continuará)

VSZ.

SEMINARIOS

Valoración del Aprendizaje

Participantes: S. del Campo Urbano, E. García Camarero, L. García-Ramos, M. Gil Gayarre, J.A. Martínez Carrillo, R. Moya Quiles, I. Ramos, V. Sánchez de Zavala.

Reuniones: 14 de mayo y 11 de junio.

Comunicaciones:

A continuación reproducimos la comunicación de Sánchez de Zavala en la reunión del día 11 de junio:

Antes de continuar en el punto a que habíamos llegado el día anterior vamos a introducir una leve modificación en la simbología, con objeto de simplificar las fórmulas y de tener en cuenta explícitamente (esto es, de modo que pueda efectuarse la automatización sin errores) la advertencia que hacíamos en la sección "FORMULAS GENERALES", apartado "1. Simbología", ad finem.

FORMULAS GENERALES (Nueva versión)

1. Simbología (alteraciones con respecto al día anterior)

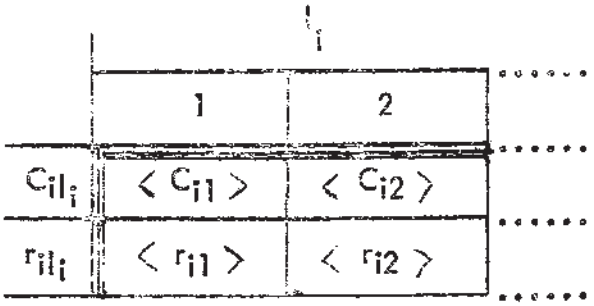
Vamos a sustituir

$$r_i\,(k_{il}) \text{ por } r_{il_i}$$

y

$$C_i\,(k_{il}) \text{ por } C_{il_i}\,.$$

La tabla será ahora la siguiente:



2. Gamas de respuestas

A) Exclusivamente con los conceptos relacionados

1º Siendo $i = 1, \ldots n$ y $l_i = 1, \ldots p_i$, tendremos

$$\left.\begin{array}{ll} B + C_{il_i} \longrightarrow r_{il_i} & \\ * \; B + C_{il_i} \longrightarrow r_{il_i'} & l_i' \neq l_i \end{array}\right\} \quad (p_i \geq 4)$$

(Podremos plantear n preguntas).

2º Siendo i y l_i como antes, tendremos

$$\left.\begin{array}{lll} B & C_{il_i} \longrightarrow r_{il_i} & \\ * \; B & C_{il_i} \longrightarrow r_{il_i'} & l_i' \neq l_i \\ - \; B & C_{il_i} \longrightarrow r_{i'l_{i'}} & i' \neq i \end{array}\right\} \quad (p_i + n \geq 4)$$

(Podremos plantear n preguntas, pero véase la Nota).

3º Siendo i y l_i como antes, tendremos

$$\left.\begin{array}{lll} B & C_{il_i} \longrightarrow r_{il_i} & \\ * \; B & C_{il_i'} \longrightarrow r_{il_i} & l_i' \neq l_i \\ - \; B & C_{i'l_i} \longrightarrow r_{il_i} & i' \neq i \end{array}\right\} \quad (p_i + n \geq 4)$$

(Podremos plantear n preguntas, pero véase la Nota).

Nota.

Como vimos el día anterior, si queremos que no haya ninguna respuesta repetida (en cada esquema), el número máximo de preguntas que podremos formular por esquema será

$$T = \frac{n}{1+h}$$

(siendo h el número de valores distintos de i´que utilicemos en la gama de respuestas de cada pregunta).

B) Con los conceptos limitadores

1º Siendo $j = 1, \ldots m$, tendremos

$$\left.\begin{array}{l} E \longrightarrow B \\ * \; E \longrightarrow L_j \\ E \longrightarrow G \end{array}\right\} \qquad (m \geq 2)$$

2º Siendo $i = 1, \ldots n$, $j = 1, \ldots m$ y $l_i = 1, \ldots p_i$, tendremos

$$\left.\begin{array}{l} E \longrightarrow B \\ E \longrightarrow B \; r_{il_i} \\ * \; E \longrightarrow L_i \end{array}\right\} \qquad (p_i n \;\; m \geq 3)$$

3º Siendo i, j y l_i como antes, tendremos

$$\left.\begin{array}{l} E \longrightarrow B \\ E \longrightarrow B \; r_{il_i} \\ * \; E \longrightarrow L_i \\ - \; E \longrightarrow \neg L_i + r_{il_i} \end{array}\right\} \qquad ((1+m)\, p_i n + m \geq 3)$$

4º Siendo i, j y l_i como antes, tendremos

$$\left.\begin{array}{l} E \longrightarrow B + r_{il_i} \\ E \longrightarrow \neg L_i + r_{il_i} \end{array}\right\} \quad ((1+m)\, p_{in} \geq 4)$$

3. Discusión

(Lo mismo que el día pasado).

EJEMPLOS (Nueva versión y continuación)

1. Interpretación de los símbolos

Tenemos que corregir el defecto de que, el día anterior, el concepto E, o sea, el de adaptación, entraba en los r_i. Por consiguiente, sin alterar lo demás, adoptaremos - ahora

r_1 = plasticidad (en las modificaciones morfológicas o de funcionamiento)

con lo que el nuevo cuadro correspondiente será:

	I	
	1	2
C_{1l_1}	const. animal	const. vegetal
r_{1l_1}	plast. pequeña	plast. grande

De modo análogo, adoptaremos ahora

r_2 = velocidad de modificación (morfológica o de funcionam.)

C_2 = nivel al que se produzca la modificación

y el nuevo cuadro de correspondencia será:

	I		
	1	2	3
C_{2I_2}	nivel genético	nivel fisiológico	nivel comport.
r_{2I_2}	veloc. pequeña	veloc. mediana	veloc. grande

2. Obtención de esquemas posibles de gamas de respuestas

A) Con los conceptos relacionados exclusivamente

1º No puede emplearse el esquema 1º, que exigía $p_i \geqslant 4$, ya que $p_1 = 2$ y $p_2 = 3$.

2º Como para todo valor de n se cumple la condición (inecuación) exigida, ya que $p_1 + n = 4$ y $p_2 + n = 5$, cabe plantear las n preguntas permitidas por el esquema 2º, que serán, esquemáticamente, a) y b).

a) $\begin{cases} B + C_{1I_1} \longrightarrow r_{1I_1} \\ * \;\; " \quad " \longrightarrow r_{1I_1'} \\ - \;\; " \quad " \longrightarrow r_{2I_2} \end{cases}$ (3 respuestas, por ser $p_2 = 3$)

En el caso de nuestro ejemplo, podremos elegir, pues, cualquiera de las siguientes gamas:

$a_1)$ $\begin{cases} \text{aum. de viab. + const. animal} \longrightarrow \text{plast. pequeña} \\ * \quad " \quad " \longrightarrow " \quad \text{grande} \\ - \quad " \quad " \longrightarrow \text{veloc. pequeña} \\ - \quad " \quad " \longrightarrow " \quad \text{mediana} \\ - \quad " \quad " \longrightarrow " \quad \text{grande} \end{cases}$

De las tres últimas respuestas podemos quedarnos sólo con 2, teniendo en cuenta "3. Discusión");

a_2)

	aum. de viab. + const. veget	⟶	plast.	grande
*	" "	⟶	"	pequeña
–	" "	⟶	veloc.	pequeña
–	" "	⟶	"	mediana
–	" "	⟶	"	grande

(de las 3 últimas respuestas podemos quedarnos sólo con 2, como antes).

b)

B	C_{2l_2}	⟶ r_{2l_2}
* "	"	⟶ $r_{2l'_2}$
– "	"	⟶ r_{1l_1}

(2 respuestas, por ser $p_1 = 2$)

En el caso de nuestro ejemplo, podremos elegir, pues, cualquiera de las siguientes gamas:

b_1)

	aum. de viab. + nivel genét.	⟶	veloc.	pequeña
*	" "	⟶	"	mediana
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

b'_1)

	aum. de viab. + nivel genét.	⟶	veloc.	pequeña
*	" "	⟶	"	grande
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

b_2)

	aum. de viab. + nivel fisiol.	⟶	veloc.	mediana
*	" "	⟶	"	pequeña
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

b'_2)

	aum. de viab. + nivel fisiol.	⟶	veloc.	mediana
*	" "	⟶	"	grande
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

a_2)

	aum. de viab. + const. veget	⟶	plast.	grande
*	" "	⟶	"	pequeña
–	" "	⟶	veloc.	pequeña
–	" "	⟶	"	mediana
–	" "	⟶	"	grande

(de las 3 últimas respuestas podemos quedarnos sólo con 2, como antes).

b)

	B C_{2l_2}	⟶	r_{2l_2}
*	" "	⟶	$r_{2l'_2}$
–	" "	⟶	r_{1l_1}

(2 respuestas, por ser $p_1 = 2$)

En el caso de nuestro ejemplo, podremos elegir, pues, cualquiera de las siguientes gamas:

b_1)

	aum. de viab. + nivel genét.	⟶	veloc.	pequeña
*	" "	⟶	"	mediana
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

b'_1)

	aum. de viab. + nivel genét.	⟶	veloc.	pequeña
*	" "	⟶	"	grande
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

b_2)

	aum. de viab. + nivel fisiol.	⟶	veloc.	mediana
*	" "	⟶	"	pequeña
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

b'_2)

	aum. de viab. + nivel fisiol.	⟶	veloc.	mediana
*	" "	⟶	"	grande
–	" "	⟶	plast.	pequeña
–	" "	⟶	"	grande

b_3) { aum. de viab. + nivel comp. → veloc. grande
* " " → " pequeña
– " " → plast. pequeña
* " " → " grande

b_3') { aum. de viab. + nivel comp. → veloc. grande
* " " → " mediana
– " " → plast. pequeña
* " " → " grande

(Obsérvese que, como preveía la Nota de la sección "FORMULAS GENERALES" apartado "2. Gamas de respuestas", párrafo "A)", en este esquema, tal y como lo hemos desarrollado - al máximo -, se nos repiten respuestas. También merece advertirse que el carácter "neutral" de algunos esquemas de respuestas quedan ahora resuelto " en las gamas b_3 y b_3' -, debido a que, de los distintos valores de C, algunos de ellos no son enteramente independientes entre sí: en nuestro caso, C_{12} y C_{23} son incompatibles, cosa que da lugar a la simultánea incompatibilidad de r_{12} y r_{23}.)

3º También ahora se cumplen las condiciones (inecuaciones) para plantear las n preguntas permitidas por el esquema 3º, que serán, esquemáticamente, a) y b).

a) { $B + C_{1l_1} \longrightarrow r_{1l_1}$
* $B + C_{1l_1'} \longrightarrow$ "
– $B + C_{2l_2} \longrightarrow$ " (3 respuestas, por ser $p_2 = 3$)

En el caso de nuestro ejemplo, podremos elegir, pues, cualquiera de las siguientes gamas:

a_1) { aum. de viab. + const. animal → plast. pequeña
* " + const. veget. → "
– " + nivel genético → "
– " + nivel fisiol. → "
– " + nivel compart. → "

a_2'

	aum. de viab.	+ const. veget.	⟶	plast. grande
*	"	+ " animal	⟶	"
-	"	+ nivel genético	⟶	"
-	"	+ " fisiol.	⟶	"
-	"	+ " comport.	⟶	"

(como la hemos indicado en el apartado 2º, de las tres últimas respuestas de cada una de estas gamas podemos quedarnos sólo con dos).

b) $B + C_{2l_2} \longrightarrow r_{2!_2}$

* $B + C_{2l_2'} \longrightarrow r_{2l_2}$

\- $B + C_{1l_1} \longrightarrow r_{2l_2}$ (2 respuestas, por ser $p_1 = 2$)

(En el caso de nuestro ejemplo, podremos elegir, pues, cualquiera de las siguientes gamas:

b_1)

	aum. de viab.	+ nivel genét.	⟶	veloc. pequeña
*	"	+ " fisiol.	⟶	"
-	"	+ const. anim.	⟶	"
-	"	+ " veget.	⟶	"

b_1')

	aum. de viab.	+ nivel genét.	⟶	veloc. pequeña
*	"	+ " comp.	⟶	"
-	"	+ const. animal	⟶	"
-	"	+ " veget.	⟶	"

b_2)

	aum. de viab.	+ nivel fisiol.	⟶	veloc. mediana
*	"	+ " genet.	⟶	"
-	"	+ const. animal	⟶	"
-	"	+ " veget.	⟶	"

b_2'

	aum. de viab.	+ nivel fisiol.	⟶	veloc. mediana
*	"	+ " comp.	⟶	"
-	"	+ const. anim.	⟶	"
-	"	+ " veget.	⟶	"

$b_3)$ $\left\{\begin{array}{llll} & \text{aum. de viab.} & \text{nivel comport.} \longrightarrow & \text{veloc. grande} \\ * & " & " \text{ genét.} \longrightarrow & " \\ - & " & \text{const. animal} \longrightarrow & " \\ * & " & " \text{ veget.} \longrightarrow & " \end{array}\right.$

$\left\{\begin{array}{llll} & \text{aum. de viab.} & \text{nivel comport.} \longrightarrow & \text{veloc. grande} \\ * & " & " \text{ fisiol.} \longrightarrow & " \\ - & " & \text{const. animal} \longrightarrow & " \\ * & " & " \text{ veget.} \longrightarrow & " \end{array}\right.$

B) Con los conceptos limitadores

1º Como m = 1, no se puede utilizar el esquema 1º

2º Como para todo valor de i se cumple la condición exigida por el esquema 2º, ya que

$p_1 n \quad m = 2.2 \quad 1 = 5$, y

$p_2 n \quad m = 3.2 \quad 1 = 7$,

se puede utilizar este esquema, que dará las respuestas esquematizadas a continuación:

$E \longrightarrow B$

$" \longrightarrow B + r_{11_1}$

$" \longrightarrow B + r_{11'_1}$ $\quad (p_1 - 1$ respuestas)

$" \longrightarrow B + r_{21_2}$

$" \longrightarrow B + r_{21'_2}$ $\quad (p_2 - 1$ respuestas)

$* \quad " \longrightarrow L_1$

En el caso de nuestro ejemplo, podemos elegir la gama que queramos de 4 o más respuestas de entre la siguiente serie de ellas:

adaptación ⟶ aum. de viab.
" ⟶ " plast. pequeña
" ⟶ " " grande
" ⟶ " veloc. pequeña
" ⟶ " " mediana
" ⟶ " " grande
* " ⟶ transform. instrum. del medio

3º Con mayor razón que antes podremos utilizar el esquema 3º, que dará las respuestas esquematizadas a continuación:

(todas las del esquema 2º), y además,

- $E \longrightarrow \neg L_1 + r_{1l_1}$
- " $\longrightarrow \neg L_1 + r_{1l_1'}$ ($p_1 - 1$ respuestas)
- " $\longrightarrow \neg L_1 + r_{2l_2}$
- " $\longrightarrow \neg L_1 + r_{2l_2'}$ ($p_2 - 1$ respuestas)

En el caso de nuestro ejemplo, podremos elegir, por lo tanto, la gama de 4 o más que queramos de entre las respuestas enumeradas o bajo el esquema - 2º más las que siguen:

- adapt. ⟶ no hay transf. instr. + plast. pequeña
- " ⟶ " + " grande
- " ⟶ " + veloc. pequeña
- " ⟶ " + " mediana
- " ⟶ " + " grande

4º También podremos utilizar el esquema 4º, ya que

$$(1+m)\, p_1 n = \text{ó} \text{ , y}$$
$$(1+m)\, p_2 n = 12 \text{ ,}$$

luego se cumple la condición exigida; y las respuestas esquemáticas serán las siguientes:

$E \longrightarrow B + r_{1}l_{1}$

" $\longrightarrow B + r_{1}l_{1}'$ ($p_1 - 1$ respuestas)

" $\longrightarrow B + r_{2}l_{2}$

" $\longrightarrow B + r_{2}l_{2}'$ ($p_2 - 1$ respuestas)

" $\longrightarrow L_1 + r_{1}l_{1}$

" $\longrightarrow L_1 + r_{1}l_{1}'$ ($p_1 - 1$ respuestas)

" $\longrightarrow L_1 + r_{2}l_{2}$

" $\longrightarrow L_1 + r_{2}l_{2}'$ ($p_2 - 1$ respuestas)

En el caso de nuestro ejemplo, podremos elegir, por consiguiente, la gama de 4 o más que queramos de la lista de respuestas correspondientes a estos esquemas, que no escribimos porque son las mismas que las del esquema 3º, excepto la primera y la última correspondientes al esquema 2º que ahora faltan.

(Obsérvese que en el caso de los conceptos limitadores no resulta este último caso o esquema "falsa" ninguna de las respuestas conseguidas. Esto hace que sea preciso introducir alguna incompatibilidad desde el exterior si es que se quiere dividir las respuestas en aceptables e inaceptables).

3. Construcción de preguntas concretas y sus gamas de respuestas

Vamos a elegir al azar una gama de cada uno de los tipos principales y a construir algún ejemplo concreto de pregunta y gama de respuesta para una prueba automatizada.

Ejemplo de tipo A-2º-a_1):

"En el desierto de Mohave, en California, hay un perrito de los desiertos que consigue sobrevivir en la época tórrida, cuando la mayoría de los animales perecen o emigran; pero no sabemos bien como lo consigue, pues presenta grandes dificultades observarlo en esa estación. Qué hipótesis le parece más apropiada para dirigir las subsecuentes

investigaciones al respecto, de las que enumeramos más abajo? (Elija la que le parezca preferible, aunque alguna o algunas de las otras pudieran también no ser erróneas, a su juicio).

1. Se alteran lo suficiente los hábitos y la fisiología del animal para permitirlo soportar el exceso de calor y la sequedad.

2. Se produce una completa reorganización del animal, mediante la cual soporta cómodamente tales condiciones ambientales.

3. El animal reacciona directamente a la temperatura y sequedad con las actividades que le permiten combatirlos.

4. A lo largo de generaciones y generaciones de soportar difícilmente esas condiciones ambientales, este animal ha terminado por tolerarlas bien.

(Es claro como podría modificarse cualquiera de las dos últimas respuestas para hacer que se volviera "errónea", aunque sin ser absurda).

Ejemplo de tipo A-3º-b3):

"Al transportar una colección de animales y plantas tropicales a un parque zoológico se encierra, por error, en la misma caja (herméticamente cerrada) un animal muy activo y una planta de grandes hojas y flores, tales que cada uno de ellos, por sí solo, consume normalmente la casi totalidad del oxígeno del aire de la caja durante las pocas horas de duración del viaje. Sin embargo, al bajar del avión la colección y abrir la caja, tanto el animal como la planta se encuentran en perfecto estado. Cuál de las cuatro explicaciones siguientes le parece más acertada?.

1. Se ha modificado el ritmo respiratorio de los dos seres vivos, o de uno solo de ellos, de modo que sobrevivan a la disminución de oxígeno y el aumento de anhidrido carbónico.

2. Se ha alterado la fisiología respiratoria de ambos, o de uno solo de ellos, de modo que, de ahora en adelante, podrán vivir en circunstancias de análoga escasez de oxígeno (ventaja que transmitirán a sus descendientes).

3. Es sólo el animal el que se ha adaptado a la escasez de oxígeno, gracias a lo cual ha podido sobrevivir (y, de paso, también la planta).

4. Es sólo la planta la adaptada momentáneamente a la escasez de oxígeno, cosa que le ha permitido sobrevivir (y, de paso, también al animal).

Ejemplo de tipo B-3º:

Hay bastantes casos conocidos de adaptación de una especie, o unos pocos individuos de ella, a un medio hostil; por ejemplo, es relativamente frecuente la adaptación de pájaros continentales, arrastrados por una tempestad, a las condiciones de vida de una isla oceánica. En uno de estos casos conocidos, en el que la isla de llegada es mucho más fría que el lugar de procedencia, y en el que, por lo tanto, muchos de los individuos del grupo de pájaros allí arribado debieron de morir de frío poco después de la llegada, de qué manera cree Vd. preferible exponer lo sucedido en la adaptación? (elija la que le parezca mejor de las indicadas a continuación):

1. Esas muertes se compensaron por una mayor descendencia de los supervivientes, o por una descendencia más resistente al frío.

2. Lo mismo que en la respuesta anterior, pero puntualizando, además, que también la primera generación nacida en la isla tuvo que tener una elevada tasa de mortalidad y que sólo poco a poco pudo irse consiguiendo contrarrestar el efecto de las bajas temperaturas.

3. Los supervivientes tuvieron que deber su resistencia a conservar de algún modo el medio ambiente del país de origen (por ejemplo, se apiñarían, dándose calor unos a otros, y de este modo se conservarían unos a otros en un "medio artificial" muy parecido al que estaban acostumbrados).

4. No, los pájaros llegados a la isla no pudieron adaptarse creando un "ambiente artificial" (aunque no es imposible que, además, lo crearan), sino que la especie experimentó poco a poco una modificación, lo suficientemente pequeña como para que pudiera lograrla por sí sola, y a la vez, capaz de hacerles soportar el frío.

(También aquí sería fácil modificar ligeramente algunas de estas respuestas para volverlas "erróneas" sin ser absurdas).

4. Valoración de la construcción

Como acabamos de ver, a partir de las fórmulas generales, y apoyándonos en lo que hemos llamado interpretación de los símbolos (que no es otra cosa que la aplicación de las fórmulas a los conceptos cuyo conocimiento por parte de los alumnos se trate de poner a prueba), se forma sin dificultad, por medio de una combinatoria elemental, un número verdaderamente elevado de gamas esquemáticas de respuestas; y a partir de éstas cabe construir "hacia atrás" ejemplos variados que pongan de manifiesto tanto la manera de enfocar el tema los alumnos como su mayor o menor conocimiento de él. En cierto sentido, podemos decir que el problema que nos habíamos planteado al final del trabajo presentado el 19 de febrero, y comenzado a atacar en la comunicación siguiente, el 16 de abril, ha quedado resuelto y dispuesto para una aplicación sencilla de computadora en cuanto se cuente con una especificación exacta de las nociones de partida en que nos apoyamos (concepto eje, rasgos básicos, conceptos limitadores, etc.).

Queda, pues, la tarea de obtener todas estas nociones igualmente de un modo más o menos automático a base de los textos (explicaciones, obras, artículos) que se consideren fundamentales. Y debe observarse asimismo que esta tarea, en definitiva, tendrá que preceder también a la construcción de módulos de prueba no conceptuales, esto es, en los que no se trate de explorar la aprehensión y comprensión de unos conceptos por sí solos, sino la capacidad de resolver problemas: véase la comunicación del 28 de noviembre, "Sobre el apartado III", epígrafe "1º. Qué queremos medir", a 2) y b 1), y epígrafe "2º. Los instrumentos de medida", B.

PERSPECTIVAS DE ESTUDIO

Teniendo en cuenta lo que acabamos de decir, nos parece que el camino más fructífero sería el de dedicarnos momentáneamente al estudio de la automatización del análisis de contenido y del análisis conceptual, temas sobre los cuales hay cierta bibliografía y que, si no resolvemos previamente, nos impedirán conseguir lo que pretendemos en este seminario.

Seminario de Generación de Formas Plásticas.

Participantes: Málaga: M. Barbadillo

Madrid: F. Alvarez Cienfuegos, M. de las Casas Gómez, M. Fernández Barberá, I. Fernández Florez, E. García Camarero, A. García Quijada, A. Martín, J. Montero, I. Ramos, G. Searle, J. Seguí, R. Sempere, S. Sevilla Portillo

Valencia: V. Aguilera Corni, J. Mª López Yturralde

Reuniones: Primera reunión 18 de diciembre a las 13 horas.
Próximas reuniones: a partir del día 16 de enero a las 9 de la mañana con un período quincenal.

Comunicaciones:

En la reunión García Camarero, tras exponer la marcha que se estaba llevando a cabo en los otros seminarios y cuales eran los objetivos generales que se pretenden alcanzar con los mismos, indicó que la idea para la creación de este seminario surgió como consecuencia de la memoria presentada por el pintor Manuel Barbadillo, solicitando una de las becas para trabajos monográficos convocadas por el CCUM en abril de 1968. En dicha memoria manifestaba su creencia de que el ordenador podría colaborar en la solución de alguno de los problemas con que se enfrenta la investigación plástica actual; conceptos que desarrolló en el coloquio de clausura de uno de los cursos de programación desarrollados en el CCUM. Esta creencia la comparten Mario Barberá y los arquitectos Seguí de la Riba, de las Casas Gómez, de la Prada Poole, Searle, que en la actualidad participan en el seminario de organización de espacios arquitectónicos. Así mismo el crítico de arte Aguilera Cerni, expuso en líneas generales el interés del grupo de Valencia en la investigación de la génesis plástica y su vinculación con la percepción y la psicología, indicando la obra de Yturralde como un intento en este sentido. García Camarero estima posible la generalización de los modelos de la gramática generativa para la descripción de la estructura plástica de un cuadro, y especialmente aplicables al ejemplo de la estética de Barbadillo. Se propuso invitar a Tomás Marco por considerarse interesantes sus estudios sobre semántica musical. También se consideró que sería útil tomar la obra de Mondrian, Kantisky, Albers, Malevich como punto inicial de los estudios y ponerse en contacto con los Profs. Cecrati y Borsari, quienes en Milan ya han realizado diversas experiencias de la aplicación de ordenadores en la composición plástica y en la percepción.

Ordenación de la Construcción

Participantes: J.J. Aracil Bellod, A. Arranz, A. García Arangoa, E. García Camarero, T. González de Vega, J.M. Monteagudo Muñoz, G. Ramírez Gallardo.

Reuniones: Primera reunión 27 de noviembre
Segunda reunión 11 de diciembre
Próximas reuniones: a partir del miércoles 15 de enero a las 6,30 de la tarde con un período quincenal.

Comunicaciones:

Para replantear las líneas generales tratadas en la primera sesión, cada uno de los representantes expuso su interés particular sobre el tema. Por sugerencia de García - Camarero se convino que sería útil añadir un camino, a los ya indicados por V. Sánchez de Zavala y estudiar los aspectos semánticos de los lenguajes artificiales del tipo Algol; un lingüista profesional podría ver en ellos posiblemente con más claridad (dada la simplicidad de su estructura) muchas cuestiones que iluminarán el estudio de la semántica de los lenguajes naturales.

Otro aspecto sugerido en esta reunión fué el plantear un lenguaje de orientación específica para el tratamiento de los problemas surgidos en el proyecto arquitectónico.

Bibliografía:

Proceedings of the International Conference on Machine Translation of Languajes and Appl. Lg. Analysis. Teddington. 1961.
Contiene esta obra: Matthews "Analysis by Synthesis of Sentences of Natural language"

BOOTH, A.D. - Progress in Machine Translation,- 1967. Amsterdam.

FEIGENBAUM, E.A., Feldman, 5 - Computers and thought .- 1963. New York . Mc Graw Hill

FROGER, D.T. - La critique des textes et son automatisation.- 1968. Paris. Dunod.
Les textes et leur histoire.
Les méthodes de critique textuelle
Théorie de la critique textuelle
La théorie des ensembles et la critique textuelle
L'automatisation dans la critique des textes

STONE, Ph. J. et alt.- The general inquirer: A computer approach to content Analysis. 1967. Cambridge. The M.I.T. Press.

MACHINE INTELLIGENCE.- Papers read at Annual Workshop held in Edinburgh

DATA PROCESSING FOR DECISION-MAKING. McMillan. 1968. Amsterdam.

Generacion de formas plásticas

Participantes:
Málaga: M. Barbadillo
Madrid: Alexanco, F. Alvarez Cienfuegos, F. Briones, M. de las Casas Gómez, E. Delgado, M. Fernández Barberá, I. Fernández Flórez, E. García Camarero, A. García Quijada, A. Martín, J. Montero, J. Peña, I. Ramos, G. Searle, J. Seguí, R. Sempere, S. Sevilla Portillo
Valencia: V. Aguilera Cerní, J. Ma. L. Yturralde.

Reuniones: Se celebraron los días 16 y 30 a las 9 de la mañana

Comunicaciones:

Aguilera Cerní nos indica desde Valencia las siguientes notas bibliográficas:

"Hay multitud de títulos referentes a psicología de la forma, psicología experimental y estadística, etc., pero os supongo mejor informados que yo.

En realidad, aquí menciono solamente ciertos títulos y pistas sin pretensión alguna de completar nada, y sólo porque a veces pueden pasarnos desapercibidos."

Algunos de los trabajos mencionados llevan bibliografías detalladas.

I.C.A. BULLETIN
Institute of Contemporary Arts
17-18 Dover Street
London W1
Grosvenor 6186-6119
Nº 162, October 1966
Hans Jürger Eysenck: Aesthetics and Personalyti
(Cita Bibliografía)

CONVEGNO INTERNAZIONALE DEGLI ARTISTI, CRITICE E STUDIOSI D'ARTE
Secretario permanente, Gerarso F. Dasi
Verucchio (Forli) Italia
Actas del XII Convegno, pág. 81.
Borsari: La simetría: orientamenti per un'estetica sperimentale.

H. WEYL
La simetría
E. Nueva Visión (Buenos Aires)

PIERRE FRANCASTEL
Aspects sociaux de la symétrie
Revista Les Temps modernes, nº 202. Paris, marzo 1963.

GILLO DORFLES
Símbolo, comunicación, consumo
Ed. Ariel
Artificio e Natura. Ed. Einaudi

R.L. GREGORY
Ojo y cerebro
Ed. Guadarrama.

SUSANNE, K. LANGER
Sentimento e forma
Ed. Feltrinelli
Título original: Freeling and Form. Ed. Charles Scribner's Sons

ADAM SCHAFF
Introducción a la semántica
Ed. Fondo de Cultura Económica

Pensamientos y lenguaje

MARIO BUNGE
Antología Semántica
Ed. Nueva Visión

SVEN HESSELGREN
Los medios de expresión de la arquitectura
(aplicando la psicología experimental y la semántica)
Ed. Universitaria. Buenos Aires.

REVISTA "D'ARS"
Director, Prof. Oscar Signorini
Via S. Agnese, 3
Milán
Nº 41-42 (1963)
Lyns: Analisi superficiali e analisi profonde
Nº 5, año VI
Varios: Aspetti semantici nell'arte
Elorer: Due linguaggi
Nº 4, año VI
Borsari: L'attività grafica dei calcolatori elettronici.
(en la colección de esta revista, aparecen constantemente notas y referencias sobre cibernética y arte, muchas de ellas firmadas por LYNZ, (seudónimo)).

REVISTA "LA BIENNALE DI VENEZIA"
Ca'Giustinian
Venecia
Nº 50-51
Vellinghausen: Parola e immagine
Nº 61
Ceccato: Proposte per una indagine estetica da parte di un cibernetico.
(cita bibliografía)

REVISTA "SUMA Y SIGUE"
Sr. D. José Huguet Chanzá
Joaquín Costa, 59
Valencia
Nº 7-8
Ceccato: Estética y cibernética

REVISTA "L'EUROPA LETTERARIA"
Antiguo director: Giancarlo Vigorelli
Via P.A. Micheli, 78. Roma
Crítico de arte: Lorenza Truchi
Vía Ruggero Fauro, 66. Roma

Nº 28
Zveternick: Cibernética e letteratura
(resumen de todo lo hecho en este campo en la URSS)

UMBERTO ECO
Obra abierta
Ed. Seix Barral
La Struttura assente
Ed. Bompiani

JACQUES BERTIN
Sémiologie graphique
Ed. Gauthier-Villars
(Depósito en Madrid, distribuidora Científico-Técnica, Sancho Dávila, 27)

Personas y Centros:
Prof. Silvio Ceccato (Director del Centro de Cibernética y estudios lingüísticos de la Universidad de Milán)
C. so Porta Vittoria, 32
20122 Milan

Para el Centro FAS de Bolonia (estudios de estética experimental):
Dr. Carmelo Genovese
Via Oberdan, 15

Prof. R. Borsari
C. Pepoli, 14
Bolonia

En la Biblioteca del Colegio de Arquitectos de Madrid, consúltese la colección de las revistas OUTPUT y ULM, publicadas por la Hochschule für Gestaltung de Ulm. V. A.C.
V.A.C.

En la reunión del día 16 se analizaron algunas composiciones plásticas realizadas con plotter. Estas composiciones eran de dos categorías, unas ejecutadas con expresión gráfica de una fórmula de ecuación conocida y las otras al azar, limitando solamente la orientación del desplazamiento del lápiz del plotter. Tres grupos de interés quedaron marcados: uno en la generación plástica, utilizando un desarrollo modular, otro mediante la utilización de ecuaciones matemáticas y un tercero basándose en estudios sico-

lógicos de la percepción, buscando tensiones y distensiones en el movimiento de un punto o de una figura sobre la superficie de un cuadro. La integración de los tres aspectos fecundaran el ulterior trabajo. Se habló de la necesidad de un curso breve sobre ordenadores en la pintura y en la realización en fecha próxima de una exposición de obras compuestas con la ayuda de un ordenador.

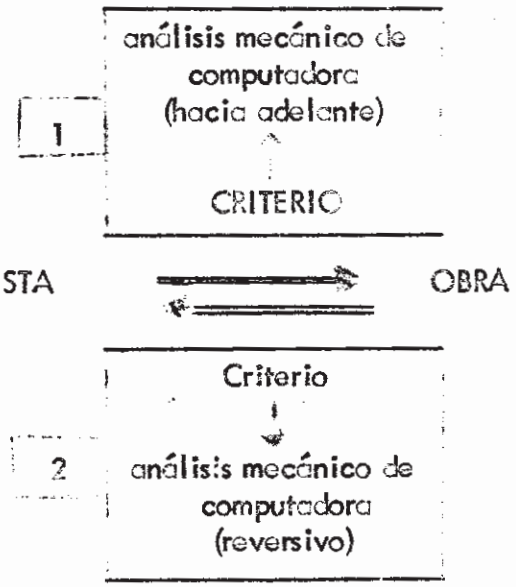
En la sesión del día 30 se expusieron algunos módulos realizados por Soledad Sevilla y consideraron que para la composición con ellos era de general interés tomar en consideración los colores utilizados en los mismos. A este respecto la idea de Barbadillo es utilizar como elemento de composición únicamente la forma en la búsqueda de un lenguaje plástico de fácil entendimiento. De la Prada expuso las siguientes normas generales para la investigación plástica:

Proposición para la obtención de un criterio de selección en la obra pictórica combinatoria.

A) Asignación de papeles en la dialéctica artista-obra

1A - ARTISTA — juez modificador y director del criterio de selección

2A - OBRA — objeto del análisis crítico, cuántico-cualitativo, que a su vez puede transformar éste para dar origen a un nuevo ciclo



El paso 1 es puntual en el tiempo

El paso 2 es lineal en el tiempo, pone al descubierto la ley de criterios y puede extrapolar a una ley de leyes o a un plano de una nueva dimensión.

B) Desarrollo Punto 1

1B - Delimitación del espacio pictórico objeto del análisis para establecer numéricamente sus dimensiones. Caben tres sistemas:

1B1 - Sistema cerrado (delimitado por el marco)

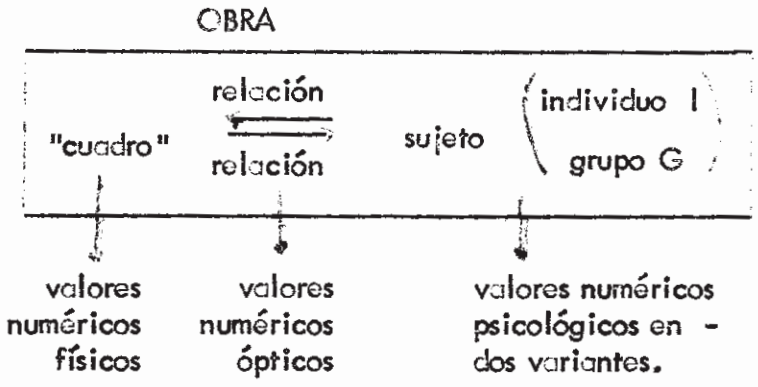
1B2 - Sistema semiabierto (se incorporan los contornos próximos, muros, paisajes, o partes de sí mismo disgregadas o montables que forman do parte del conjunto pueden adoptar diversas posturas).

1B3 - Sistema abierto (hacia la ilimitación)

2B - Hipótesis de Partida - Se denomina OBRA y por lo tanto objeto de análisis crítico en PUNTO 1, al conjunto del espacio pictórico + sujeto observante (individuo o grupo).

2B1 - Hipótesis - La obra es un sistema cerrado en equilibrio.

2B2 - El EQUILIBRIO final es el resultado de la suma de valores físicos, ópticos y psicológicos



3B - Hipótesis deducidas

3B1 ⟶ H. FISICA ⟶ OBTENCION del e.d.g. del sistema virtual de masas del "cuadro", en el que los colores constituyen una densidad teórica (Se consideran las líneas, formas y colores independientes de su significado psicológico y cultural)

382 — H. OPTICA - La fisiología de la visión modifica las dimensiones y masas obtenidas anteriormente. Aparición de los coeficientes de variación OP.

Coeficiente OPU - Los umbrales modifican las dimensiones. El - umbral mínimo haciéndolas desaparecer, el umbral del máximo campo de visión reduciendo el total a éste tamaño. Ambos umbrales dependen de la distancia normal de observación. Los umbrales sucesivos de posición y de dimensión nos valorarán la ley de proporciones armónicas.

Coeficiente OPDI - Las difracciones modifican numéricamente las dimensiones físicas haciéndolas depender de una escala de tonos, - una escala de brillos y una escala de colores que se pueden obtener fácilmente por establecimiento estadístico de umbrales sucesivos.

383 — H. SICOLOGICA. La sicología de la percepción modifica también las masas físicas mediante la introducción de los coeficientes mesurables SI

Coeficiente SIMO - Los MOVIMIENTOS desplazan la masa en su - línea de acción, en una dimensión que depende de la actividad del movimiento según una escala, que no hace falta que sea real, basada en dos movimientos extremos y dos intermedios.

Coeficiente SIAC - Los colores modifican su tamaño según una escala muy simple de actividades en la que el color más activo se encuentra situado una medida determinada (no importa que no sea -- real) más cerca que el más pasivo. El resto se interpolan.

Coeficiente SICU - El contenido cultural del individuo o grupo observante modifica el valor total restándole o añadiéndole dimensiones, con lo cual puede llegar a perder su significado.

Coeficiente SIN - Coeficiente de incomodidad psicológica valorado según una escala aleatoria.

4B - Resultado a comprobar - Sobre una línea pictórica determinada y para - un observador fijo el e.d.g. de los "cuadros" de una etapa de un artista debe de permanecer dentro de un recinto lo suficientemente pequeño como para que se le pueda considerar invariable, y por lo tanto la OBRA - ESTARA EN EQUILIBRIO.

de la Prada Poole.

Por último, E. Delgado propuso las siguientes curvas para estudiar su comportamiento plástico

Ovalos de Cassini:

Lugar de los puntos tales que el producto de distancias a dos puntos fijos es constante.

Ecuación $(a^2+x^2+y^2) - 4a^2x^2 = K^2$
para valores de K = 1,2,3,4,5,6,7,8,9,10.
para valores de a = 1,2,3,4,5,6,7,8,9,10

para $K^2 = a^2$ la ecuación se transforma en $(x^2+y^2)^2 = 2a^2(x^2-y^2)$ que es la lemniscata de Bernouilli

Es decir para K = 1 .... 10 y a = 1 hay un haz
para K = 1 .... 10 y a = 2 otro haz y así sucesivamente

Curvas Rosáceas

En polares $\rho = \rho$ sen
para valores de m = 0,1,2,3,4,5, ..., n
y valores de ω desde 0º a 360º

Folium de Descartes

$$x^3+y^3 = 3mxy$$

o bien en coordenadas paramétricas

$$x = \frac{3mt}{1+t^3}$$

y valores de m = 0,1,2,3,4, ..., n

$$y = \frac{3mt^2}{1+t^3}$$

Curvas de Lamé

Definidas en coordenadas paramétricas

$$x = \cos^m t$$
$$y = \text{sen}^m t$$

para valores de m = 0,1,2,3, ...
y valores de t desde 0º a 360º

Trifolium

En coordenadas polares

$$\rho = 4R \cos\omega \cdot \cos(2\omega + \alpha)$$

o bien en coordenadas cartesianas

$$(x^2 + y^2)^2 = 4Rx(x^2 - y^2)\cos\alpha - 8Rx^2 y \operatorname{sen}\alpha$$

para valores de 2ω = desde 0º a 360º
y para valores de α que varían en 1 décima de ángulo
para $\alpha = 0$ se obtiene el trifolium recto.

Folium simple

$$y^3 = (x^2 + y^2)^2$$

o bien en coordenadas polares $\rho = \operatorname{sen}^3(\omega + \lambda)$
donde para valores de λ = desde cero y que varían en 1 décima del ángulo ω
ω varía desde 0º a 360º

Estudiar el Haz

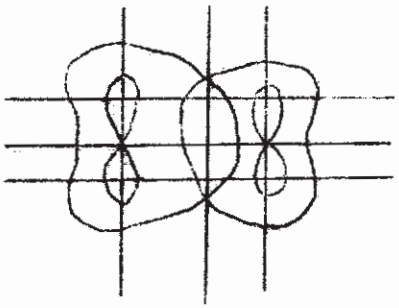
$$(x^2 - 4)^2 + (y^2 - 1)^2 = a^4$$

cuando a varía desde 0 a ∞

$$a = 0,1,2,3,4,5,6, \ldots$$

la representación corresponde a valores de

$$a = 0,1,4$$

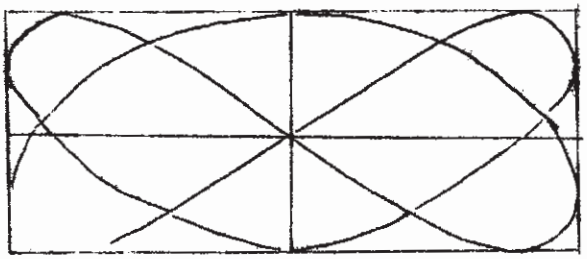


Estudiar el Haz

$$x = 3 \operatorname{sen} m\alpha$$
$$y = 2 \cos \frac{3}{2} m\alpha$$

para valores de m que difieran en una décima del ángulo α

en el caso particular de $m = \frac{1}{}$ la curva representada es



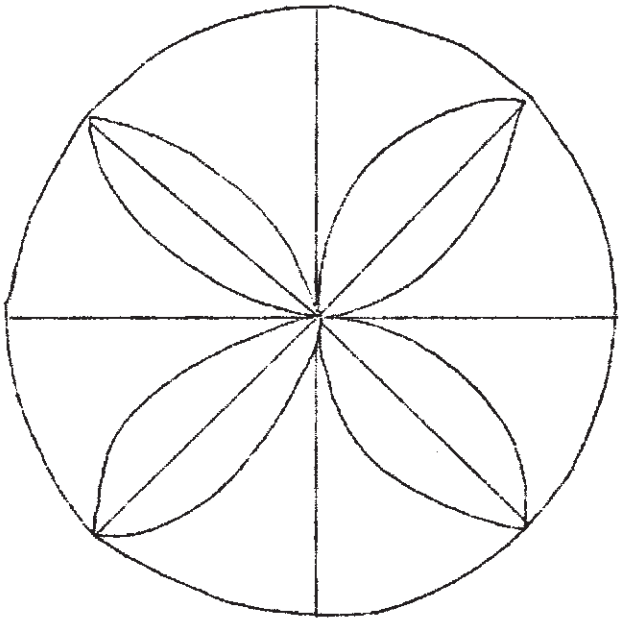
se ve que al variar m nos daría curvas análogas, pero desplazadas paralelamente o deformadas.

Estudiar

$\phi = \operatorname{sen}\theta \ (\cos\theta + \lambda)$

al variar λ

para el caso de $\lambda = 0$, esta curva es



θ varía	ρ varía
0	0
$\frac{\pi}{3}$	$\frac{3}{4}$
$\frac{\pi}{4}$	$\frac{1}{2}$
$\frac{\pi}{2}$	$\frac{3}{4}$
$\frac{\pi}{2}$	0
$\frac{3\pi}{2}$	$-\frac{1}{2}$
	0
$\frac{5\pi}{4}$	$\frac{1}{2}$
$\frac{3\pi}{2}$	0
$\frac{7\pi}{2}$	$-\frac{1}{2}$
2π	0

Estudiar

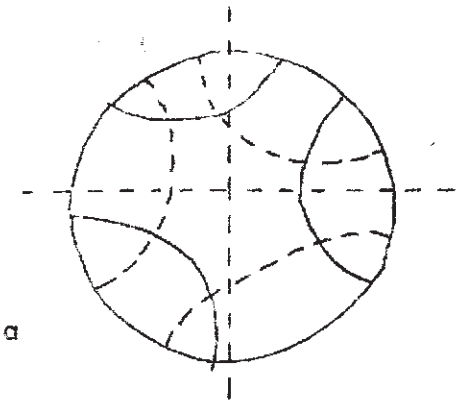
$$\rho = \alpha \sec. \frac{(3\theta + \lambda)}{2}$$

La curva completa se obtiene dando valores a θ desde 0 a 4π, los puntos más importantes para $\lambda = 0$ son

$\frac{3}{2}\ \theta = 0\ ,\ \frac{\pi}{2}\ ,\ \pi\ ,\ \frac{3}{2}\ \pi\ ,\ \ldots\ 6\pi$

$\theta = 0\ ,\ \frac{\pi}{3}\ ,\ \frac{2\pi}{3}\ ,\ \pi\ ,\ \ldots\ 4\pi$

$\rho = a\ ,\ +\infty\ ,\ -\infty\ ,\ -a\ ,\ -\infty\ ,\ +\infty\ ,\ \ldots\ a$



Estudiar

$$x^{2/3}\ \ y^{2/3} = a^{2/3}$$

Ecuación de la astroide para valores de $a = 0,1,2,3,4,5, \ldots, 10$.

Ordenación de la Construcción

Participantes: J.J. Aracil Bellod, A. Arranz, A. García Arangoa, E. García Camarero, T. González de Vega, J.M. Monteagudo Muñoz, - G. Ramírez Gallardo, J. Ma. Jenaro Garrido, J.P. Rodríguez Argenta.

Reuniones: El 15 y el 29 de enero.

Comunicaciones:

En la reunión del día 15, Ramirez Gallardo expuso las ideas básicas del sistema de clasificación SfB, utilizado para la identificación de materiales de construcción, cuyo resumen publicamos a continuación.

1 - Generalidades

El sistema SfB fué creado en Suecia en 1.950. Sus siglas se derivan del nombre en sueco del comité que lo creó: Samarbetskommitten for Byggnadsfragor - Comité Coordenador de la Construcción.

El C.I.B. (International Council for Building, Research, Studies and - - Documentation - Consejo Internacional de Investigaciones, Ensayos y Documentación) creó en 1.952 el Comité I.B.C.C. (International Building Classification Commitee - Comité Internacional de Clasificación) encargado de estudiar los problemas de clasifi-

F.P. Strawson (ed.), Philosophical Logic, Oxford, O. Univ. Press ("Oxford Readings in Philosophy"), 1967.

G.H.R. Parkinson (ed.), The Theory of Meaning, Oxford, O. Univ. Press ("Oxf. Read. in Phil."), 1968.

(En castellano:)

M. Bunge (comp.), Antología semántica, Buenos Aires, Nueva Visión, 1960.

T.M. Simpson, Formas lógicas, realidad y significado, Buenos Aires, Eudeba, 1964.

B. Russell, Ensayos sobre lógica y conocimiento (1901 - 1950), Madrid, Taurus, 1966.

N.E. Christensen, Sobre la naturaleza del significado, (1961), Barcelona, Labor ("Nueva colección Labor"), 1968.

V.S. de Z.

Generación de Formas Plásticas

Participantes: Málaga: M. Barbadillo

Madrid: Alexanco, F. Alvarez Cienfuegos, F. Briones, M. de las Casas Gómez, E. Delgado, M. Fernández Barberá, I. Fernández Florez, E. García Camarero, A. García Quijada, A. Martín, J. Peña, I. Ramos, G. Searle, J. Seguí, R. Sempere, S. Sevilla Portillo.

Valencia: V. Aguilera Cerní, J.M. L. Yturralde.

Reuniones: Se celebraron los días 13 y 31 de marzo a las nueve de la mañana

Comunicaciones:

A continuación reproducimos las comunicaciones de Alvarez Cienfuegos sobre color y de Yturralde sobre sistematización del análisis estético de una obra.

Notas sobre Color

A partir de algunos libros consultados he considerado interesante extraer los datos más claros y concretos posibles, para llegar a la obtención de unas normas que nos faciliten el estudio del color.

Programa

Definición del color.-
Características de un color. Representaciones.
Colores puros, colores complementarios.
Sistemas de clasificación del color.
Colorimetría. Bases de la colorimetría. Métodos colorimétricos.
Espectrofotometría.
Bibliografía.

Como todos sabemos un rayo de luz al atravesar el prisma de cristal transparente se descompone en siete bandas expectrales correspondientes a los colores del arco iris, a saber: violeta, añil, azul, verde, amarillo , naranja y rojo.

Onda luminosa - Se caracteriza primero por una longitud y segundo por su cantidad de energía transportada.

No todos los colores que existen en la naturaleza pueden ser captados por el - hombre, el ojo medio humano solamente aprehende aquellos que se encuentran comprendidos entre radiaciones que oscilan entre los 3.500 a 4.000 armstrongs como límite inferior y 7.000 a 7.500 armstrongs como límite superior. Por debajo del límite inferior se - encuentran los rayos ultravioletas y por encima del superior los rayos infrarrojos.

El armstrongs es la unidad usual para medir la longitud de las radiaciones y su valor es la diez-millonésima parte de un milímetro.

Según Rood el centro del espectro de cada color se encuentra como sigue:

ROJO	7.000	armstrongs
ROJO NARANJA	6.208	"
ANARANJADO	5.972	"
AMARILLO NARANJA	5.879	"
AMARILLO	5.808	"
VERDE	5.271	"

AZUL VERDE	5.082	armstrongs
AZUL CYANE	4.968	"
AZUL	4.732	"
ULTRAMAR	4.382	"
VIOLETA	4.059	"

Características de un color.- El color de un foco de luz o el emitido por un objeto puede ser descrito conforme a tres características:

a) Longitud de onda dominante.
b) Intensidad o luminancia.
c) Grado de saturación o factor de pureza

Ya hemos definido la longitud de onda, no así la luminancia que resulta ser el cociente de dividir la cantidad de luz recibida por el color de que se trate, por la superficie iluminada. El factor de luminancia (Y ó B) de un cuerpo difusor en condiciones normales de luz y observación, es la relación entre la luminancia de este cuerpo y la de un difusor perfecto que no absorba nada de la luz recibida, en las mismas condiciones de luz y de observación.

Por último el grado de saturación o factor de pureza que es el grado del color que se estudia, según se aproxime más o menos al color puro correspondiente.

Formas de representación de los colores.- Existen diferentes maneras de representación: En forma plana (Lacouture), en forma esférica (A. Marinier-Lapostolle), en forma biaxial (Oswald), etc.

Para expresar un color en forma numérica o gráfica existen también diversos sistemas. Detallaré solamente el más usual y que es el que tiene mayor vigencia internacional. Se trata del I.C.I. (International Comitté of Illumination).

Como sabemos los tres colores fundamentales desde el punto de vista colorimétrico, son el rojo, el azul y el verde y también es evidente que mediante la combinación de estos tres colores según determinadas proporciones podremos obtener cualquier color.

Si convenimos en llamar (rojo = x, verde = y, azul = z).

Supongamos ahora que queremos representar un color cualquiera, que en su caso más general estará compuesto en distintas proporciones por rojo, verde y azul, es de-

cir: $C = p \cdot x + q \cdot y + r \cdot z$, que también lo podremos expresar escribiendo:

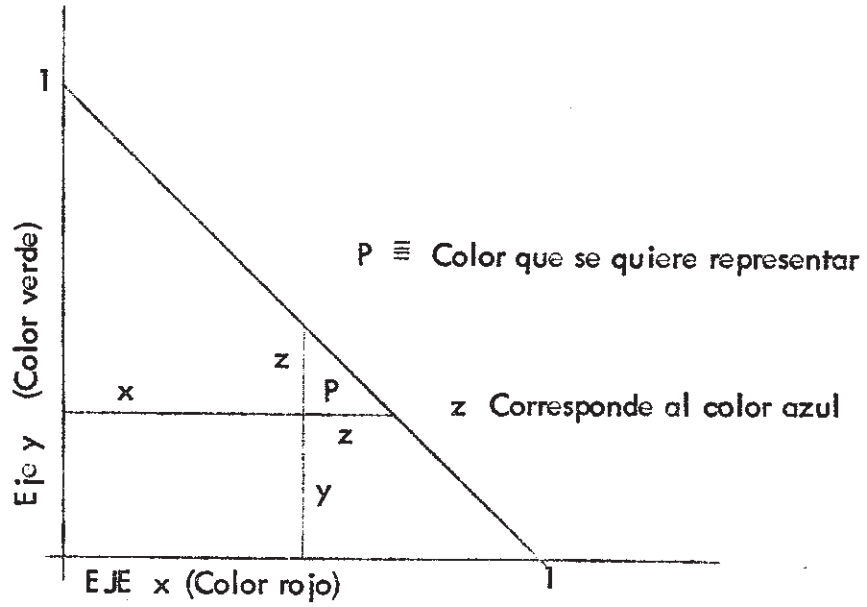
$p \cdot x = \frac{x}{x+y+z}$; $q\,y = \frac{y}{x+y+z}$; $r \cdot z = \frac{z}{x+y+z}$ es decir interesa expresar la cantidad correspondiente de cada luz emitida.

Llamando a: $\frac{x}{x+y+z} = X$; $\frac{y}{x+y+z} = Y$; $\frac{z}{x+y+z} = Z$

Podemos suponer además que la cantidad de luz total emitida por el color C sea igual a uno (solamente nos interesa el tono de C).

Entonces resultaría: $X + Y + Z = 1$

Mediante esta conversión conociendo dos bandas espectrales del color dado C podremos obtener la tercera . Gráficamente esto responde al triángulo cromático del - I.C.I. Dicho triángulo consiste en dos ejes coordenados rectangulares: Sobre el horizontal llevamos la cantidad X que hay en C (Banda roja); sobre el eje vertical llevaremos la cantidad Y (Banca verde) y sobre ambos ejes y a la distancia unidad irá apoyada una recta que formará con dichos ejes un triángulo rectángulo isósceles (Como indica la figura), siendo el valor correspondiente a Z el que se muestra en la figura, es decir la distancia desde P hasta la hipotenusa del triángulo siguiendo una dirección cualquiera de los ejes.



Triángulo cromático del I.C.I. - Forma de representación de un color.

Colores puros - Son los que corresponden a las diversas bandas espectrales monocromáticas.

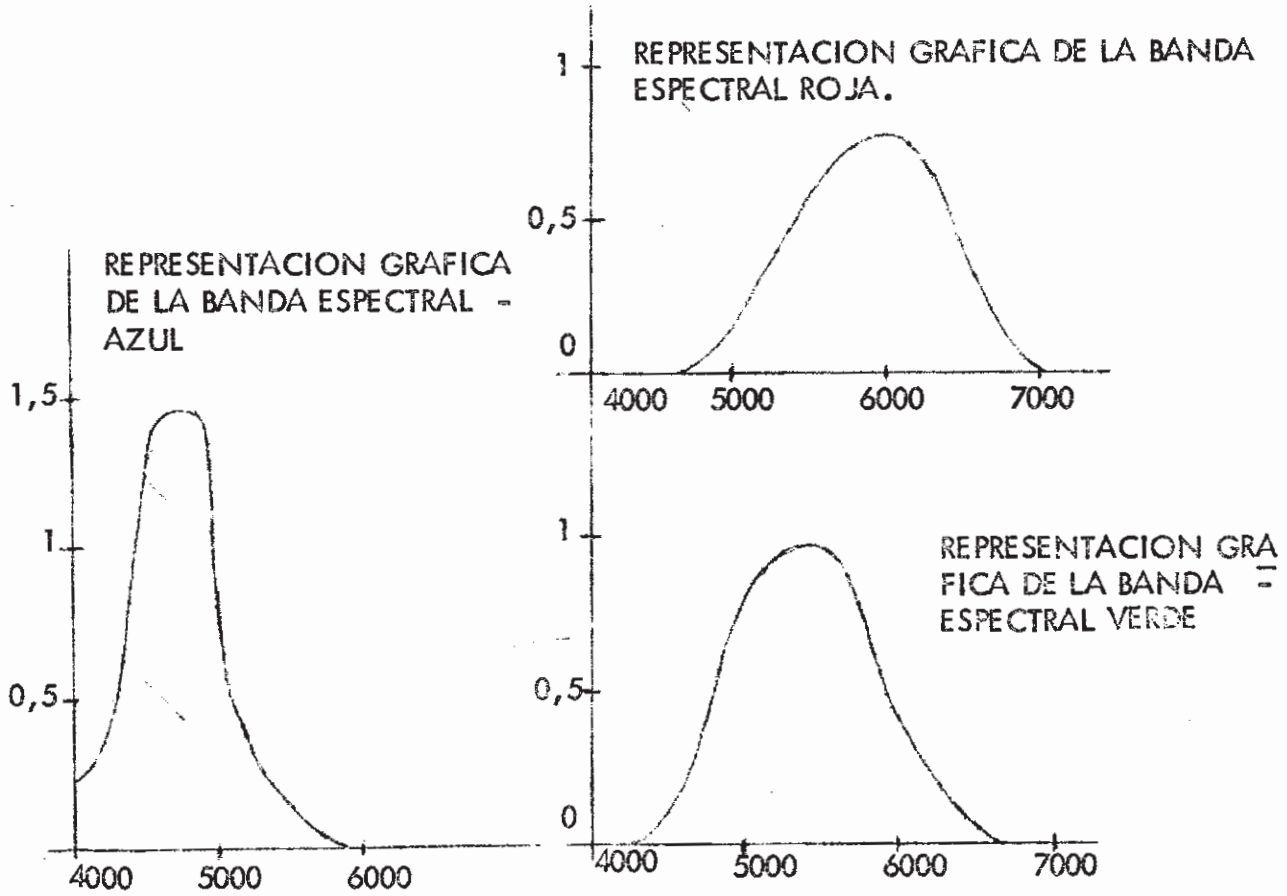
Dos luces se denominan complementarias cuando al actuar simultáneamente y en proporciones convenientes sobre el ojo medio convencional producen en él la impresión de una luz blanca.

Cuerpo blanco - Difunde en la absorción todas las radiaciones que recibe.

Cuerpo negro - Absorbe íntegramente todas las radiaciones que recibe.

Cuerpo gris o neutro - Difunde de forma igual o uniforme las diversas radiaciones que recibe.

Cuerpo coloreado - Es aquel que no es perfectamente blanco, gris o negro.



En estas gráficas colocamos sobre el eje de las x las longitudes de onda dominante, y sobre el eje de las y colocamos la intensidad de luz emitida por cada radiación.

Métodos de Medida

Los métodos actuales para la medición de colores pueden dividirse en:

c) Métodos que se refieren únicamente a las sensaciones del ojo humano medio. Estos métodos se llaman colorimétricos. Los aparatos correspondientes de medida son los colorímetros.

b) Los métodos puramente físicos (Métodos espectrofotométricos)

Visibilidad - Consiste en la capacidad del ojo humano medio para asimilar los diversos conjuntos de colores. De la obra de Pérez Dolz "La armonía de los colores" recojo algunos pares de colores muy visibles.

NEGRO sobre AMARILLO
VERDE sobre BLANCO
ROJO sobre BLANCO
AZUL sobre BLANCO
BLANCO sobre AZUL
NEGRO sobre BLANCO
AMARILLO sobre NEGRO
BLANCO sobre ROJO
BLANCO sobre VERDE
BLANCO sobre NEGRO
ROJO sobre AMARILLO
VERDE sobre ROJO
ROJO sobre VERDE

Bibliografía:

P.J. Bouma. La adaptación cromática del ojo
Revista técnica Philips. 9, P 257 1947-1948

P.J. Bouma. Los colores y su percepción visual.
Paris Ed. Dunod 1940.

Eduard Fer. Armonía de los colores. Tablas de colores.
Editorial Dunod Paris 1958.

W.D. Wright. La medida del color.
Editorial Hilger and Watts. Londres 1958.

Colores normalizados.- 1969
Instituto de Racionalización del Trabajo, Serrano 150 Madrid.

El color en las actividades humanas. Maurice Deliberé
Editorial Tecnos. O'Donnell 27 - Madrid.

Teoría de la forma y del color. Hans Interreiter.
Ediciones Ebusus. Barcelona.

Wright (W.D.) Researchs on normal and detective colour vision.
P. 224 London. Henry Kimpton 1946.

Introducción al color. R.M. Evans. Londres 1948.
Editorial Chapman and Hall.

La reproducción del color. The Fountain Press. Londres 1957.

La caracterización de los colores. M. Richter.
Revista Deustch Ing. Num 14 - 1956.

O. Sireeni. Standard for colours and basic colour substances
(Helsinki 1953).

Del mismo autor: Colour standards (Atlas de colores)
Helsinki 1956.

Illumination Engin (USA) diciembre 1942
Brainerd y Massey.

La conquista del color. R. Maurel y Brunais - Paris 1956.

J. Beaudeneau. La armonía de los colores. Paris 1952.
Dunod Ed.

Revue Science nº 117, 1953.
Area balance in colour armony, and experimental study.

F.A.C.

El Centro de Cálculo de la Universidad de Madrid, a través de uno de sus Seminarios ha puesto a disposición de los que venimos trabajando en el lenguaje, codificado o no, de representaciones gráficas o corpóreas, una poderosa herramienta de nuestro tiempo, el ordenador electrónico y el equipo de analistas y técnicos del mismo, que al igual que en otras disciplinas nos abren nuevas posibilidades para una investigación más eficiente.

Se nos planteó en primer término el problema de cómo utilizar estos medios de una manera eficaz, lo que nos obliga a reconsiderar nuestros propios métodos de trabajo y definir nuestra intención, debiendo profundizar al máximo y presentar los problemas de la forma más exhaustiva posible.

Comenzamos nuestras reuniones en el Seminario bajo el título de "Generación de Formas Plásticas". Ahora bien, según ha demostrado la Psicología de la Percepción, no existen formas básicamente "bellas" que nos sirvan de punto de partida o muestra para que todas las que se acerquen a ellas tengan que ser necesariamente bellas. Cada elemento, cada forma tiene su valoración particular en la determinada circunstancia en que se halla, y por lo tanto creo de interés aclarar que lo que buscamos debe tener un sentido más exacto. Las formas y sus desarrollos tienen infinitas variantes en varias dimensiones. Nosotros las utilizamos como elementos expresivos de un lenguaje, es decir como un sistema de signos que sirven para transcribir como intermediarios un pensamiento o una "información". Esto es, que la "Generación de Formas Plásticas" debe surgir de una necesidad de eficacia comunicativa y por lo tanto responder a una "intención", creando un estado de control exhaustivo de las formas y medios expresivos, en concordancia con una información adecuada de los datos, estímulos de las formas y colores, direcciones lógicas pregnantes, luz, movimiento, etc. etc. y su utilización racional en la expresividad.

Nosotros hemos venido trabajando de forma intuitiva, y llamando "sensibilidad" al conocimiento adquirido mediante la práctica de los fenómenos últimamente analizados por las teorías "Gestalticas". El aprendizaje de dichos fenómenos nos dá una conciencia y control de dichos datos que podremos así utilizar sistemáticamente para nuestros fines.

Por nuestra parte deberíamos:

1.- Conocer el computador y su modo de operar
(propongo que una reunión por lo menos la dediquemos a ésto)

2.- Una metodología de trabajo

3.- Una información adecuada

En cuanto al método de trabajo surgen varias direcciones o grupos de interés:

A) - Método Analítico

1.- A partir de una obra o un elemento básico, modulación de espacio bi y tridimensional, encontrando variantes, multiplicando imágenes.

2.- Proyección, abatimientos, variantes topológicas.

3.- Representación gráfica de ecuaciones matemáticas.

Una vez hecha la propuesta formal se imponen unos criterios selectivos y de valoración.

A este método le encuentro el inconveniente de que parte de unos presupuestos más o menos arbitrarios y dependientes de la intuición.

Considero que podríamos desarrollar un METODO SINTETICO que con unas premisas conocidas y controladas y mediante un proceso adecuado podamos obtener unas conclusiones buscadas.

Propuesta metodológica para un proceso de creación

Proyección o generación de una forma o un objeto

Definición del mismo

BIDIMENSIONAL	Diseño visual	Signos Señales Símbolos	Significado, relaciones, en función de
	Diseño gráfico	Aplicación de las reglas del sistema gráfico	Organigrama Mapa Estadística, etc.
	Diseño de investigación	Nuevas propuestas visuales, estructurales Búsqueda de nuevas imágenes autónomas integradas o no.	

TRIDIMENSIONAL	Diseño Arquitectónico	
	Diseño Industrial	
	Diseño de investigación	Nuevas propuestas estructurales-plásticas y visuales de tres o más dimensiones (espacio-tiempo).

El objeto debe llenar una o varias funciones - Especificarlas, así como posibles relaciones

comunicar - contemplar - manipular - habitar - jugar, etc. etc.

Sentido, intención de la forma o el objeto

DATOS A CONSIDERAR

EL OBJETO EN SI

FORMA VISUAL	Abierto - Cerrado Anguloso - Redondo Proporción Fenómenos Pregnantes Figura - Fondo			
COLOR	Tono Escala	Claro - Oscuro	brillante mate	Grado de Saturación Luminosidad Contraste
TEXTURA	Plana Profunda			
MOVIMIENTO	Percibido Real	Contínuo Acelerado Diminuído		

TIEMPO
- Presente
- Pasado (recuerdo)
- Futuro (Esperanza)

SUPERFICIE TACTIL
- Duro y liso - Duro y Aspero — elástico - plástico
- Blando y liso - Blando y Aspero — frío - caliente

FORMA HAPTICA
- Rígida - Flexible — equilibrio haptico
- Elástica - Plástica — concordancia

CINESTESICA
- Pesado - ligero
- Vacio - lleno

AUDITIVA
- Resonancia
- Eco
- Silencio

OLOR Y GUSTO

RITMO

Relación entre las distintas modalidades - Tendencia a la transformación

Fenómenos perceptivos - ópticos, de comportamiento

DATOS A CONSIDERAR

EL OBJETO EN RELACION CON EL MEDIO

Ubicación
- Fijo
- Móvil

Ambiente
- Exterior - Interior

Elementos de Influencia directa — Próximos
Elementos de Influencia indirecta — Lejanos

Fenómenos ópticos de comportamiento, etc.

El objeto en relación con determinantes

CULTURALES
SOCIALES — ECONOMICAS — POLITICAS
GEOGRAFICAS

Tecnología — Realización

MATERIAL { Características Expresivas / Características Físicas — Definición y elección del mismo

Técnica

Manual — Artesano
Maquinista — Método de fabricación

La ordenación y planteamiento de una investigación racional en este caso implica evidentemente INFORMACION ACTUALIZADA. Ante la imposibilidad de memorizar todos los datos necesarios se nos plantea un problema largo y difícil.

Tengo entendido que con respecto a esto el Centro de Cálculo está ya en contacto con un Centro de Informática y sabemos que se va a construir uno en España. Mientras tanto propongo que nos pongamos en contacto con:
Hochschule Für Gestaltung - 79 Ulm / Donau - Am Hochstrasse 8 (Alemania)
Centro FAS de Bolonia (estudios de estética experimental) Doctor Carmelo Genovese.
Via Oberdan, 15 (Italia)
Profesor R. Borsari - c. Pepoli, 14 Bolonia (Italia)
Profesor Silvio Ceccato (Director del Centro de Cibernética y estudios lingüísticos de la Universidad de Milán) c. so Porta Vittoria, 32 - 20122 Milán
Universidad de Bridgeport - Connecticut (para el Albers Color Course) Profesor August Madrigal o Sybil Wilson, Robert Morris, John Day, Jennet Lam.
Y los centros o personas que puedan interesarnos y pedirles específicamente un intercambio de información con respecto a las materias propuestas.

Los datos necesarios podrían ser: Recopilación de comportamientos, fenómenos perceptivos, hechos estructurales, económicos, geográficos, estadísticos a todos los niveles, etc.

La documentación es suministrada por ejemplo en el laboratorio de Cartographie et Centre d'Analyse des Documents Graphiques (C.A.D.G.), 131, Bd Saint-Michel, Paris (5e) en forma de microfilsm (75x105) o diapositivas (35x50 mm) en negro o en color y pueden ser vistas por medio de ampliadoras especiales que permiten una lectura inmediata o bien se piden fotocopias a escala deseada.

Desde el punto de vista del aprendizaje de las causas, estímulos sensoriales y su percepción, son especialmente interesantes, las figuras ambiguas o "imposibles" estudiadas por L.S. y R. Penrose en su artículo "Objetos Imposibles - Un tipo especial de Ilusión" -- British Journal of Psycology - 49 - 31 - 1958). Estas figuras se obtienen representando una estructura aparentemente tridimensional con datos bidimensionales. En estos casos el sistema perceptivo no recibe la suficiente información. Para localizar en profundidad las distintas partes de la figura. Nuestra experiencia sensorial es impotente para aclarar la ambigüedad de esta información. Ante la visión de una "figura imposible" el fenómeno que suele presentarse es el de aceptarlas en primer lugar como "normales" pero cuando el ojo recorre inmediatamente la totalidad de la figura y trata de analizarla, comprobamos su falta de sentido, creándose un conflicto entre la tendencia estructurante del proceso perceptivo y el análisis de la imposibilidad que le ha sido planteada.

Otros efectos producidos son las distorsiones de la falsa perspectiva aplicada, aunque estas distorsiones son casi siempre factibles de "compensar" en el dibujo y hacer así más evidente la imposibilidad de la imagen. En muchos casos se presentan unas deformaciones aparentes de las líneas por nuestra tendencia a dar sentido a la totalidad de la imagen y estructurar en sentido de profundidad la figura. La mecánica del error de nuestro sistema perceptivo trabaja así activamente tratando de reconstruir una imagen aceptable desde el punto de vista de nuestra experiencia tridimensional. Nos resistimos claramente a aceptar el error y ante la insolubilidad de la propuesta visual se crea un estado de confusión.

Puedo añadir como experiencia y después de un cierto tiempo dedicado a dibujar y pintar estas figuras que me he acomodado a su visión y representación, siéndome a veces más difícil dibujar una figura análoga pero posible que otra "imposible", creándose así un fenómeno inverso.

Dentro de los supuestos metodológicos que seguimos bajo la denominación "Antes del Arte" hemos tratado, continuando la línea general de estudios de los medios expresivos

del lenguaje plástico, trasladar a un contexto artístico, datos y nociones científicas, en este caso las "figuras imposibles", realizando estos ejemplos con los siguientes criterios:

1.- Profundizar, aportando estos datos, en el conocimiento general de los fenómenos perceptivos.

2.- Demostrar, una vez más, que el conocimiento básico de la fenomenología, comportamientos y hechos estructurales, semánticos, etc. del mundo visual y perceptivo, son necesarios para el desarrollo de una comunicación "eficaz" en la obra plástica.

3.- Crear unas imágenes, racionales y eficaces dotándolas del carácter y expresividad propias de un alto nivel artístico pero basado en datos y hechos verificables.

Descripción y Método

Objeto bidimensional, representado mediante una falsa perspectiva, una estructura aparentemente tridimensional pero imposible de representar verdaderamente en tres dimensiones.

Además de la finalidad expresada anteriormente debo añadir que desde el punto de vista psicológico e intencional, me ha interesado la tensión emocional que puede producir en el espectador la visión de estas imágenes, paradógicas en sí mismas y creadoras de un enigma visual, y que puedan, además de su pretendido simbolismo aportar una nueva dimensión formal. Para mí son y pretendo lo sean para todos, un símbolo más del universo tenso y paradógico en que vivimos. Y me refiero de modo especial como universo al hombre de hoy.

Desde el punto de vista formal, y siguiendo un orden, he procurado dar en estos casos las siguientes características que acentúan o modelan la visión total de la imagen.

FIGURA CERRADA para captar y fijar la atención que estudia así la imagen completa.

ANGULOSA estructura geométrica sencilla, tendiendo a las formas básicas pregnantes (cuadrado, círculo, etc.) pero sin salirse de una expresión agresivo-activa, acentuada por la articulación y angulusidad de la figura.

FIGURA FONDO la figura se destaca claramente del fondo, este último muy neutro y alejado.

SIMETRIA - ASIMETRIA las figuras Asimétricas dan más sensación de profundidad y se evidencia más su estructura. Las figuras Simétricas suelen tener una valoración estética más

alta, pero desde el punto de vista "información sensorial" son menos evidentes. He realizado ejemplos en ambas direcciones dando así distintos matices a la intención general.

RESUMEN tendencia a evidenciar y resaltar la figura dándole el máximo relieve y articulación, dentro de un contenido pregnante y separando totalmente la representación del fondo.

COLOR colores básicamente fundamentales y pregnantes (rojo, amarillo, azul) aprovechando al máximo su tendencia a la transformación para acentuar el relieve de la figura sin necesidad de utilizar el claro-oscuro. Contraste máximo mediante complementarios. Tonos mates o semimates de un alto grado de saturación y luminosidad máxima.

(habría que precisar más en cada representación concreta, aquí doy líneas generales a seguir sin especificar ni hacer una propuesta exhaustiva).

TEXTURA plana "impersonal", reflejo del mundo de la máquina y la seriación. (la obra está pensada como prototipo de una posible serie).

MOVIMIENTO PERCIBIDO figuras dotadas de un lento movimiento (en sentido figurado) horizontal, hacia afuera y hacia el espectador, en las figuras simétricas, en las asimétricas el movimiento es hacia afuera y lateralmente. Estas sensaciones se consiguen sobre todo por el color, el sentido de la perspectiva y el tono lejano y quieto del fondo (negro, azul oscuro, o gris) y la oposición constante frío - cálido figura - fondo.

SUPERFICIE TACTIL dura y lisa

CINESTESICA mediante la tendencia a la transformación percibimos la "potencia" de un objeto sólido y por lo tanto dotado de un cierto peso.

AUDITIVA Sensación de silencio y vacío.

RITMO general monótono, cortado sin embargo de forma un tanto brutal no por la articulación de la figura sino por la conseguida percepción de imposibilidad de la imagen.

DIMENSIONES en consonancia con el sitio donde vaya a ubicarse la imagen.

TECNICA obra prototipo, pensada para realizarse en varias escalas tipo. Como diseño - gráfico sobre papel las más pequeñas (procedimiento de serigrafía o litografía) y tamaños mayores sobre madera prensada, preparada y pintada con procedimientos industriales.

Bibliografía más corrientemente utilizada:

SVEN HESSELGREN
Los medios de expresión de la Arquitectura
Ed. Universitaria (Buenos Aires)

JACQUES BERTIN
Semiologie graphique
Ed. Garthier - Villars

GRILLO DORFLES
El diseño Industrial y su estética

R.L. Gregory - Ojo y Cerebro Ed. Guadarrama

Aruheau = Arte y percepción Visual
= Nueva Visión =

Aguilera Cerni
Catálogo de la Exposición "Ritos del Arte"

J.M.L.Y.

Felicitamos a los participantes de nuestros Seminarios ALEXANCO y J.M. YTURRALDE, por los premios que recientemente obtuvieron en el - X Salón Nacional de Marzo celebrado en Valencia. Los premios fueron - Medalla Mariano Benlliure de Escultura para Alexanco y Medalla Jose - Ribera de Pintura para Yturralde.

Generación de Formas Plásticas

Participantes: Málaga: M. Barbadillo

Madrid: Alexanco, F. Alvarez Cienfuegos, F. Briones, J.L. de Carlos, M. de las Casas Gómez, E. Delgado, M. Fernández Barberá, I. Fernández Florez, E. García Camarero, A. García Quijada, A. Martín, I. Ramos, G. Searle, J. Segúi, R. Sempere, S. Sevilla Portillo.

Valencia: V. Aguilera Cerní, J.M., L. Yturralde.

Reuniones: Se celebraron los días 10 y 24 de abril.

Comunicaciones:

A continuación reproducimos la comunicación de E. Delgado sobre aplicación de las computadoras a la Generación de Formas Plásticas:

El gran desarrollo alcanzado por las calculadoras electrónicas a partir de la segunda guerra mundial, y su gran aplicación en todos los campos tanto científicos como experimentales, ha hecho que el hombre las considere hoy día como elemento imprescindible en todo proceso de investigación, dada su gran aplicación a la resolución de problemas complejos y a su rapidez y exactitud en los resultados.

El hombre, consciente de la importancia del hallazgo del calculador electrónico como herramienta de cálculo, ha intentado explotarlo llevando su aplicación a campos que quedaban fuera de los previstos. Así el gran desarrollo que en Estados Unidos experimentaron las calculadoras durante la segunda guerra mundial fué consecuencia de la complejidad de los problemas planteados y la necesidad de resolverlos con la mayor rapidez.

La aplicación de los computadores a problemas tan complejos como el control de los volúmenes de producción de las empresas, la aplicación en el campo de la medicina con la aparición de las máquinas de diagnosticar así como en el campo económico nos lo revelan como la herramienta más útil para la investigación.

Debido a todo esto podemos pensar que estamos asistiendo a una nueva era, " la era de las computadoras" que trae consigo la era de la automatización. En efecto, esta nueva era, que solo pueden conocer en toda su extensión los países más avanzados tecnológicamente, se implantará cada vez con mayor rapidez en éstos haciendo que con su -

poderío industrial crezca cada vez más en relación a los países menos adelantados, de tal manera que refiriéndonos a uno de los informes realizados sobre el porvenir tecnológico de Europa con relación a los Estados Unidos por el Instituto Gallup prevé éste que Europa dentro de veinte años será, con relación a los Estados Unidos, lo que el antiguo Egipto es hoy con relación a Europa.

El progreso científico, sin embargo, podemos decir que no ha sido "expansivo", en el sentido de que sólo ha sido científico, y que ha producido una individualización de los sectores más avanzados intelectualmente dando lugar a la creación de minorías muy selectas que han tenido acceso a las nuevas ciencias que en el mundo han ido creándose y al proceso de investigación de dichas ciencias. Por otra parte hay una inmensa mayoría de los hombres que no han tenido acceso ni a estas técnicas ni al lenguaje por ellas creado y que a pesar de tener una formación intelectual sólida ha quedado atrás.

Observamos un profundo divorcio entre estas dos clases de hombres, de manera que la minoría selecta formada por los científicos da lugar a una supercivilización(1) a la cual no tienen acceso los demás. Este divorcio, observamos que es cada día más profundo en el terreno de las ciencias y de las artes, de manera que comparativamente podemos observar el gran desarrollo experimentado por todas las ciencias exactas y aplicadas y la posición de inmovilismo total que se ha producido en el campo de las artes.

No quiere esto decir que en el campo de las artes no haya habido intentos renovadores (que los ha habido), sino que los pocos intentos que en este campo se han producido han sido poco coherente, faltos de sistematización y que estos intentos se han limitado a experimentar individualmente nuevas teorías y nuevas formas de ver el arte de manera irregular y poco metódica, lo cual ha provocado en los artistas períodos de lucidez al descubrir soluciones nuevas en el arte y períodos de estancamiento producidos al no saber desarrollar dichas formas y no darles un planteamiento formal.

El problema se presenta muy complejo de solución pues esta labor personal de búsqueda, dentro de las artes, ha sido siempre patrimonio exclusivo del artista, de forma que cuando este encuentra una forma plástica nueva que la satisface se la atribuye como propia, y en muchos casos no la desarrolla al máximo limitándose a jugar con ella

---

(1) Utilizamos el término supercivilización en el sentido de crear una civilización superior basada en un mayor conocimiento científico y una mayor información.

y haciendo perder universalidad a su descubrimiento, dándole un sentido excesivamente circunscrito a su persona y a su forma de ver las cosas.

Esto no quiere decir que se niegue la creación personal, pues esto sería absurdo, pero sí afirmar que, hasta nuestros días, muchos artistas han rodeado a su obra de un matiz de misterio en cuanto a su creación y composición, que ha supuesto una tara realmente grande al proceso de evolución del arte en el sentido más universal, siendo esta evolución puramente personal y manteniendo un principio creador que se ha llamado "genio" o también inspiración.

Hace ya algunos años se publicó en París un libro en el cual se hacía un examen exhaustivo del teatro y de sus situaciones; se llegó a la conclusión de que una obra de teatro se podía plantear de cincuenta formas distintas y con arreglo a estas fórmulas, desarrollar dicha situación para llegar al resultado apetecido. Ahora, al correr el tiempo, vemos como aquel libro que pareció situar al teatro en un callejón sin salida está totalmente superado por la aparición de nuevos problemas, nuevos estilos y nuevas formas de comprender el teatro haciéndole cambiar de esencia.

Pues bien, esto mismo es lo que ha sucedido con la pintura, la música y la poesía. Podemos pensar en quedarnos para siempre encasillados en unos moldes rígidos?. A todas luces la respuesta es que no. El empeño por romper con formas exageradamente rígidas y poco atemperadas a la realidad se ha observado en todos los ámbitos artísticos, así nace la música dodecafónica, como intento, como búsqueda, como posibilidad de enriquecimiento; más tarde nacerá la música electrónica y poco después la música funcional se irá incorporando poco a poco en bandas sonoras que nos pasarán desapercibidas y que han tenido su aplicación en el cine.

Paralelamente la poesía sigue por los mismos cauces, intenta perder de vista la métrica y el ritmo que en otros tiempos fueron considerados como los criterios básicos de acuerdo a los cuales se decidía si la poesía era o no buena, la necesidad de romper estos moldes para atemperar la poesía a la vida actual, da lugar a diversas corrientes; y por fin nace como corriente más avanzada la poesía concreta.

El mismo fenómeno se produce en la pintura, con la incorporación de nuevos materiales, y así vemos que ésta se desprende de los tubos de color para ir en busca de nuevos sistemas de representación, de forma que llegamos no solo a formas nuevas sino a entrever nuevos caminos. Sin embargo y a pesar de todos estos cambios trascendentales, que de hecho marcan épocas y estilos la evolución de las artes en relación con las ciencias exactas y aplicadas ha sido mínima. Por eso es de vital importancia afirmar

que hoy día las artes deben de ponerse al día de una forma total y trabajar con los instrumentos más eficaces, según hemos ya dicho, es la computadora, y su utilización e incorporación como útil de trabajo al campo de las artes es una tendencia y un deseo compartido por numerosos artistas, que ven en ella una solución a problemas complejos cuya resolución supondría mucho tiempo y vasto conocimiento científico.

El campo de utilización de estas máquinas es verdaderamente colosal, abarcando desde el figurativo más académico hasta las composiciones más atrevidas al estilo de Mondrian o de Pollock, observando como estas composiciones intuídas individualmente por los pintores tienen un planteamiento formal riguroso de acuerdo con unas leyes matemáticas, probabilísticas o físicas.

Así planteado el problema, podemos pensar en utilizar las computadoras, dando un sentido riguroso a la investigación de formas plásticas, (investigación que creo inédita y muy importante). Inmediatamente se nos plantea la pregunta Cuál es el objeto de dicha investigación?. La respuesta no tiene espera: dar un nuevo y radical planteamiento a las artes, incorporando a las mismas los principios científicos mediante los cuales podamos obtener determinadas formas plásticas, que se ajusten a unas leyes de composición y desarrollar dichas formas utilizando dichas leyes.

Muchos pensarán que este esfuerzo de investigación va en contra de la esencia misma del arte y que lo deshumaniza sometiéndole a reglas y leyes todavía más estrictas que las que anteriormente conocía el hombre, pero los que así piensen pierden de vista (quizá porque no han sido iniciados en este terreno) la belleza que se puede encontrar en determinadas formas matemáticas representables por medio de ecuaciones más o menos complejas. Entonces Es que el arte se va a convertir en una ciencia pura como puedan ser las matemáticas?. A esto contestamos que no, pero también pensamos que las ciencias exactas y aplicadas pueden prestar una gran ayuda al arte a encontrar nuevos caminos enriqueciéndolo hasta límites sorprendentes para el mismo artista.

Naturalmente, en esta investigación los obstáculos con que chocamos son múltiples, ya que si queremos darle un planteamiento formal y riguroso, necesitaremos unos conocimientos básicos matemáticos, conocimientos que en muchos casos es prácticamente imposible adquirir, pues supondría toda una vida de estudio y para resolver este problema tenemos las computadoras.

Hoy día la programación de formas y de familias de curvas por medio de computadoras es un problema relativamente sencillo pero que requiere conocer dichas curvas; sin embargo el avance de la técnica en este terreno nos ha proporcionado aparatos

tan útiles como el "dis-play" que consiste en una pantalla sobre la cual tenemos una representación gráfica del problema a estudiar y que podemos modificar a voluntad sobre la misma utilizando un lápiz electrónico "pen-light", con el cual modificamos las coordenadas del dibujo directamente sobre la pantalla, obteniendo el resultado apetecido. Vemos por tanto que esta investigación trae consigo la necesidad del trabajo en equipo entre los científicos y los artistas.

Algo análogo pasa con los módulos arquitectónicos y con su investigación, pero ante todo Qué es un módulo?. Pues sencillamente podemos definir como módulo a toda forma, simple o compleja, que se ajuste a unas leyes de composición y que cumpla determinadas condiciones restrictivas.

El problema del estudio de los módulos arquitectónicos y su aplicación a la vida real considero que es de gran importancia, por lo que puede tener de aportación para crear una nueva estética. Naturalmente este estudio individualmente es casi impracticable en el sentido de que se llega a resultados muy limitados; esta limitación nos la resuelven las computadoras, las cuales nos dan el número exacto de combinaciones que se pueden hacer de un módulo simple. Naturalmente, el pretender dar forma a todas ellas es una tarea casi irrealizable, pero sí podemos analizar aquellas formas que pueden resultar importantes y realizables.

Como ya hemos dicho anteriormente, se impone el trabajo de equipo y la colaboración entre los hombres de ciencia y los artistas en este intento de conseguir darle a las artes una nueva visión, enriqueciéndola con nuevos principios y criterios, introduciendo nuevos conceptos y basando dicha investigación en una gran comunicación de los métodos y de los resultados obtenidos.

Creo que romper con el conservadurismo en que el arte ha estado encerrado será difícil y que no será tarea fácil tampoco la generación de formas plásticas, pues exigirá una dedicación y también un cambio de mentalidad en la concepción del arte; pero juzgo importantísimo el nuevo camino que tiene el arte planteado entre sí y todavía más importante desarrollarlo al máximo utilizando siempre los principios y métodos más modernos , que aunque no nos den soluciones definitivas nos abran nuevos caminos por los cuales puedan discurrir las artes, sacándolas del inmovilismo en que éstas se han mantenido con relación a las ciencias, y dándoles un mayor dinamismo.
ED.

SEMINARIO DE GENERACION DE FORMAS PLASTICAS

Participantes: Florentino Briones, Ernesto García Camarero, Ramón Garriga, Ignacio Gómez de Liaño, Isidro Ramos y Javier Segui de la Riva.

Coordinador: Ignacio Gómez de Liaño.

Primera sesión: el 20 de octubre

Más que reunión de seminario se tuvo un encuentro, un cambio de impresiones, el primero de este curso. Con esta reunión se comenzaba el segundo año de funcionamiento del seminario de "Generación de Formas Plásticas", teniéndose en cuenta que el curso precedente se había dedicado a plantear, a experimentar la posibilidad del trabajo artístico y de su análisis empleando nuevos métodos y nuevas técnicas, consiguiéndose algunos resultados parciales que quedaron patentes en la exposición de "Formas computables".

Los asistentes coincidieron en que por encima de todo de lo que se trataba era de realizar trabajos empíricos y analíticos sobre la obra de arte para llegar a resultados metodológicos y teóricos en el campo de la estética científica y de la generación automática de formas plásticas.

El trabajo empírico y de análisis se referiría en primer lugar al de las obras de los artistas que entrarían a componer el seminario, prosiguiéndose los ya comenzados el curso anterior con las obras de Alexanco, Barbadillo e Yturralde; a estos nombres se añadieron los de Sempere, Quejido y Molero como reuniendo características de especial interés. Se señaló cómo los trabajos

de este tipo del período anterior habían resultado altamente satisfactorios para los propios artistas. En segundo lugar se propusieron algunos pintores extranjeros cuyas obras merecería la pena fuesen estudiadas, como Vasarely, Kandinsky, Malevitch, Feininger y Paolo Uccello.

Por estar ausentes de Madrid y por otras circunstancias la asistencia a esta primera sesión del seminario fue reducida, previéndose sin embargo para la próxima mayor afluencia. Se trataron también temas de organización interna, tales como crear dentro del seminario equipos reducidos de trabajo que presentaran en las reuniones quincenales sus resultados, igualmente se propuso trimestralmente una reunión interseminarial.

Con todo la discusión se centró en algunos aspectos de la Estética. El coloquio mantenido puede focalizarse en los siguientes temas: el objeto estético; la semántica de la obra de arte y sus implicaciones psicológicas; el factor sociológico. Este coloquio era tanto más importante cuanto que en lo sucesivo la coincidencia en un lenguaje común e inteligente sería la mejor garantía para las investigaciones que se emprendieran.

Gómez de Liaño partía de una concepción de la Estética reducida al análisis del objeto estético, siendo éste un destilado teórico o ideal que hiciera posible el estudio de los objetos artísticos. Señaló en este "objeto" tres componentes fundamentales, uno material o "hilético", del griego "hyle": materia, construido por analogía con la Fonética, a este nivel habrían de ser inventariados los elementos materiales que intervienen en la composición de la obra de arte así como recíprocamente definidos por sus diferencias; el segundo componente sería el sintáctico en el que se estudiarían los tipos de ordenaciones, de estructuras relacionales y conexiones que se dieran entre los "hilemas" al objeto de establecer algunas reglas generales; por último se refirió al componente semántico, considerándolo como una clasificación y jerarquización de las categorías que intervienen en la obra de arte. A este respecto García Camarero señaló la posibilidad y el interés que tendría otra interpretación de la semántica de la obra de arte entendiendo el objeto artístico como un estímulo que "impresiona" al contemplador provocándole un comportamiento determinado. Según esta interpretación podríamos hablar de un campo más o menos determinado de significaciones que habrían de ser estudiadas estadísticamente para tratar de llegar a algunas conclusiones respecto al estímulo artístico y a la respuesta. Segui de la Riva mostró el interés de esta interpretación señalando algunos trabajos realizados en este sentido.

Segui de la Riva al igual que García Camarero no entendían que esta concepción de la Semántica estuviera en contradicción con la de Gómez de Liaño, pudiendo decirse que ambas tenían un campo de aplicación propio. Gómez de Liaño señaló otro tipo posible de estudio analítico del objeto estético a base de los análisis estadístico, métrico y topológico de los elementos componentes de la obra de arte. A este respecto Briones apuntó que a un estudio de este tipo se aproximaba el realizado sobre la obra de Mondrian el pasado curso, señalando que este año debería ser completado añadiendo las dos variables que no se habían tenido en cuenta, el color y la anchura de los trazos.

Varias veces a lo largo del diálogo Garriga señaló la relevancia del factor sociológico en la obra de arte, en lo cual todos los asistentes estaban de acuerdo. Advirtió Garriga que al ser esto lo que más le interesaba trataría de presentar en las próximas reuniones algún plan de trabajo. García Camarero objetó que el planteamiento de nuestros estudios no podría probablemente entretenerse, por exigencias del propio estudio, en variables excesivamente indeterminadas como las sociológicas. Igualmente Gómez de Liaño tras reiterar la importancia que concedía al factor sociológico mostró la dificultad, ya formulada por Marx, de relacionar in concreto el producto cultural y las condiciones materiales dentro de las cuales se produce en el sentido de su cualificación estética.

Se terminó la reunión quedando para el día 10 de noviembre, esperando que para entonces se pusieran en marcha los trabajos de seminario, se añadió que desde ya habría de comenzarse las relaciones con los grupos extranjeros que trabajasen sobre estos problemas, así como seguirse con los artistas e investigadores españoles interesados por la temática. Se prevé para este año como trabajos complementarios del seminario la exposición alemana de gráficos generados por ordenador y las conferencias del Prof. Dr. Abraham Moles y del Prof. Georg Nees.

POSIBILIDADES DERIVADAS DE LA AUTOMATIZACION EN LA CORRECCION DE EXAMENES

POR

M. GIL GAYARRE
I. RAMOS
M.ª JESUS MANZANAS
C. GIL GAYARRE

PUBLICADO EN
REVISTA DE AUTOMATICA
AÑO III - NUM. 7
ENERO - FEBRERO - MARZO 1970

Gil Gayarre, M., Ramos, I., et al., Posibilidades de la Automatización en la corrección de exámenes. Revista de Automática. Año III, nº 7. 1970.

POSIBILIDADES DERIVADAS DE LA AUTOMATIZACION EN LA CORRECCION DE EXAMENES

Por M. GIL GAYARRE (*)
I. RAMOS (**)
M.ª JESUS MANZANAS (***)
C. GIL GAYARRE (****)

Uno de los problemas más arduos, si no el más espinoso, de la Docencia, es el que se presenta a la hora de la valoración de los conocimientos adquiridos por el alumno y el problema es difícil porque entran en juego muchos factores subjetivos por parte del alumno y por parte del profesor, que pueden enmascarar la magnitud que se trata medir: el nivel de conocimientos del alumno. Magnitud que no tiene dimensiones ni formulación física alguna y que carece de unidades aplicables, no ya como criterio internacional, sino ni siquiera de orden nacional, provincial ni local.

No es difícil, dentro de una misma institución docente, el ver cómo cada profesor califica a su manera, según sus propios criterios, a sus alumnos.

La cuantificación, por tanto, de esta magnitud es difícil siendo prácticamente imposible el establecimiento de comparaciones en los diferentes niveles de poblaciones estudiantiles.

La dificultad de expresión oral o escrita, la inspiración extraña (vulgo copia), el desparpajo o la timidez, pueden ser factores modificativos que enmascaran por defecto o exceso la realidad de los conocimientos del alumno. El factor azar, de influencia no despreciable en los sistemas que podríamos llamar tradicionales puede modificar profundamente la impresión que sobre los conocimientos momentáneos de un alumno adquiera el profesor.

La diferencia de estado de ánimo y, por tanto, de criterio, a lo largo de la corrección de un mismo examen en poblaciones estudiantiles grandes, induce aun involuntariamente a injusticias que, pese a la mala fama general suelen ser más por benignidad que por malignidad. La involuntaria, aunque humana reacción positiva o negativa en la información sobre el examinado, ajena a lo que ha reflejado en su examen (conocimiento personal del alumno, cartas de recomendación, etc., etc.), son asimismo elementos que influyen en que la valoración de cada examen concreto no se pueda realizar de manera homogénea frente a todos los alumnos.

No vamos a discutir las ventajas o inconvenientes de los diferentes tipos de exámenes: oral, escrito, mixto, puesto que nos alejaría notablementne del tema; únicamente, sin embargo, señalaremos que el sistema que reduce al mínimo los factores personales de la prueba, es el denominado globalmente "métodos objetivos de examen", una de cuyas formas es el *test* de respuestas a elección múltiple.

La comparación entre las respuestas proporcionadas por el alumno y las contestaciones verdaderas nos dará siempre el mismo resultado, ya sea el examen corregido por el mismo profesor que ha explicado la asignatura, por el ordenador electrónico o por el propio alumno.

El sistema no es ni mucho menos nuevo, ya que ha sido aplicado con carácter local o general en numerosos centros extranjeros (Grenoble, Manchester, diversas Universidades USA). En España, asimismo, se emplea desde hace algunos años en diversas Facultades universitarias y Escuelas Técnicas, así como algunas pruebas de carácter colectivo (ejercicio teórico para la obtención del carnet de conducir).

En este trabajo nos vamos a preocupar del proceso que comenzamos a desarrollar en la Cátedra de Terapéutica Física de la Universi-

(*) Catedrático de Terapéutica Física de la Facultal de Medicina de Madrid.

(**) Analista del Centro de Cálculo de la Universidad de Madrid.

(***) Profesor Ayudante de Clases Prácticas de Terapéutica Física de la Facultad de Medicina de Madrid.

(****) Profesor Adjunto Encargado de Curso de Radiología y Electrología Médicas de la Facultad de Medicina de Madrid.

dad de Madrid para la corrección automática de los exámenes por *test* de respuestas de elección múltiple y las posibilidades que en orden a la valoración de la docencia y de la población estudiantil se pueden derivar. Representa el fruto del equipo de trabajo constituido por el personal de la Cátedra de Terapéutica Física (Dres. Gil Gayarre, Larruga Rey, Machado Quintana, Srta. Manzanas Artigas) y del equipo de Analistas del Centro de Cálculo de la Universidad de Madrid dirigido por el Dr. Briones, (Sres. Isidro Ramos, Julio Montero, Lorenzo Carbonell); también queremos agradecer en la persona del Profesor D. Salustiano del Campo su colaboración en la lectura de marca sensible en el Instituto de la Opinión Pública, así como a los Sres. Serrano, Parreño, Gómez y Gómez, de la firma IBM, por su colaboración en la resolución de los problemas planteados por la lectura en marca sensible.

El proceso total de la automatización en la valoración de conocimientos abarca tres fases que estudiaremos sucesivamente (fig. 1).

1.ª Fase: Elaboración del examen.
2.ª Fase: Redacción práctica del examen y calificación individual del examen.
3.ª Fase: Calificación colectiva de los exámenes y posibilidades de estudio derivadas de la calificación colectiva.

Cada una de estas fases puede ser realizada de manera manual y nuestros esfuerzos han sido dirigidos hacia la automatización total o parcial de cada uno de los procesos.

1.ª Fase: *Elaboración del examen.*

No vamos a entrar en la descripción del examen por *test* de respuestas de elección múltiple por ser suficientemente conocido el procedimiento (1, 2, 3, 5).

La elaboración de las preguntas que han de ser sometidas al alumno es un trabajo que realizamos de forma manual a partir de las fuentes bibliográficas que responden al programa de la asignatura.

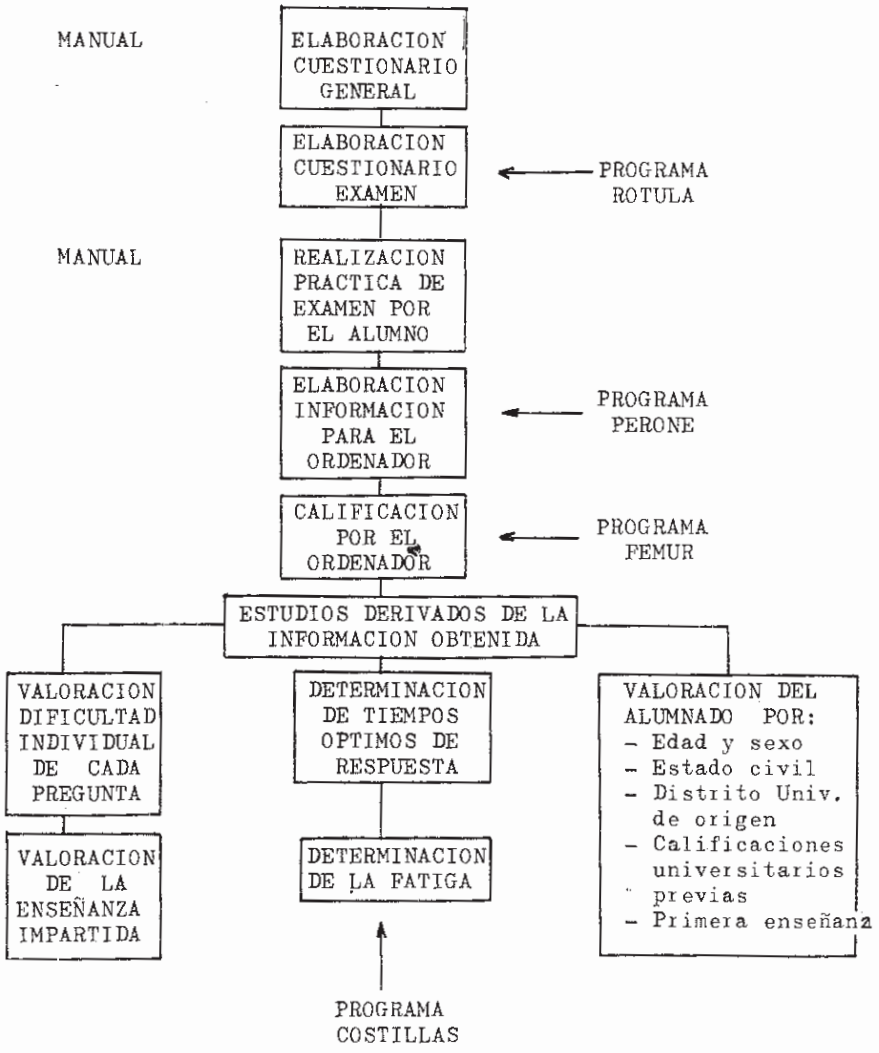


Fig. 1 — Esquema general de la automatización en la corrección de exámenes objetivos por test de respuestas a elección múltiple.

Los trabajos de Sánchez de Zavala (6) en el Seminario de Valoración del Aprendizaje del Centro de Cálculo de la Universidad de Madrid, representan una vía hacia la elaboración automática de preguntas que simplifique el procedimiento.

La elaboración (artesana) de una pregunta con cuatro posibles respuestas, de las cuales una es la verdadera, y que las demás no sean absolutamente ilógicas, con lo cual se puede adivinar la respuesta sin conocer realmente la materia objeto de examen, representa un trabajo que puede oscilar entre los cinco y diez minutos por pregunta.

Ello representaría, teóricamente, un trabajo de unas setecientas cincuenta horas para 6.000 preguntas. Ahora bien, a medida que se van realizando preguntas, el encontrar nuevas posibilidades de respuesta hace que el tiempo de elaboración por pregunta se duplique fácilmente.

Una vez construido el fichero básico de preguntas de examen hay que proceder a la selección de las preguntas que van a constituir la prueba que realizarán los alumnos.

A diferencia de lo que ocurre en los exámenes tradicionales es difícil prever, en las preguntas aisladas, la dificultad de cada una. Sin embargo, *a posteriori* y con experiencia suficiente podemos tener datos estadísticos que permitirían, prácticamente, vaticinar el resultado del examen antes de su realización, es decir, por monstruoso que parezca, se puede aprobar o suspender a todo un curso con un procedimiento arbitrario de selección de preguntas ("fáciles" o "difíciles").

Para evitar esto hemos elaborado un programa que selecciona, al azar, las preguntas del fichero general (fig. 2), en el que el criterio de selección no tiene en cuenta el historial de la pregunta, esto es, si en convocatorias anteriores ha sido contestada correctamente o no por la mayoría de los alumnos.

Unicamente se busca el que el total de preguntas esté homogéneamente repartido a lo largo del programa y que de esta manera no queden lagunas por explorar en los conocimientos del alumno.

En la figura 3 se muestra el organigrama de la selección al azar de preguntas.

En el fichero general cada pregunta viene determinada por un número de 7 dígitos, de los cuales los dos primeros indican el número de la lección dentro del programa; el 3.º y 4.º indican el item dentro de cada lección y los 3 últimos expresan el número de cada pregunta dentro de cada item. Esta última ordenación no sigue en la actualidad ningún criterio fijo y las preguntas se van numerando a medida que se van fabricando al realizar un examen.

En cada caso concreto fijamos previamente las lecciones que van a ser objeto de examen y el número total de preguntas, lo cual nos proporcionará el número de preguntas de cada lección que han de entrar en el examen. Si este número fuera mayor que el de items contenidos en cada lección determinaríamos asimismo el número de preguntas de cada item objeto del examen. Un criterio de selección haría que al rebasarse el número de preguntas señaladas para cada item o para cada lección la pregunta generada no fuera introducida en el examen y se generará otra nueva.

La elaboración manual de un examen de 100 preguntas representa, aproximadamente, cincuenta horas/hombre, según el reparto siguiente:

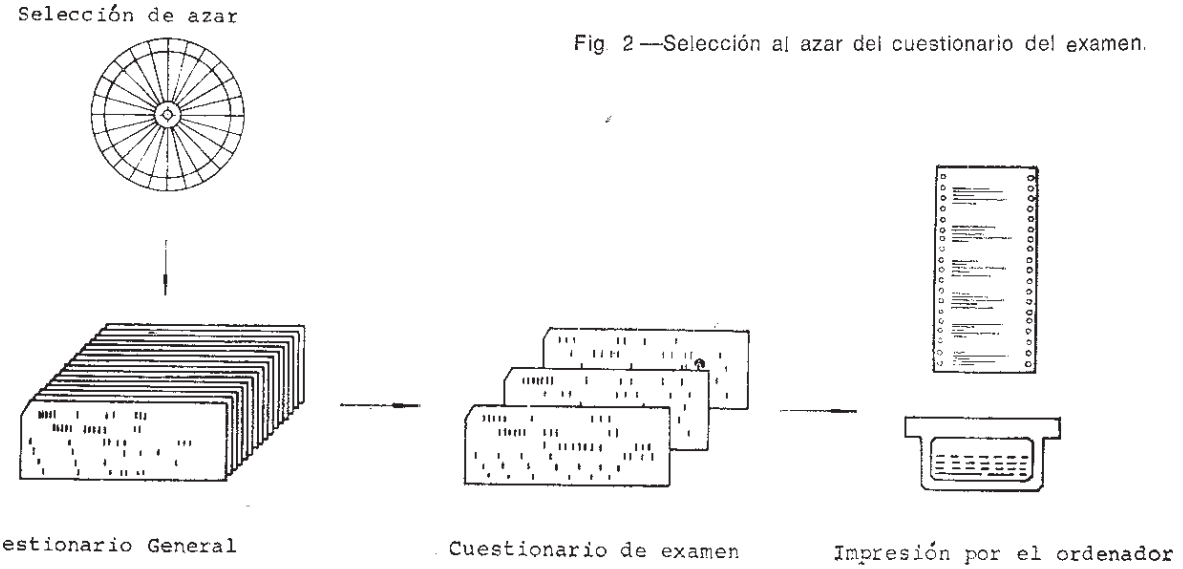


Fig. 2—Selección al azar del cuestionario del examen.

	Minutos
Confección de 100 preguntas a 7,5 minutos por pregunta	750
Elaboración de los clichés (10 preguntas cliché) a un promedio de 20 minutos por cliché (teniendo en cuenta que cada 100 preguntas se realizan 3 cuestionarios diferentes en los que las preguntas están en orden distinto)	600
Verificación de los clichés (4 personas) durante 3 horas	720
Tirada a multicopista de unos 40-50 ejemplares por cliché	150
Confección de los cuadernillos y verificación de los mismos (6 personas) durante dos horas	720
TOTAL	2 940

Esta labor realizada en el ordenador representa una economía en tiempo aproximadamente de la séptima parte de la labor correspondiente a la elaboración de los clisés y del procedimiento de multicopia.

2.ª Fase: *Realización práctica del examen y calificación individual del examen.*

Existen, fundamentalmente, tres procedimientos para que el alumno conteste al cuestionario elaborado como hemos indicado en el apartado anterior.

En orden de mayor a menor complejidad de elaboración posterior es:

1.º Inscripción de los resultados en papel de marca sensible.

2.º Perforación de tarjeta. Otro procedimiento muy utilizado es el uso de tarjetas pre-

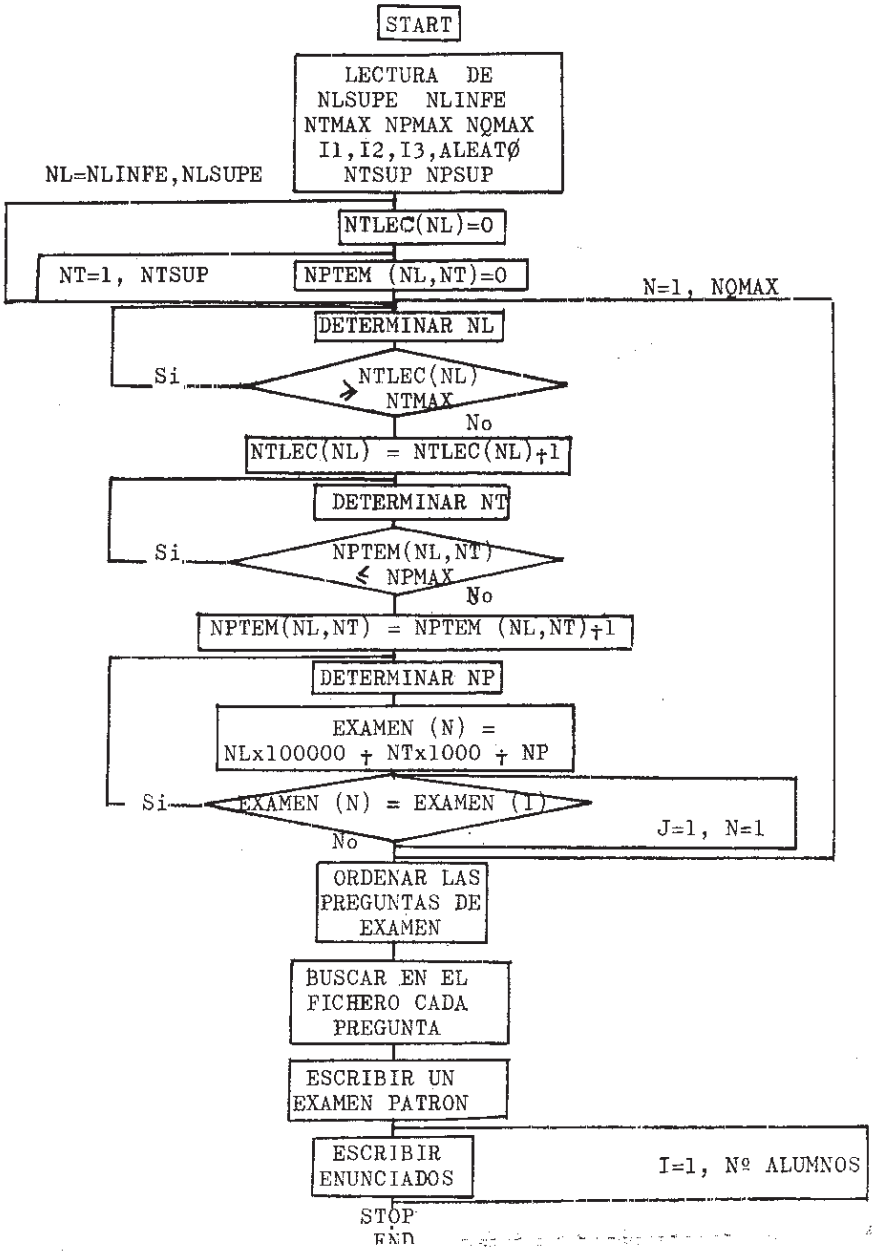


Fig. 3—Organigrama de la elaboración automática del examen.

Clave:

NL = dígitos de la lección.
NT = número de temas por lección.
NP = número de preguntas por lección.
NQ = número de preguntas de que consta cada examen.
SUP, SUPE = superior
INFE = inferior.
I_1, I_2, I_3, ALEATO = números aleatorios para la subrutina del azar.
NTLEC (NL) = número total de ítems de una lección determinada que han sido preguntados en ese examen.
MAX = máximo.
NPTEM (NL, NT) = número de preguntas del ítem que han salido en el examen.

perforadas, con el que si bien se ahorra el paso de papel de marca sensible a tarjetas, el alumno una vez contestada una pregunta no tiene posibilidad de corrección. Además que, por ser un método poco usual de contestación (no es hacer una marca sencillamente, sino hacer saltar la perforación) puede representar una fuente de nerviosismo y errores por parte del alumno.

3.º Respuesta directa al ordenador a través de un terminal en dos vertientes:

a) El ordenador dispone de todas las preguntas que podríamos considerar básicas para una asignatura. Si el alumno no responde al 100 por 100 de las preguntas planteadas queda excluido (NO APTO) pudiendo, en caso contrario, pasar a la fase siguiente.

b) Como antes el ordenador dispone de todas las preguntas de la asignatura (básicas y no) y realiza una serie de N preguntas al azar. Si el alumno responde a un nivel mínimo (65 por 100, por ejemplo,) es APTO y no lo es en caso contrario.

Si quiere mejorar nota, en el caso de salir APTO, se pueden seguir dos caminos:

b-1) Aumentar el nivel mínimo (75 80 por 100 etc.) y seguir generando preguntas al azar.

b-2) Dejar el nivel mínimo en su valor, pero seleccionar las preguntas, aumentando su dificultad.

La lectura en papel de marca sensible es el procedimiento que nosotros hemos adoptado en razón fundamental de su economía.

Es evidentemente mucho menos caro que la presencia de suficiente número de terminales para poder examinar poblaciones estudiantiles que oscilan alrededor de los 500 alumnos por curso y, asimismo, menos caro que la adquisición de suficiente número de "*Cart a punch*", (coste aproximado 500 pesetas por unidad), para poder examinar en grupos reducidos. Asimismo la información que se puede almacenar en las tarjetas perforadas a mano es notablemente inferior (aproximadamente 1/4,5) de la información que puede albergar una tarjeta de 80 columnas en la que se han perforado los resultados inscritos en el papel de marca sensible (coste por hoja 60-80 céntimos). El proceso se indica en la figura 1 y comprende los siguientes pasos (figura 4):

1.º Lectura óptica del papel de marca sensible y perforación en tarjetas.

2.º Paso a cinta magnética.

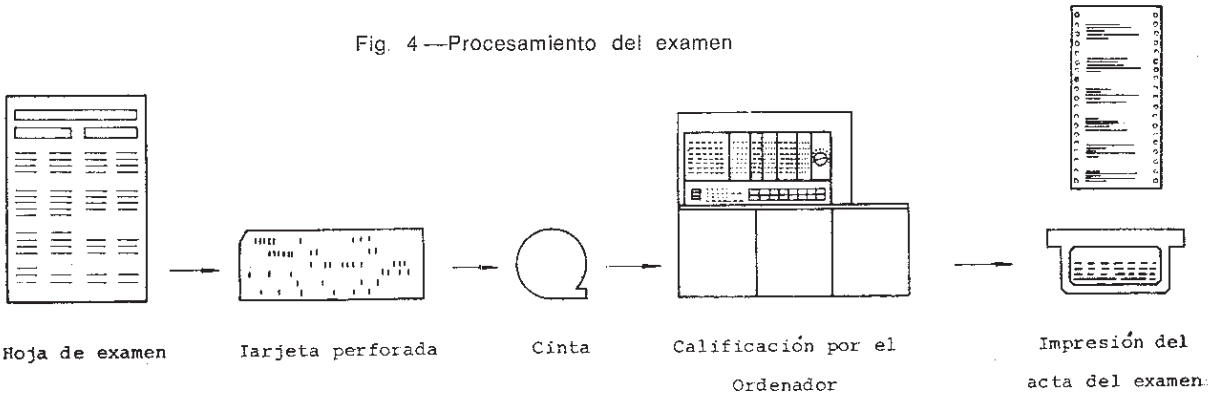
3.º Comparación con el ordenador de cada examen con el patrón.

4.º Salida del ordenador en forma de lista alfabética con la calificación de cada alumno.

El principal inconveniente del papel de marca sensible y que ha hecho que haya sido abandonado por algunos autores está representado por el hecho de que el sistema de perforación puede modificar el ejercicio del alumno. En otras palabras la tarjeta perforada (imagen) puede no ser igual al papel de marca sensible (objeto) realizado por el alumno. Las diferencias pueden provenir por una mala realización técnica del examen por parte del alumno (marcas realizadas fuera de sitio, marca de insuficiente densidad, correcciones del ejercicio no realizadas correctamente) o por parte de la máquina en forma de errores sistemáticos o accidentales.

Los errores accidentales pueden ser detectados por la reiteración del proceso en la misma máquina y comparación de las tarjetas obtenidas. Ahora bien, los errores sistemáticos no podrán ser detectados por este procedimiento, puesto que por muchas veces que repitamos la operación siempre se produciría el fallo al mismo nivel.

Fig. 4.—Procesamiento del examen



Cada máquina lectora tiene una sensibilidad de lectura característica que, evidentemente, no puede ser ajustada a la densidad de inscripción de cada examen; al ajustar un valor de la sensibilidad nos podemos encontrar con los dos casos extremos de sensibilidad insuficiente —que no recogerá respuestas realizadas por el alumno— —o sensibilidad excesiva— que recogerá respuestas borradas o anuladas por el alumno o podrá dar como respuestas pequeñas manchas o impurezas del papel. En ambos casos la tarjeta perforada no se corresponderá exactamente con el ejemplar realizado por el alumno en el papel de marca sensible.

Un procedimiento para obviar este problema es la perforación de dos tarjetas en dos máquinas diferentes y su comparación ulterior, resolviéndose simultáneamente los errores accidentales y sistemáticos.

Este es el procedimiento que realizamos en la actualidad (fig. 5) en el que aquellas tarjetas que muestran una diferencia de perforación son analizadas individualmente y corregidas con el ejercicio realizado por el alumno.

Las diferencias son corregidas manualmente y de esta forma las tarjetas son el retrato fiel y absoluto del ejercicio del alumno.

La información contenida en las tarjetas perforadas es pasada a cinta magnética y procesada por el ordenador.

Cada ejercicio es identificado por el número de matrícula del alumno, existiendo previamente en la memoria del ordenador un fichero de alumnos con la equivalencia, que contiene además del número de matrícula, nombre y apellidos, una serie de datos que nos serán útiles para los estudios ulteriores: edad, sexo, estado civil, religión, distrito universitario de origen, enseñanza primaria y secundaria recibida, etc.,

La calificación individual de cada examen no precisa de grandes ordenadores; es más, incluso puede ser realizada manualmente por simple superposición del examen de cada alumno con un examen patrón, realizando el recuento de las coincidencias. Este método a pesar de ser extremadamente simple puede introducir errores (no superiores, en general, al 2-3 por 100 (4)) debidos a la fatiga visual de quien corrige la prueba cuando el número de ejercicios a corregir es elevado.

La misma máquina lectora (IBM 1230 Optical Mark Scoring Reader) puede automáticamente identificar el resultado inscribiendo sobre el ejercicio el número total de respuestas que coinciden con las marcas de los ejercicios patrón. Previamente se ha corregido el ejercicio patrón comparándolo consigo mismo y se ha verificado que se obtiene el 100 por 100 de respuestas correctas.

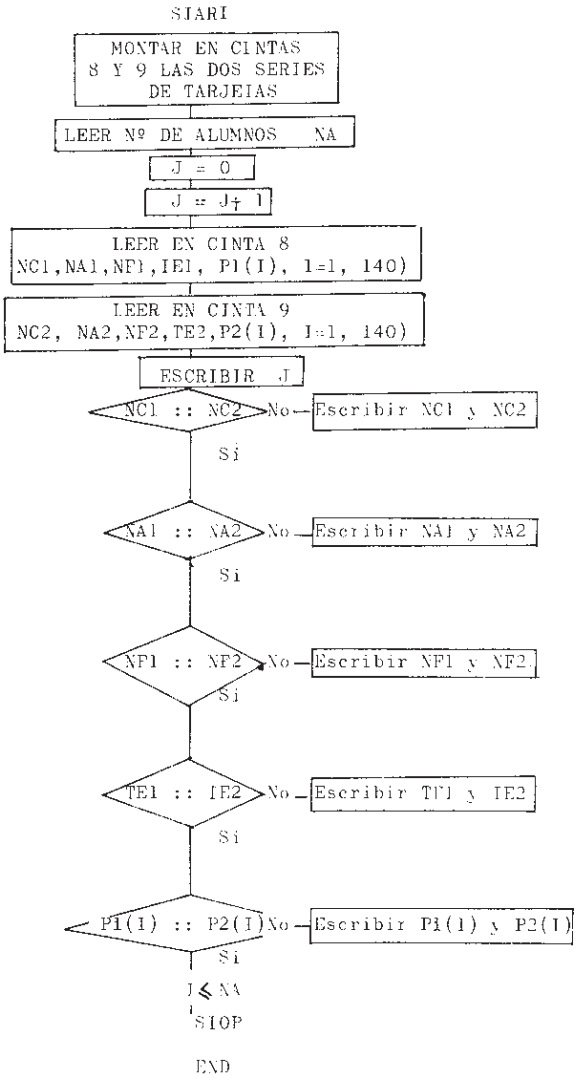


Fig 5.—Organigrama de la comparación de la doble perforación a partir de cada examen

Clave

NC = códico de la Cátedra
NA = número de alumno
NF = número del formulario de examen
TE = tiempo empleado
P(I) = número identificativo de cada pregunta
NC_1, NA_1, NF_1 TE_1, $P(I)_1$ = variables en la cinta 1
NC_2 NA_2 NF_2, TE_2, $P(I)_2$ = variables en la cinta 2

3.ª Fase: *Calificación colectiva de los exámenes y posibilidades de estudio derivadas de la calificación colectiva.*

La calificación de un gran número de exámenes puede ser realizada, como hemos visto antes, manualmente. Sin embargo, el tratamiento de toda la información encerrada en el examen y su valoración estadística posterior, precisa programas mucho más complejos que no pueden ser resueltos por una máquina sencilla (fig. 6).

Los problemas que se pueden resolver son, entre otros, los siguientes:

1.º *Elaboración automática del acta de examen,* que nos traducirá la calificación individual de cada examen con arreglo a criterios previos que introduciremos sobre la jerarquización y puntuación de los diferentes niveles de conocimiento (apto/no apto, no apto/aprobado-notable, sobresaliente, puntuación total, etc.) (fig. 7).

2.º *Valoración colectiva de la población estudiantil examinada:* Podemos obtener una imagen global del nivel de conocimientos de los alumnos realizando el histograma de número de preguntas correctamente contestadas y porcentaje de alumnos (o número absoluto) en cada valor de aciertos. Se obtiene, en general, curvas en forma de campana, en

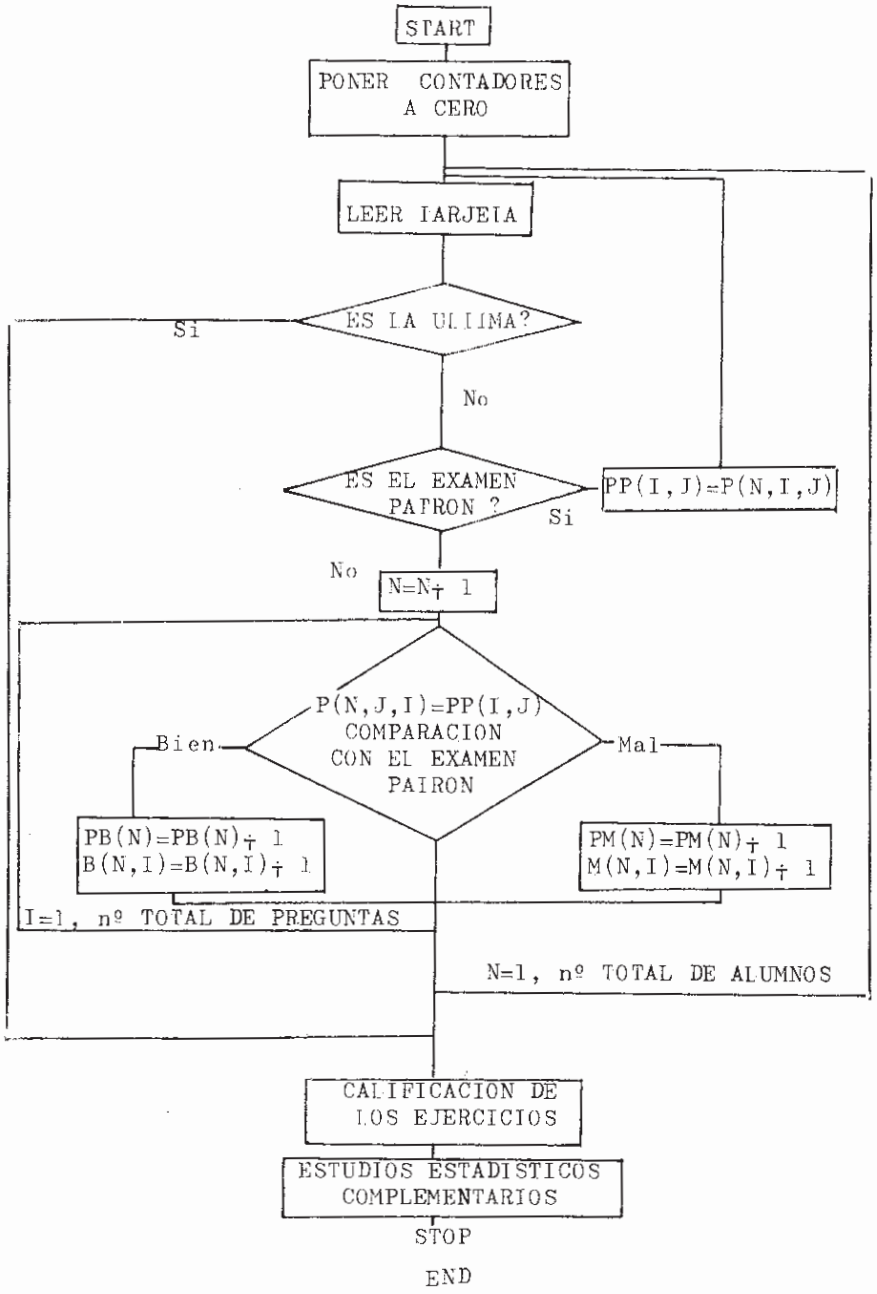


Fig 6—Organigrama de la corrección automática.

FORMULARIO NUMERO 60

NO ALUMNO	NOMBRE		PREG.BIEN	TIEMPO
2250	GUGLIERI BRAVO	CARLOS	30	45
3494	GUTIERREZ LOPEZ-VAZQUEZ	FRANCISCO	30	25
2449	GUTIERREZ RODRIGUEZ	TRINIDAD	24	40
2626	HERNANDEZ S.RICO	MERCEDES	28	35
2389	HERNANZ BENITO	LUIS	20	30
1975	HERRANZ CASADO	MIGUEL ANG	34	45
2314	HERRERO GALERA	EMILIO JES	29	30
1493	HERRERO PIQUERAS	M LUISA	31	35
2384	HIDALGO CHERCOLES	RAMON	35	40
2726	IGLESIA BRUN	FEDERICO D	26	45
9060	IPARRAGUIRRE URANGA	M TERESA	30	40
2406	GARCIA-MORATO JARRETO	VICENTE	15	45
2564	JIMENEZ HIDALGO	M GLORIA	23	35
3753	JIMENEZ LOPEZ	CARLOS ENR	30	45
2779	JUAN URIEN	JOSE ALFRE	32	40
2821	KOLYBABIUK	YAROSLAW J	35	40
1077	LADRERO ALBARRAN	MARIA ELEN	31	40
3817	LANDA GARCIA	JOSE IGNAC	22	40
2650	LAZARO LAZARO	PABLO	30	40
2206	LAZARO Y DE MERCADO	PABLO	35	40
2477	LEYVA COBIAN	FRANCISCO	27	45
2020	LITON MUNOZ	MIGUEL	18	35
1692	LOPEZ BARTOLOZZI	LUIS FERNA	29	37
2824	LOPEZ BURILLO	ANA MARIA	28	45
2300	LOPEZ CID	JOSE ANDRE	25	45
3554	LOPEZ PALOMINO	JORGE	23	40
3019	LOPEZ CAMARA CRESPO	MARIA FRAN	23	43
3026	LORENTE MARTINEZ	DARIO	22	35
1492	LOZANO ALVAREZ	MATILDE	28	38
EL ALUMNO NUMERO 3802	1 NO ESTA EN LA LISTA			
3802	LOZANO ALVAREZ	MATILDE	28	43
3161	MARFIL MONTES	MARIA TERE	23	43
2131	MARIN RODRIGUEZ	ALEJANDRO	31	34
3060	MARTIN GARCIA	FERNANDO	19	40
3195	MARTIN-FORERO RODRIGUEZ	ESPERANZA	29	35
2016	MARTIN-SERRANO VELASCO	JESUALDO P	27	38
2175	MARTINEZ MARTINEZ	MARIA ISAB	26	37
2099	MASEGOSA DE JESUS	SALVADOR	32	40
3127	MATEO MARTIN	ILDEFONSO	29	40
1391	MEDINA CHAMORRO	CRISTOBAL	30	37
2209	MENDOZA FERNANDEZ	MARIA LUIS	23	34
3281	MIRANDA ARROYO	JOSE LUIS	33	35
2129	MOLINERO ATIENZA	M DOLORES	33	44
2401	MONTES GONZALEZ	LUIS MANUE	21	44
259	MORALES GUTIERREZ	CARLOS LUI	33	38
9081	MOREIRAS JIMENEZ	M DEL PILA	28	40
1994	MORENO MORENO	M ANGELES	23	30
3219	MORENO TELLO	BENIGNO	22	44
2655	MOYA CAMPOS	PEDRO	25	35

Fig. 7—Acta del examen

las que el desplazamiento hacia la derecha del vértice indica un mejor aprovechamiento general (fig. 8).

3.º *Valoración de la "dificultad de cada pregunta"* determinando el porcentaje de alumnos que han contestado correctamente a cada pregunta (fig. 9). La reiteración de este estudio cuando una misma pregunta se plantea en diferentes exámenes nos proporcionará la "historia" de la pregunta, lo que nos podrá servir también para valorar cómo ha sido explicada la pregunta a lo largo de cursos sucesivos.

Esto también podría servirnos de *feedback* asignando un peso a cada pregunta dependiente de su dificultad, lo que podría anular la valoración de cada pregunta que no hubiera sido contestada por nadie o que hubiera sido contestada por todos, puesto que en ambos casos no sería significativa, tanto por inaccesible al alumno o por demasiado elemental. La función dificultad/peso y su aplicación es un camino nuevo de trabajo que todavía no hemos desarrollado.

4.º *Estudio del "índice de fiabilidad"* o "capacidad de discernimiento" de una pregunta, esto es, estudiando si los aciertos ordenados en cada pregunta corresponden a los alumnos que han aprobado el examen o si, por el contrario, los aciertos han sido debidos al azar.

5.º *Determinación del tiempo óptimo del examen:* Es evidente el hecho señalado en otro trabajo (3) de que el factor tiempo puede ser muy importante en este tipo de exámenes. La representación de la calificación obtenida en función del tiempo invertido nos proporcionará una información sobre el tiempo óptimo de realización del examen (en el que el rendimiento del alumno es mayor). Los primeros resultados muestran una distribución en campana de esta función pendiente de comprobación con experiencias más amplias.

6.º Suponiendo que el alumno responda secuencialmente a las preguntas, se puede realizar el estudio de la *fatiga del alumno;* tiene como fundamento este estudio determinar el número máximo de preguntas o el tiempo máximo de examen, en el cual el

alumno se encuentra dentro de los límites de rendimiento máximo. Para ello, se puede estudiar el número de aciertos en las diferentes partes del examen, siendo este estudio relativamente complejo, puesto que únicamente tenemos un dato (porcentaje de alumnos que han contestado a una pregunta) para la valoración de dos magnitudes diferentes: dificultad de la pregunta y posible cansancio del alumno si la pregunta está situada en la parte final de un examen largo o con un tiempo escaso (fig. 10).

7.º Es interesante el estudio de *factores sociológicos* determinantes del rendimiento del alumno en un examen, como pueden ser:

1. Edad (fig. 11).
2. Sexo (fig. 12).
3. Estado civil.
4. Formación académica preuniversitaria (rendimiento en 1.ª, 2.ª enseñanza).

El aumento evidente de la población femenina en la Facultad de Medicina, el incremento progresivo de estudiantes casados incluso en edad temprana, la progresiva elevación de la edad media de la población estudiantil (debido por una parte a las dificultades de los estudios previos o cursos selectivos, etc.), el incremento de estudiantes que se incorporan a la Universidad después de haber solucionado con otros estudios su situación económica, son factores que indican el interés de estos estudios.

RESUMEN

1. La corrección por ordenador de exámenes objetivos por *test* de respuestas de elección múltiple no introduce modificación alguna en la calificación.

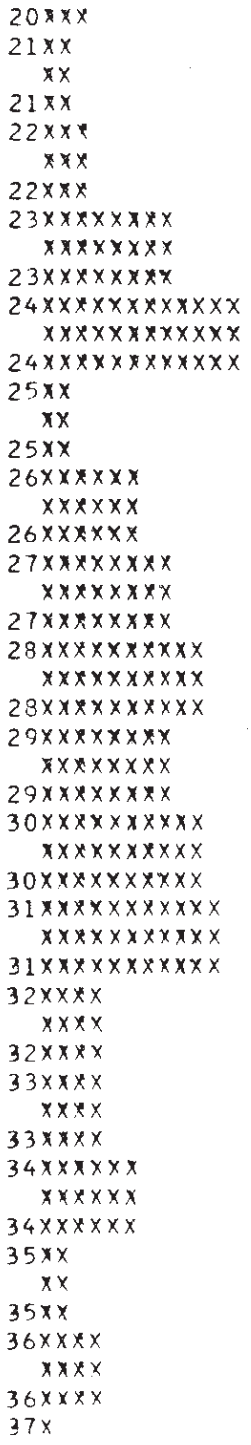


Fig 8.—Distribución de los aciertos En abcisas: número de preguntas contestadas (sobre un máximo de 50 preguntas) En ordenadas: valores proporcionales al por ciento de alumnos que contestaron correctamente

Fig. 9.—Valoración de la dificultad de cada pregunta

NO DE LA PREGUNTA	0/0 DE ALUMNOS QUE LA CONTESTAN BIEN	NO DE LA PREGUNTA	0/0 DE ALUMNOS QUE LA CONTESTAN BIEN	NO DE LA PREGUNTA	0/0 DE ALUMNOS QUE LA CONTESTAN BIEN
1	80.3	2	86.8	3	26.3
4	14.5	5	93.4	6	63.2
7	43.4	8	46.1	9	61.8
10	30.3	11	46.1	12	61.8
13	40.8	14	17.1	15	0.
16	64.5	17	93.4	18	90.8
19	75.0	20	22.4	21	40.8
22	73.7	23	43.4	24	76.3
25	89.5	26	39.5	27	59.2
28	55.3	29	64.5	30	48.7
31	59.2	32	65.8	33	51.3
34	36.8	35	93.4	36	43.4
37	56.6	38	90.8	39	18.4
40	34.2	41	48.7	42	81.6
43	61.8	44	15.8	45	23.7
46	36.8	47	43.4	48	76.3
49	30.3	50	48.7		

FORMULARIO NUMERO 60

	INTERVALO DE PREGUNTA (COMPRENDIDO ENTRE LAS PREGUNTAS)	TANTO POR CIENTO QUE RESPONDEN
1	1- 5	60.26
2	6- 10	48.95
3	11- 15	33.16
4	16- 20	69.21
5	21- 25	64.74
6	26- 30	53.42
7	31- 35	61.32
8	36- 40	48.68
9	41- 45	46.32
10	46- 50	47.11

Fig 10—Porcentaje de aciertos en las diferentes partes del examen

FORMULARIO NUMERO 60

EDAD DEL GRUPO	NO DE EXAMINADOS CON ESA EDAD	TANTO POR CIENTO CONTESTADO
18	0	0.
19	0	0.
20	16	56.62
21	19	52.84
22	16	54.00
23	11	52.00
24	6	50.33
25	4	50.00
26	1	46.00
27	1	44.00
28	0	0.
29	1	52.00
30	0	0.
31	0	0.
32	0	0.
33	0	0.
34	0	0.
35	1	62.00
36	0	0.
37	0	0.
38	0	0.
39	0	0.
40	0	0.

Fig 11—Porcentaje de aciertos en las diferentes edades

FORMULARIO	NO DE VARONES	TANTO POR CIENTO CONTESTADO	NO DE MUJERES	TANTO POR CIENTO CONTESTADO
60	50	53.04	26	53.85

Fig 12—Porcentaje de aciertos y sexo de los alumnos

2. En nuestra experiencia la mayor dificultad del procedimiento está en la transmisión de la marca sensible a la tarjeta perforada, resolviendo este problema por la doble perforación y comparación ulterior.

3. El ordenador puede resolver estudios estadísticos sobre la valoración colectiva e individual de las preguntas presentadas, así como del rendimiento de la población estudiantil en función de diferentes parámetros, e incluso existe la posibilidad de valoración de la docencia impartida.

REFERENCIAS

1. ANDERSON, J ; WOOK, H y TOMILSON, R W S.: Examination making by computer . Brit J. Med. Ed , 198, 2, 210-212, 1969.

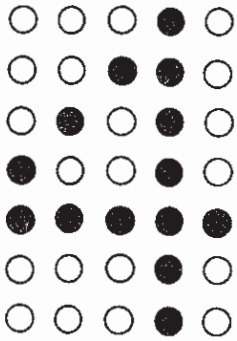
2. GASTINEL, N : 'Correction automatique de test a l Universitè de Grenoble . Chiffres, 3/4, 1960

3. GIL GAYARE, M y GIL GAYARRE,C. Arch. Fac Med Madrid 16/3, septiembre 1969

4. GIL GAYARRE, M; RODRIGUEZ, J. A.; RODRIGUEZ, L P y GIL GAYARRE, C : "Comparación de métodos objetivos y clásicos de examen' (Datos no publicados, 1970)

5 HUBBARD J P. y CLEMANS, W. V.: "Múltiple-choice examinations in medicine . Lea & Febiger, Philadelphia, 1961.

6 SANCHEZ de ZAVALA, V.: 'Seminario de valoración del Aprendizaje". C.C.U.M., 1969.



3

TRADUCTOR 70/13

Por

Irene Fernández Flórez
Centro de Cálculo de la Universidad
de Madrid

Isidro Ramos Salavert
Centro de Cálculo de la Universidad
de Madrid

Fernández Flores, I. Ramos I., Traductor 70/13. Boletín CCUM. Volumen 10. 1970.

TRADUCTOR 70/13

Por I. Fernández Flórez e I. Ramos

Sección I - Generalidades

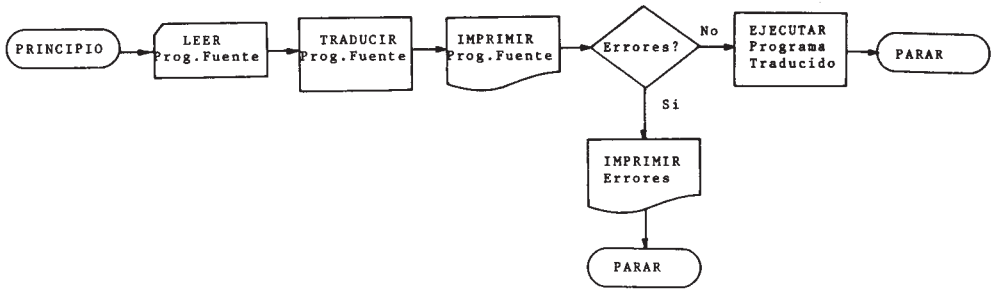
El Traductor 70/13 es un programa escrito en FORTRAN IV y MAP que simula un ordenador ideal en la IBM 7090 del CCUM. Este ordenador ideal consta de una Memoria de 100 palabras direccionables, un Contador de Dirección y un Registro Acumulador donde se realizan todas las operaciones aritméticas; como Unidad de Entrada posee una lectora de fichas perforadas y como Unidad de Salida una impresora de líneas; ambos dispositivos están conectados on-line con la 7090.

Su lenguaje fuente tiene las siguientes características generales: es un lenguaje de nivel máquina que trabaja con direcciones absolutas, está escrito en castellano y posee casi todas las operaciones fundamentales de un ordenador digital real. Las instrucciones posibles son:

LEER EN LA POSICION (XX) EL CONTENIDO DE LA UNIDAD DE ENTRADA.
DEPOSITAR EN LA POSICION (XX) EL CONTENIDO DEL ACUMULADOR.
ESCRIBIR EL CONTENIDO DE LA POSICION (XX) EN LA UNIDAD DE SALIDA.
TRASLADAR EL CONTENIDO DE LA POSICION (XX) AL ACUMULADOR.
TRASLADAR AL ACUMULADOR EL NUMERO = XXXXXX.
SUMAR AL ACUMULADOR EL CONTENIDO DE LA POSICION (XX).
SUMAR AL ACUMULADOR EL NUMERO = XXXXXX.
RESTAR DEL ACUMULADOR EL CONTENIDO DE LA POSICION (XX).
RESTAR DEL ACUMULADOR EL NUMERO = XXXXXX.
MULTIPLICAR EL ACUMULADOR POR EL CONTENIDO DE LA POSICION (XX).
MULTIPLICAR EL ACUMULADOR POR EL NUMERO = XXXXXX.
DIVIDIR EL ACUMULADOR POR EL CONTENIDO DE LA POSICION (XX).
DIVIDIR EL ACUMULADOR POR EL NUMERO = XXXXXX.
HALLAR LA RAIZ CUADRADA DEL ACUMULADOR.
PONER EN EL CD LA DIRECCION (XX).
SI EL ACUMULADOR ES MAYOR QUE CERO PONER (XX) EN EL CD.
SI LA POSICION (XX) ES MENOR QUE EL ACUMULADOR PONER (YY) EN EL CD.
SI LA POSICION (XX) ES IGUAL QUE EL ACUMULADOR PONER (YY) EN EL CD.
SI LA POSICION (XX) ES MAYOR QUE EL ACUMULADOR PONER (YY) EN EL CD.
HACER MM VECES LAS NN INSTRUCCIONES SIGUIENTES.
PARAR
FIN
DATOS
FINAL

Estas instrucciones se traducen a sentencias escritas en Fortran IV en una correspondencia una a una; este programa Fortran, escrito por el Traductor en una cinta magnética, es ejecutado bajo control del Sistema Operativo IBSYS y da los resultados del problema en la impresora on-line de la 7090.

El organigrama general del Programa Traductor es el siguiente:



Como puede verse en la Fig. 1, las Fases principales del programa son:

1.- Lectura de las Instrucciones Fuente. Estas instrucciones van perforadas una en cada tarjeta, empezando siempre en columna 1. El número entre paréntesis puede variar de 1 a 99, indicando una de las posibles direcciones de la Memoria de la máquina; en caso de que tenga una sola cifra, se puede poner (0X) o simplemente (X), donde X varía de 1 a 9. El número que puede aparecer a la derecha del signo = puede ser entero o real con un máximo de 6 cifras (incluyendo el punto si es real).

En el caso particular de la instrucción HACER, se puede omitir el segundo número y entonces se supone que vale 1; es decir, se puede redactar así:

HACER MM VECES LA INSTRUCCION SIGUIENTE

La abreviatura CD significa Contador de Dirección, es decir, se refiere al registro que contiene la dirección de la instrucción que se está ejecutando.

La última instrucción de cada programa debe ser la instrucción FIN y debe tener al menos una instrucción PARAR.

Los datos fuente deben ir precedidos de la instrucción DATOS e irán perforados en las 6 primeras columnas de cada ficha, a razón de uno por tarjeta; pueden ser números enteros o reales, los primeros sin punto, los segundos con él, en cualquier posición de las 6 previstas para ello.

Después del último dato se pondrá una instrucción FINAL, que marca el fin físico del paquete de fichas.

2.- Traducción del Programa Fuente. Cada una de las instrucciones fuente da lugar a una sentencia Fortran IV equivalente. Las 6 primeras letras de la instrucción fuente constituyen el código de operación y especifican la sentencia Fortran correspondiente. Las direcciones entre paréntesis y los literales que van detrás de un signo = son la parte variable de la instrucción y son introducidas en la parte variable de la sentencia Fortran equivalente. Todas las sentencias Fortran generadas llevan número de sentencia con objeto de facilitar las transferencias de control dentro del programa.

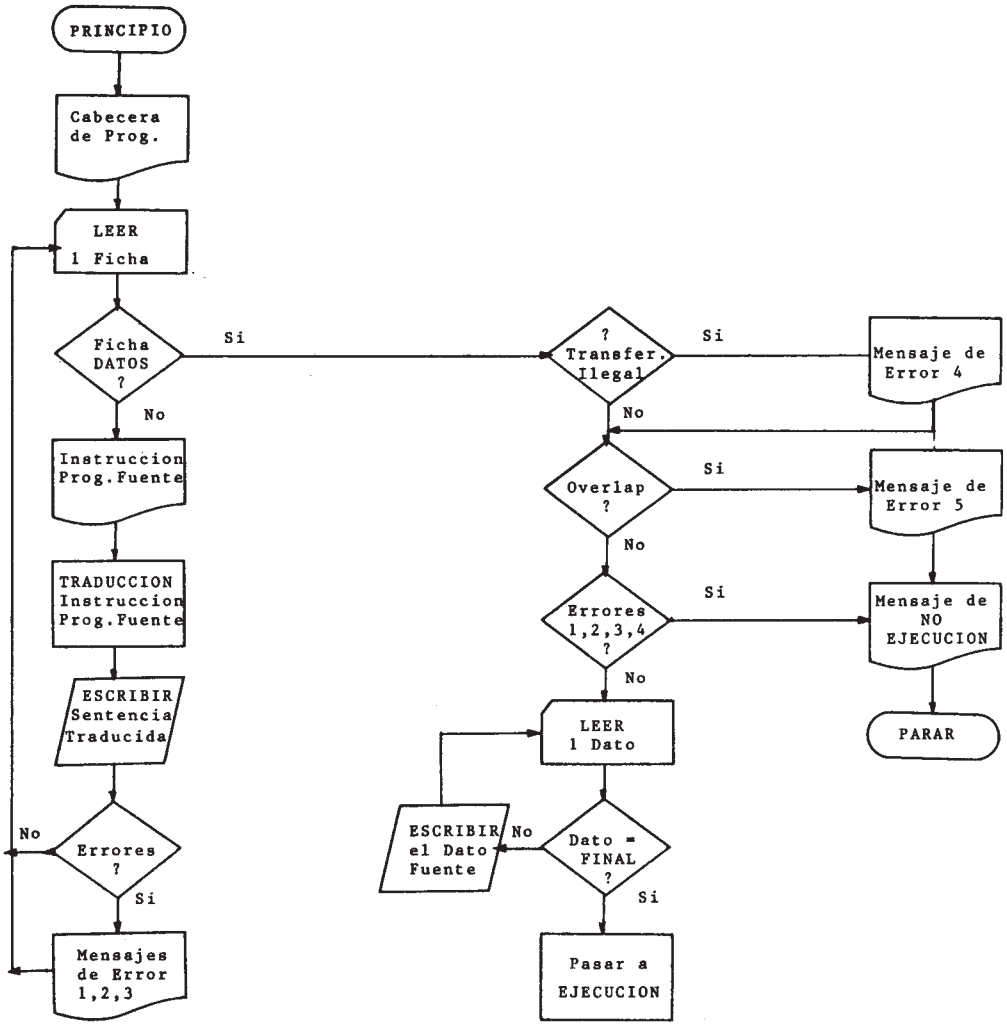
3.- Impresión o listado del Programa Fuente. En todos los casos se imprime el programa fuente tal como fue escrito por el programador, precedido de unas cabeceras o títulos generales (Véase el Ejemplo del final).

4.- Búsqueda de Errores en el Programa Fuente. En el proceso de traducción está prevista la detección de algunos errores sintácticos (Véase colección de mensajes de error). Esta colección se irá ampliando a medida que la experiencia demuestre la necesidad de nuevos mensajes de error. Si se detecta algún error de los previstos en el programa traductor, se imprime el mensaje correspondiente debajo de la instrucción errónea.

5.- Ejecución del Programa Traducido. Si no se han detectado errores, se ejecuta el programa Fortran y se imprimen los resultados.

Sección II

En la Fig. 2 puede verse un organigrama algo más detallado que el anterior, que examinaremos con más detenimiento.


PRINCIPIO
Cabecera de Prog.
LEER 1 Ficha
Ficha DATOS ?
Si
No
Instruccion Prog.Fuente
TRADUCCION Instruccion Prog.Fuente
ESCRIBIR Sentencia Traducida
Errores ?
No
Si
Mensajes de Error 1,2,3
? Transfer. Ilegal
Si
Mensaje de Error 4
No
Overlap ?
Si
Mensaje de Error 5
No
Errores 1,2,3,4 ?
Si
Mensaje de NO EJECUCION
No
PARAR
LEER 1 Dato
Dato = FINAL ?
No
ESCRIBIR el Dato Fuente
Si
Pasar a EJECUCION

Primero se escriben en la impresora unas cabeceras como las del Ejemplo final; conviene a nuestro juicio suprimir en la impresión cualquier mensaje del Sistema Operativo, así como la impresión de las fichas de control de éste para que sea más efectivo desde el punto de vista didáctico, ya que se pretende que la simulación de la máquina ideal sea lo más completa posible.

A continuación se lee una ficha instrucción del programa fuente; se pregunta si es la ficha DATOS, en cuyo caso ya se ha leido todo el programa. Si no es así, se imprime la instrucción fuente y se pasa a la fase de Traducción que, en nuestro caso, efectúa una subrutina escrita en lenguaje simbólico MAP, cuyo funcionamiento se describirá en la Sección III. Una vez traducida, esta sentencia se escribe en algún dispositivo de Salida (perforadora, cinta magnética, etc.) para que pueda ser procesada después. Si en la instrucción se ha detectado algún error, se imprime el mensaje correspondiente y se pasa a leer otra instrucción del programa fuente. Si no hubiese errores, se pasa directamente a leer otra ficha.

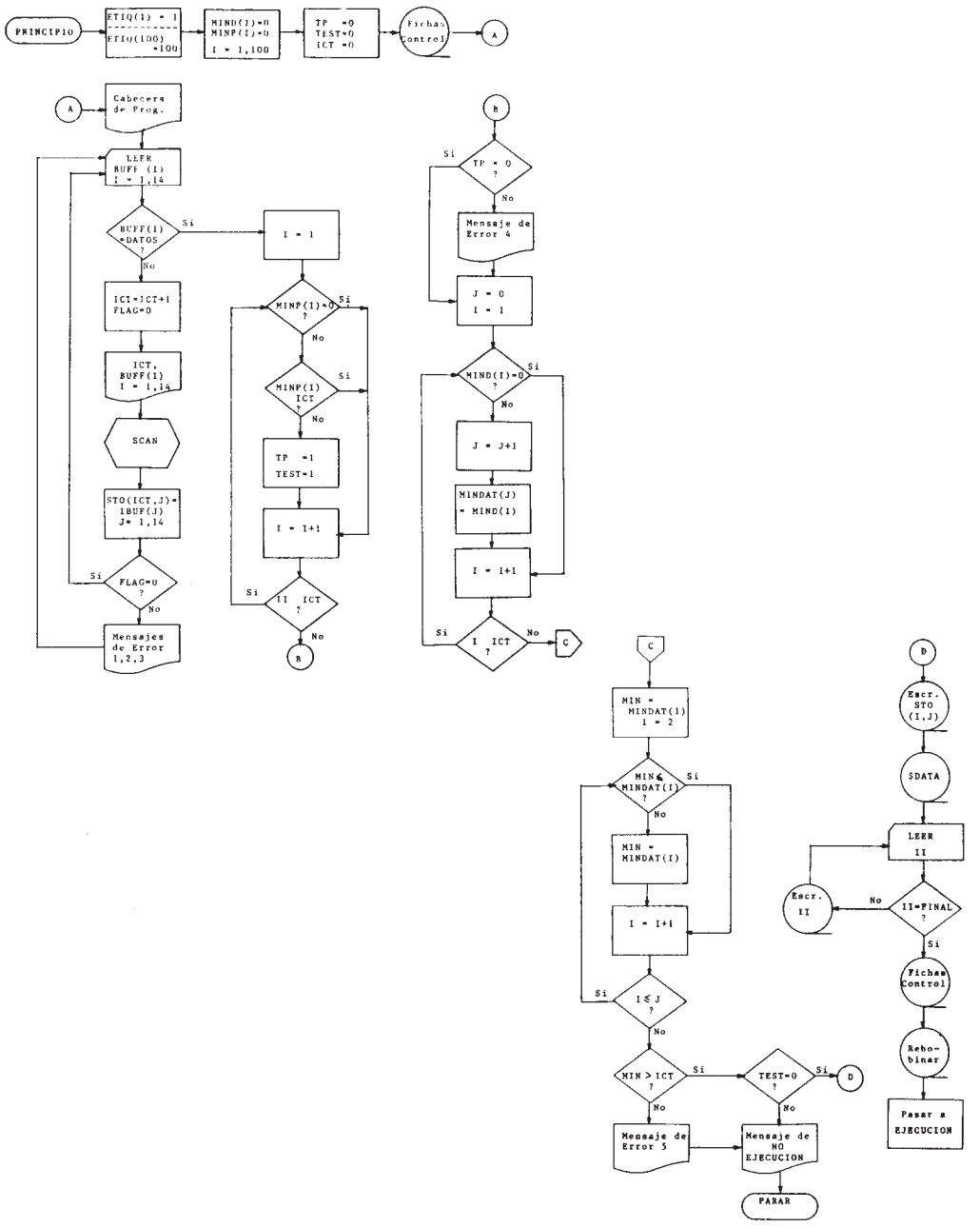
Cuando se lee la instrucción DATOS, se investiga si ha habido alguna transferencia de control ilegal, es decir, si el contador de dirección ha alcanzado en algún momento un valor superior al correspondiente a la última instrucción del programa. Si ha sido así, se imprime un mensaje de error; en cualquiera de los dos casos se pregunta si se ha solapado el programa con el área de datos, es decir, si en algún momento se ha depositado un dato o resultado en una posición de memoria correspondiente al programa. Si se ha producido este hecho, se imprime un mensaje de error y un mensaje de NO EJECUCION y se detiene el proceso; si no ha habido solapamiento, se investiga si se ha detectado algún error en el proceso de traducción; si lo ha habido, se imprime el mensaje de NO EJECUCION y se para y, si no ha habido ningún error, en el programa, se pasa a la fase de lectura de los datos fuente.

Estos datos se leen y se escriben después en un dispositivo de Salida del ordenador (el mismo que se haya empleado para escribir las instrucciones traducidas) hasta que se alcance la instrucción FINAL que marca el fin físico del paquete fuente.

Una vez que se ha llegado a FINAL, se pasa a ejecutar el programa traducido, que imprimirá en la impresora los resultados del problema planteado.

Sección III

En esta sección vamos a describir el programa traductor tal como funciona en la IBM 7090 del CCUM. Adjuntamos el organigrama detallado y el listado del programa, así como la colección de mensajes de error.


PRINCIPIO
ETIQ(1) = 1
ETIQ(100) =100
MIND(I)=0
MINP(I)=0
I = 1,100
TP =0
TEST=0
ICT =0
Fichas Control
A
Cabecera de Prog.
LEER BUFF (I) I = 1,14
BUFF(1) =DATOS ?
Si
No
I = 1
ICT=ICT+1
FLAG=0
ICT, BUFF(1) I = 1,14
SCAN
STO(ICT,J)= IBUF(J) J= 1,14
FLAG=0 ?
Mensajes de Error 1,2,3
MINP(I)=0 ?
MINP(I) ICT ?
TP =1
TEST=1
I = I+1
II ICT ?
B
TP = 0 ?
Mensaje de Error 4
J = 0
I = 1
MIND(I)=0 ?
J = J+1
MINDAT(J) = MIND(I)
I = I+1
I ICT ?
C
MIN = MINDAT(1) I = 2
MIN≤ MINDAT(I) ?
MIN = MINDAT(I)
I = I+1
I≤J ?
MIN > ICT ?
TEST=0 ?
D
Mensaje de Error 5
Mensaje de NO EJECUCION
PARAR
Escr. STO (I,J)
$DATA
LEER II
II=FINAL ?
Escr. II
Fichas Control
Rebobinar
Pasar a EJECUCION

La variable ETIQ es una variable alfanumérica cuyos valores van de 1 a 100, los cuales serán las etiquetas o números de sentencia de las sentencias Fortran generadas.

Los 100 valores de la variable MIND se ponen inicialmente a cero, junto con los 100 valores de MINP. En MIND se almacenarán más tarde las direcciones de datos empleadas por el programa fuente y en MINP las direcciones de programa (valor del contador de dirección) empleadas en las instrucciones de transferencia de control.

Las variables TP, TEST e ICT se ponen inicialmente a cero; el valor de TP indicará si se ha producido una transferencia de control ilegal; el valor de TEST al final de la traducción de todas las instrucciones fuente, nos dirá si ha habido algún error de cualquier tipo de los previstos por el programa e ICT marcará el orden de las instrucciones fuente.

A continuación se escriben en una cinta magnética las tarjetas de control necesarias para poder pasar el programa traducido a Fortran IV en la 7090 del CCUM ($JOB, $EXECUTE IBJOB, $IBJOB e $IBFTC).

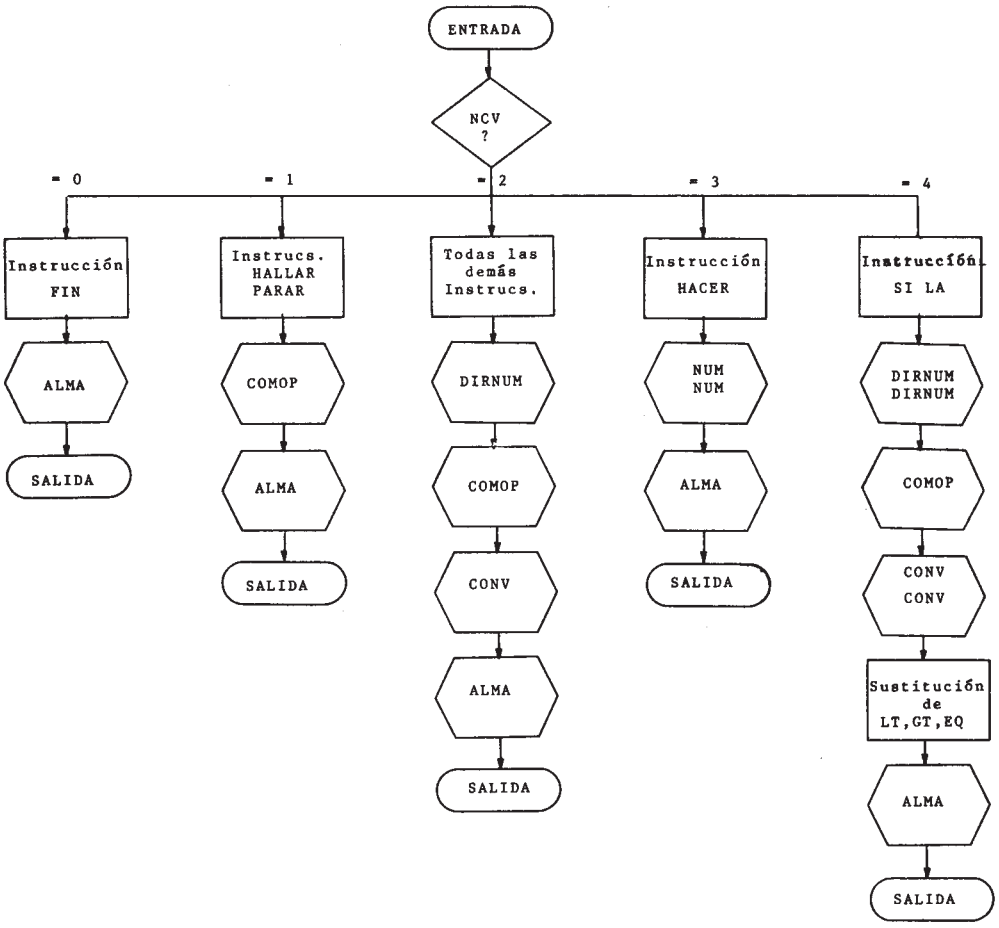
Las cabeceras de programa impresas on-line pueden verse en el Ejemplo del final.

La variable BUFF contiene la instrucción fuente, es decir, el contenido de una de las tarjetas del paquete fuente. La variable FLAG se pone a cero cada vez que se lee una instrucción fuente y sirve para detectar, según su valor, el tipo de error encontrado en el proceso de traducción.

Subrutina SCAN

Esta subrutina es la encargada del proceso de traducción y del análisis sintáctico de las instrucciones fuente. Sus argumentos de Entrada son las variables BUFF y ETIQ correspondientes a la instrucción que va a ser traducida. Sus argumentos de Salida son IBUF (sentencia Fortran equivalente), FLAG (señalizador de error) y los valores de MIND y MINP correspondientes.

Su organigrama es el siguiente:


ENTRADA
NCV ?
= 0
= 1
= 2
= 3
= 4
Instrucción FIN
ALMA
SALIDA
Instrucs. HALLAR PARAR
COMOP
ALMA
SALIDA
Todas las demás Instrucs.
DIRNUM
COMOP
CONV
ALMA
SALIDA
Instrucción HACER
NUM NUM
ALMA
SALIDA
Instrucción SI LA
DIRNUM DIRNUM
COMOP
CONV CONV
Sustitución de LT,GT,EQ
ALMA
SALIDA

Al entrar en la subrutina lo primero que hace es comparar las primeras 6 letras de la instrucción que está en BUFF con los elementos de la Tabla de Operaciones incorporados en la propia subrutina; si no encuentra concordancia con ninguno de ellos, hace FLAG=1 y devuelve control al programa principal.

El proceso de traducción depende esencialmente del número de campos variables que tenga la sentencia Fortran correspondiente (variable NCV del organigrama). Veamos lo que hacen las macro-operaciones representadas en el organigrama mediante exágonos.

Macro-operación ALMA

Esta macro lo que hace es almacenar en IBUF la sentencia Fortran generada.

Macro-operación COMOP

Esta macro sustituye el valor de ETIQ en las dos primeras posiciones de la sentencia Fortran.

Macro-operación DIRNUM

Esta macro hace las siguientes funciones:

1) Busca un paréntesis abierto o un signo = en la instrucción fuente (BUFF). Si no lo encuentra, hace FLAG=2 y devuelve control al programa principal. Si encuentra un paréntesis abierto almacena el número que va detrás (que es una dirección absoluta) en el campo variable correspondiente de la sentencia Fortran. Si encuentra un signo =, almacena el número que va detrás (que es un valor entero o real) en el campo variable de la forma alternativa de la sentencia Fortran.

2) Además de lo anterior, esta macro guarda en una posición de almacenamiento temporal el valor alfanumérico de la dirección utilizada en la instrucción fuente, posición que más tarde usará la macro CONV.

Macro-operación CONV

Esta macro transforma el valor alfanumérico de la dirección utilizada en la instrucción fuente (almacenado en una palabra por DIRNUM) en su valor numérico equivalente. Este valor queda almacenado en MIND o MINP según que sea una dirección-dato o una dirección que afecta al contador de dirección.

Macro-operación NUM

Esta macro es muy parecida a DIRNUM y se emplea exclusivamente para la instrucción HACER; ahora lo que busca es un campo numérico en la instrucción fuente y lo sustituye en la sentencia Fortran (el número de veces que se va a repetir el ciclo); la segunda vez que es llamada esta macro, se busca el segundo campo numérico de la instrucción; si lo encuentra, le suma este valor a la propia dirección de la instrucción para obtener el número de sentencia de la última sentencia del rango del ciclo. Si no lo encuentra, supone que es la unidad.

Sustitución de LT, GT y EQ

La instrucción SI LA da lugar a cuatro campos variables: el número de sentencia (obtenido por COMOP), la dirección de la posición de memoria que se compara con el acumulador (obtenica por DIRNUM), el valor que hay que poner en el contador de dirección en el caso de que se cumpla la condición impuesta (obtenido en una segunda llamada a DIRNUM) y el operador de relación que liga el valor del acumulador y el valor de la posición de memoria. Las palabras clave para la determinación del operador de relación son: MAYOR, MENOR e IGUAL que dan lugar a los operadores GT, LT y EQ en la sentencia Fortran equivalente.

Una vez que la subrutina SCAN ha devuelto control al programa principal, se almacena en la ICT-ésima fila de la variable STO la sentencia Fortran generada (IBUF). Si FLAG = 0 no ha habido errores y se pasa a leer otra ficha; si FLAG es distinto de cero, es que ha habido algún error en la instrucción fuente, se imprime el mensaje de error correspondiente y se pasa a leer otra ficha.

Cuando se lee la instrucción DATOS, se pasa a la fase de comprobar si ha habido alguna transferencia ilegal en el programa; para ello se eliminan todos los MINP = 0 (correspondientes a las instrucciones que no usan direcciones que afecten al contador de dirección) y con los restantes se comprueba que son menores que el número total de instrucciones del programa (contando la instrucción FIN). Si se encuentra alguna transferencia ilegal, hace TP = 1 y TEST = 1 e imprime un mensaje de error.

Después se comprueba si se ha depositado algún dato o resultado en alguna posición correspondiente al programa. Para ello se eliminan todos los MIND = 0 (correspondientes a las instrucciones que no usan direcciones del área de datos) y se meten en MINDAT todos los MIND distintos de cero. Una vez hecho esto, se obtiene el valor mínimo de MINDAT que se coloca en MIN y se pregunta si este valor es mayor que el número de instrucciones del programa. Si es menor o igual, se imprime un mensaje de error, el mensaje de NO EJECUCIÓN, y se detiene el proceso.

Si no ha habido solapamiento, se pregunta por el valor de TEST; si es distinto de cero es que ha habido algún error y se detiene el proceso después de imprimir el mensaje de NO EJECUCION.

Si TEST = 0, se escribe en la cinta magnética la matriz STO que contiene las sentencias Fortran generadas; luego se escribe la ficha de control $DATA.

A continuación se pasa a leer los datos fuente (variable II) que son escritos igualmente en la cinta.

Para cerrar el proceso se escriben en la cinta las fichas de control terminales ($EOF, $IBSYS, $JOB, $STOP), se rebobina la cinta y se pasa a ejecución el programa Fortran contenido en ella.

APLICACION DEL HASH-CODING AL TRATAMIENTO AUTOMATICO DE GRAMATICAS

Por Roberto Moya Quiles e Isidro Ramos Salavert

I. INTRODUCCION

En el tratamiento automático de la sintaxis de los lenguajes de programación, se presentan nuevos problemas que requieren para su resolución una tecnología nueva, por ahora no estructurada, que llena las revistas especializadas de algoritmos que resuelven determinados aspectos de un problema.

En el trabajo que desarrollamos conjuntamente sobre el diseño de un metaensamblador (y análogamente podríamos hablar del diseño de un metacompilador, compilador de compiladores, en definitiva, de todo sistema metaprogramado) la descripción formalizada (en el aspecto sintáctico) de los lenguajes que intervienen en el proceso la hacemos en el metalenguaje FNB (forma normal de Backus) codificada de forma que permita un tratamiento cómodo al ser tomada como dato por nuestro sistema.

El tratamiento ulterior que sufren estas descripciones sintácticas (paso a forma determinista, ...) hace que necesitemos extraer determinada información, que en nuestro caso podemos resumir en:

1°) Tablas de referencias cruzadas para metavariables y caracteres terminales.
2°) Terminales que encabezan cada metavariable.
3°) Metavariables que empiezan por un terminal dado.

Moya R., Ramos, I. Aplicación de Hash-Coding al tratamiento automático de Gramáticas". Boletín CCUM. nº 17. 1971

Exponemos en el punto II los algoritmos clásicos para la resolución de estos problemas basados en la teoría de relaciones, el punto III será la exposición de nuestro método y en el IV haremos una comparación de ambos métodos enumerando las ventajas e inconvenientes.

II. RELACIONES BINARIAS FUNDAMENTALES DEFINIDAS POR UNA GRAMATICA LIBRE DE CONTEXTO. APLICACIONES

Dada la gramática Tipo 2 $G = \{V_T, V_N, P, S\}$ podemos definir las siguientes relaciones binarias, impuestas por P, sobre los elementos de $V = V_T \cup V_N$ (conjunto finito) (1).

1) Dependencia simple (representada por R_S)

Diremos que $\Sigma \in V_N$ y $\alpha \in V$ cumplen: $\Sigma R_s \alpha$ si y solamente si α aparece en una parte derecha de la regla en la que Σ es la parte izquierda.

O sea:

$$\Sigma \rightarrow \sigma_1 \alpha \sigma_2 \quad , \quad \sigma_1 \in V^* \quad , \quad \sigma_2 \in V^*$$

En el grafo de dependencia simple habrá una arista que una Σ y α.

Representando mediante una matriz booleana esta relación binaria, las columnas de esta matriz nos proporcionan las tablas de referencias cruzadas que dábamos en el apartado 1) de la Introducción.

2) Dependencia a izquierdas (representada por R_I)

Diremos que $\Sigma \in V_n$ y $\alpha \in V$ cumplen $\Sigma R_I \alpha$ si y solamente si α aparece como primer símbolo (distinto de la cadena vacía) en una producción de la gramática en la que Σ es la parte izquierda.

O sea:

$$\exists\ \Sigma \rightarrow \Sigma_1 \Sigma_2 \ldots \Sigma_p \alpha \phi$$
$$p \geq 0$$
$$\Sigma_1, \Sigma_2 \ldots \Sigma_p \in V_N \quad , \quad \phi \in V^*$$
$$\Sigma_1, \overset{*}{\rightarrow} \phi$$
$$\Sigma_2, \overset{*}{\rightarrow} \phi$$
$$\vdots$$
$$\Sigma_p, \overset{*}{\rightarrow} \phi$$

Donde por $\overset{*}{\rightarrow}$ representamos el cierre transitivo de la relación $\rightarrow$ ("produce") que definen las producciones de una gramática.

Tomando esta relación binaria y representándola en forma de la matriz booleana A(n,n) siendo n=cardinal (V), obtenemos una representación del grafo de dependencia a izquierdas fácilmente representable en un ordenador.

Si nos interesan los terminales que pueden encabezar una clase sintáctica dada o las clases sintácticas que pueden empezar por un terminal dado (puntos 2 y 3 de la Introducción) es suficiente con calcular el cierre transitivo de esta relación que representaremos por R_I^+ y que usando el teorema de Warshall calcularíamos mediante el siguiente algoritmo expresado en ALGOL (2)

```
Procedure WARS (A,N)
boolean array A[1:n,1:n];
begin integer i,j,k;
      for i: = 1 step 1 until n do
      for j: = 1 step 1 until n do
           if A(j,i) then
           for k: = 1 step 1 until n do
               A(j,k): = A(j,k) ∪ A(i,k);
end WARS
```

esto nos daría una nueva matriz $A^+(n,n)$ en la que, considerando las filas correspondientes a las metavariables, obtendríamos los terminales que pueden encabezarlas y considerando las columnas correspondientes a los terminales, las clases sintácticas que empiezan cada uno de ellos.

Veamos mediante un ejemplo sencillo la aplicación de estos métodos:

Dada la gramática $G\{V_T, U_N, P, S\}$

$V_T = \{a, b, (,), +, \dashv\}$

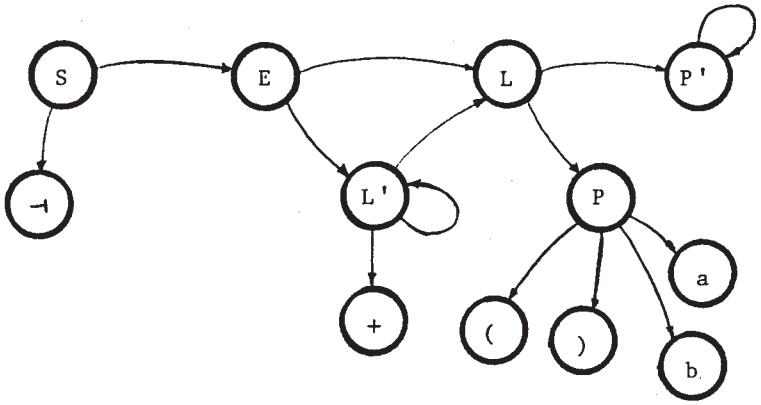
$V_N = \{E, L, L', P, P', S\}$

axioma: S

Producciones: P

$E \rightarrow LL'\dashv$	$P \rightarrow a$
$L' \rightarrow +LL'$	$P \rightarrow b$
$L' \rightarrow \Phi$	$P \rightarrow (E)$
$L \rightarrow PP'$	$S \rightarrow E$
$P' \rightarrow PP'$	
$P' \rightarrow \Phi$	

El grafo de dependencia simple sería:



que aceptaría la representación matricial:

		V_N						V_T					
		S	E	L'	L	P'	P	a	b	(	)	+	⊣
V_N	S	0	1	0	0	0	0	0	0	0	0	0	1
	E	0	0	1	1	0	0	0	0	0	0	0	0
	L'	0	0	1	1	0	0	0	0	0	0	1	0
	L	0	0	0	0	1	1	0	0	0	0	0	0
	P'	0	0	0	0	1	1	0	0	0	0	0	0
	P	0	1	0	0	0	0	1	1	1	1	0	0
V_T	a	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	0	0	0	0	0	0	0
	(	0	0	0	0	0	0	0	0	0	0	0	0
	)	0	0	0	0	0	0	0	0	0	0	0	0
	+	0	0	0	0	0	0	0	0	0	0	0	0
	⊣	0	0	0	0	0	0	0	0	0	0	0	0

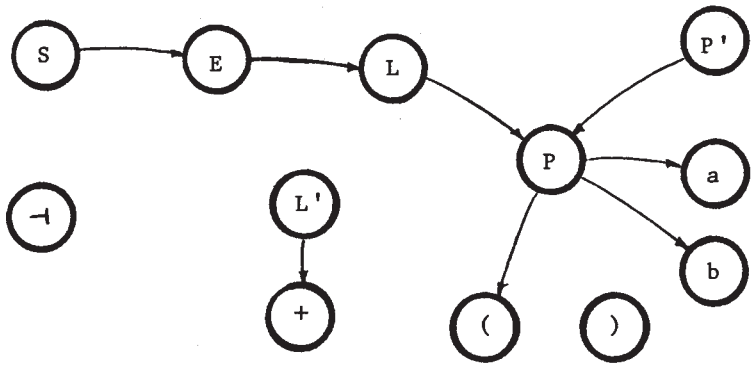
En la que vemos:

Escogiendo una columna cualquiera, hay un 1 en la regla donde aparece esta clase sintáctica o terminal (a la derecha) y un cero en caso contrario. Así:

Símbolo	Aparece en
E	S , P
L'	E , L'
⋮	⋮
a	P

Que son las tablas de referencia cruzadas a las que hacíamos referencia en I-1).

El grafo de dependencia a izquierdas sería:



que se puede representar matricialmente en la forma que se da en la primera tabla de la página siguiente.

Calculando el cierre transitivo de esta matriz usando el teorema de Warshall, llegaríamos a la matriz que se da en la segunda tabla de la página siguiente.

Escogiendo una fila cualquiera, hay un 1 en la columna cuyo símbolo puede encabezar la clase sintáctica asociada con la columna. Por ejemplo:

Símbolo	Puede empezar por
S	E , L , P , a , b , (
E	L , P , a , b , (
⋮	⋮

		V_N						V_T					
		S	E	L'	L	P'	P	a	b	(	)	+	⊣
V_N	S	0	1	0	0	0	0	0	0	0	0	0	0
	E	0	0	0	1	0	0	0	0	0	0	0	0
	L'	0	0	0	0	0	0	0	0	0	0	1	0
	L	0	0	0	0	0	1	0	0	0	0	0	0
	P'	0	0	0	0	0	1	0	0	0	0	0	0
	P	0	0	0	0	0	0	1	1	1	0	0	0
V_T	a	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	0	0	0	0	0	0	0
	(	0	0	0	0	0	0	0	0	0	0	0	0
	)	0	0	0	0	0	0	0	0	0	0	0	0
	+	0	0	0	0	0	0	0	0	0	0	0	0
	⊣	0	0	0	0	0	0	0	0	0	0	0	0

		V_N						V_T					
		S	E	L'	L	P'	P	a	b	(	)	+	⊣
V_N	S	0	1	0	1	0	1	1	1	1	0	0	0
	E	0	0	0	1	0	1	1	1	1	0	0	0
	L'	0	0	0	0	0	0	0	0	0	0	1	0
	L	0	0	0	0	0	1	1	1	1	0	0	0
	P'	0	0	0	0	0	1	1	1	1	0	0	0
	P	0	0	0	0	0	0	1	1	1	0	0	0
V_T	a	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	0	0	0	0	0	0	0
	(	0	0	0	0	0	0	0	0	0	0	0	0
	)	0	0	0	0	0	0	0	0	0	0	0	0
	+	0	0	0	0	0	0	0	0	0	0	0	0
	⊣	0	0	0	0	0	0	0	0	0	0	0	0

O sea, tenemos fácilmente los terminales que pueden encabezar una clase sintáctica (punto I-2)).

Si, por el contrario, escogemos una columna cualquiera (correspondiente a un terminal) la presencia de un 1 en ella indica que la clase sintáctica asociada con la fila en la que está puede empezar por él, con lo que fácilmente podemos construir la siguiente tabla:

Símbolo	Clases sintácticas que pueden encabezarlo
a	S , E , L , P', P
b	S , E , L , P', P
(	S , E , L , P', P
⋮	⋮

con lo que tenemos resuelto el punto I-3) y con ello la forma tradicional de resolver este problema. La memoria ocupada por la implementación de estos algoritmos (al margen de la procedure ALGOL,del algoritmo de definición de los grafos) sería:

cardinal (V) x cardinal (V)

que en nuestro sencillo ejemplo es 12 x 12 = 144 posiciones, o sea, depende de: n^2 siendo n=cardinal (V). Podemos pensar que si el lenguaje tratado es uno cualquiera de los habituales, este método es prohibitivo en memoria.

Se podría usar menos memoria a base de

1) Representar las matrices booleanas usando un bit por entrada. Pero esto aumenta mucho el tiempo de tratamiento de las tablas (3).
2) Como son matrices muy vacías hacer una representación sobre listas circulares ortogonales (4). Pero esto dificultaría también la implementación del algoritmo de WARSHALL.

Veamos a continuación nuestro método.

III. HASH-CODING Y GRAMATICAS

El procedimiento que hemos utilizado para resolver nuestro problema hace uso de una tabla hash, en la que las colisiones se representan sobre listas lineales (en la terminología de Knuth) implementadas sobre vectores.

Veamos paso a paso nuestro algoritmo.

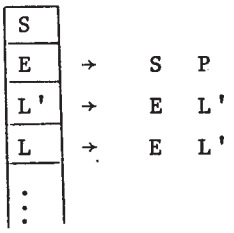
1) Obtención de las tablas I-1 y I-2 (tablas de referencias cruzadas).

Barremos la gramática de forma que obtenemos la representación en forma de diccionario de siguientes del grafo de dependencia simple:

Diccionario de siguientes (en R_s)

S	E , ⊣
E	L , L'
L'	+ , L , L'
L	P , P'
:	:

Tomando como clave, uno a uno, los siguientes de cada elemento, metemos en las listas de overflow los antecedentes asociados con cada clave. Al final del proceso tenemos, en forma de tabla hash las tablas de referencias cruzadas



vector Hash — listas de overflow

Este punto lo resolveremos en forma análoga al procedimiento clásico, la única diferencia es que tomamos como representación del grafo el diccionario de siguientes en vez de la matriz booleana.

Para resolver el problema siguiente: terminales por los que empieza cada clase sintáctica y clases sintácticas que empiezan por un terminal dado, utilizamos el siguiente algoritmo: recolectemos, analizando la gramática, las cabeceras de cada parte derecha de las producciones construyendo así el diccionario de siguientes del grafo de dependencia a izquierdas:

Diccionario de siguientes (en R_I)

símbolo	siguiente
S	E
E	L
L	P
L'	+
P'	P
P	a, b, (

y a partir del diccionario de siguientes buscamos el diccionario de antecedentes, inicialmente sólo para los terminales. Esto nos da:

Diccionario de antecedentes

símbolo	antecedente
+	L'
a	P
b	P
c	P
(	P

Buscando en la tabla de antecedentes anterior, si encontramos una clase sintáctica (por ejemplo en a\|P, la P) con ella buscamos en la tabla de siguientes cuál es su símbolo antecedente (en nuestro caso P : P' y L) y añadimos estos antecedentes en el diccionario de antecedentes y en la fila del terminal donde encontramos la clase sintáctica. Esto lo haremos en todos los sitios donde aparezca esta clase sintáctica, lo que nos dará:

+	L'
a	P, P', L
b	P, P', L
c	P, P', L
(	P, P', L

Nuevamente (y ya sin considerar la clase sintáctica ya usada) aplicamos el mismo criterio para todas las demás clases sintácticas que aparecen como antecedentes en la nueva tabla, lo que da al final la tabla de clases sintácticas que empiezan por un terminal dado.

+	L'
a	P, P', L, E, S
b	P, P', L, E, S
c	P, P', L, E, S
(	P, P', L, E, S

El proceso de acceso a esta tabla de antecedentes se hace usando como clave el terminal que estemos considerando y usando como función de "mapping" el {AND} lógico con '3F' (hexadecimal) del propio terminal, lo que nos da un acceso directo al vector Hash, en donde encontramos una referencia a su lista de overflow y empezamos una búsqueda secuencial hasta encontrar una clase sintáctica no tratada todavía.

Al final disponemos de una tabla que nos da para cada símbolo terminal las clases sintácticas que pueden empezar por él. Sólo nos queda obtener los terminales que encabezan cada clase sintáctica; esta tabla la obtenemos sin más que buscar en las listas de overflow cada clase sintáctica y añadir los terminales que las encabezan en la entrada correspondiente a cada clase sintáctica.

IV. CONCLUSIONES

El método que hemos descrito lo hemos implementado sobre UNIVAC-9300. En nuestro caso y dadas las limitaciones de memoria, el tomar como representación del grafo el diccionario de siguientes (y/o antecedentes) ha significado un considerable ahorro de memoria, unido a una fácil implementación.

El proceso ha sido aplicado a la gramática del lenguaje 70/13(5), cuyo traductor estamos metaprogramando.

Nuestros requisitos de memoria son considerablemente inferiores a los del método clásico, aunque el tiempo de tratamiento sea mayor.

Damos en el apéndice los resultados obtenidos en forma de output directo de la máquina.

Bibliografía

1) BOUSSARD, J.C.- Analyse syntaxique et compilation. IMAG. Grenoble, 1968-69.
2) GOMEZ BUENO, L., RAMOS SALAVERT, I.- Cierre transitivo de una relación binaria entre los elementos de un conjunto finito: Teorema de Warshall.- Nota interna del Instituto de Informática. Madrid, 1971.
3) DUBY, J.J., MALING, G.- A set-theoretic application of IBM 7094 computer instructions.- Report n° 6, IBM. 1964.
4) KNUTH, D.E.- The art of computer programming.- Vol. 1. Addison Wesley, 1968.
5) FERNANDEZ-FLOREZ, I., GARCIA CAMARERO, E., RAMOS SALAVERT, I. A basic language oriented to secondary schools.- IFIP 71. Ljubljana. August 1971.

INFORMATION PROCESSING 71 – NORTH-HOLLAND PUBLISHING COMPANY (1972)

A BASIC LANGUAGE ORIENTED TO SECONDARY SCHOOL

I. FERNANDEZ-FLOREZ, E. GARCIA CAMARERO and I. RAMOS

Centro de Cálculo de la Universidad de Madrid
Madrid, Spain

A language, together with its translator, is presented. The area of application is the teaching of Computer Science at a Secondary School level. The language is at an assembly level and is implemented on an ideal computer. This allows a very simple minded design of the system, which is easily understood by students (12–13 years old). The assembly level has been chosen to keep the student near the computer for educational purposes, but the syntax of the language allows a free way of writing the instructions, which are correct Spanish sentences. In this way we have the advantages of the high level languages (syntax), and at the same time those of the assembly languages (semantics). The system is implemented on an IBM-7090 computer, and it has been used for a year with satisfactory results. Other Universities in Spain have implemented the same system on different computers.

1. INTRODUCTION

The teaching of computer science in Spain has grown considerably the last year and the need of languages oriented to Secondary School has arisen. In this way the child must have a contact with the computer as soon as possible [2]. High level languages, easy to learn, have the inconvenience that they are far away from the principles of computers, and the assembly languages are hard to learn. With this problem in mind, we have designed an ideal computer and an assembly language [3] implemented within it which allows programs to be written in a simple form. The system has been implemented on the IBM 7090 of the University of Madrid and has been tested and used for a year. The experience with children shows the extreme feasibility of the system and the facility of teaching the principles of computer science in a practical way.

The system is used currently and the programmers (12–13 years old) give suggestions as feedback for the modification of the design.

Following is a description of the language and the translator and an example of a typical run of the system.

2. THE LANGUAGE

The language implemented is an assembly language with a simple syntax that makes the writing of programs as easy as possible.

The assembly level of the language keeps the children near the machine and allows the understanding of the algorithms from a machine point of view. The simulated machine has a core storage of one hundred "positions" (each position capable of holding instruction or datum), a register accumulator for the arithmetic computation, and an address counter accessible from the program. This allows the control and looping instructions of programs.

The instructions are written in Spanish and they are sentences of the usual language. The constraints are on the verbs and on the absolute address manipulated. The rest of the instruction is in free form and usually contains the necessary words to make the instruction a correct Spanish sentence.

We give a context-free grammar describing the language. There is a one-to-one correspondence between the source instruction and the object instruction. The latter is a FORTRAN IV instruction equivalent to the former, and the object program is in FORTRAN IV of the IBM 7090.

Fernández Flórez, I., García Camarero, E., Ramos, I. A Basic Language Oriented to secondary School. Information Processing 71. 1972.

The grammar is:

```
<program>      ::= <instruction> <programcont>

<programcont>  ::= ∅ |
                   <instruction> <programcont>

<instruction>  ::= <ioinst> |
                   <aritinst> |
                   <controlinst> |
                   <datainst>

<ioinst>       ::= leer <continst> |
                   escribir <continst>

<continst>     ::= <string> <address> <string>

<aritinst>     ::= sumar <cont> |
                   restar <cont> |
                   multiplicar <cont> |
                   dividir <cont> |
                   hallar <string>

<cont>         ::= <continst> |
                   <string> = <number>

<controlinst>  ::= poner <continst> |
                   si la <continst> <relop> <continst> |
                   si el <continst> |
                   hacer <number> <string> <number>
                                          <string> |
                   parar |
                   fin

<datainst>     ::= depositar <continst> |
                   trasladar <cont>

<string>       ::= <character> <contstring>

<contstring>   ::= ∅ |
                   <character> <contstring>

<character>    ::= a |
                   b |
                   .
                   .
                   .
                   z |
                   b

<number>       ::= int . int |
                   int . |
                   . int |
                   int

<int>          ::= <digit> <intcont>

<intcont>      ::= ∅ |
                   <digit> <intcont>

<relop>        ::= mayor |
                   menor |
                   igual

<address>      ::= (<digit> <digit>)

<digit>        ::= 0 |
                   1 |
                   .
                   .
                   .
                   9
```

Following is a list of the possible instructions:

LEER EN LA POSICION (XX) EL CONTENIDO DE LA UNIDAD DE ENTRADA.

DEPOSITAR EN LA POSICION (XX) EL CONTENIDO DEL ACUMULADOR.

ESCRIBIR EL CONTENIDO DE LA POSICION (XX) EN LA UNIDAD DE SALIDA.

TRASLADAR EL CONTENIDO DE LA POSICION (XX) AL ACUMULADOR.

TRASLADAR AL ACUMULADOR EL NUMERO=XXXXXX

SUMAR AL ACUMULADOR EL CONTENIDO DE LA POSICION (XX).

SUMAR AL ACUMULADOR EL NUMERO = XXXXXX.

RESTAR DEL ACUMULADOR EL CONTENIDO DE LA POSICION (XX).

RESTAR DEL ACUMULADOR EL NUMERO = XXXXXX.

MULTIPLICAR EL ACUMULADOR POR EL CONTENIDO DE LA POSICION (XX).

MULTIPLICAR EL ACUMULADOR POR EL NUMERO = XXXXXX.

DIVIDIR EL ACUMULADOR POR EL CONTENIDO DE LA POSICION (XX).

DIVIDIR EL ACUMULADOR POR EL NUMERO = XXXXXX.

HALLAR LA RAIZ CUADRADA DEL ACUMULADOR.

PONER EN EL CD LA DIRECCION (XX).

SI EL ACUMULADOR EL MAYOR QUE CERO PONER (XX) EN EL CD.

SI LA POSICION (XX) ES MENOR QUE EL ACUMULADOR PONER (YY) EN EL CD.

SI LA POSICION (XX) ES IGUAL QUE EL ACUMULADOR PONER (YY) EN EL CD.

SI LA POSICION (XX) ES MAYOR QUE EL ACUMULADOR PONER (YY) EN EL CD.

HACER MM VECES LAS NN INSTRUCCIONES SIGUIENTES.

PARAR

FIN

DATOS

FINAL

```
***    MAQUINA DE CALCULAR 7013    ***

SI HAY ERRORES EN ESTE PROGRAMA APARECERAN
   DEBAJO DE LA INSTRUCCION EQUIVOCADA

                P R O G R A M A
                ---------------

POSICION                  INSTRUCCION
--------                  -----------

 0   LEE EN LA POSICION (99) EL CONTENIDO DE LA UNIDAD DE ENTRADA
 1   LEE EN LA POSICION (98) EL CONTENIDO DE LA UNIDAD DE ENTRADA
 2   LEE EN LA POSICION (97) EL CONTENIDO DE LA UNIDAD DE ENTRADA
 3   TRASLADA EL CONTENIDO DE LA POSICION (99) AL ACUMULADOR
 4   SI LA POSICION (90) ES IGUAL AL ACUMULADOR PONER (7) EN EL ID
 5   MULTIPLICA EL ACUMULADOR POR EL NUMERO = 3600
 6   DEPOSITA EN LA POSICION (96) EL CONTENIDO DEL ACUMULADOR
 7   TRASLADA EL CONTENIDO DE LA POSICION (98) AL ACUMULADOR
 8   SI LA POSICION (90) ES IGUAL AL ACUMULADOR PONER (11) EN EL ID
 9   MULTIPLICA EL ACUMULADOR POR EL NUMERO = 60
10   DEPOSITA EN LA POSICION (95) EL CONTENIDO DEL ACUMULADOR
11   TRASLADA EL CONTENIDO DE LA POSICION (97) AL ACUMULADOR
12   SUMA AL ACUMULADOR EL CONTENIDO DE LA POSICION (96)
13   SUMA AL ACUMULADOR EL CONTENIDO DE LA POSICION (95)
14   DEPOSITA EN LA POSICION (94) EL CONTENIDO DEL ACUMULADOR
15   ESCRIBE EL CONTENIDO DE LA POSICION (94) EN LA UNIDAD DE SALIDA
16   PARA
17   FIN
                        D A T O S
                        ---------
                        25
                        0
                        26
                    R E S U L T A D O S
                    -------------------
                      90026 00
```

Fig 1 Source program

```
        ERROR        -  EFN    SOURCE STATEMENT

        DIMENSION A(100)
        PRINT 300
300     FORMAT(////25X,19HR E S U L T A D O S/25X)
100     FORMAT(F6.2)
200     FORMAT(25X,F10.2)
        DO 500 I=1,100
500     A(I)=0
        AC=0
1       READ ( 5, 100 ) A(99)
2       READ ( 5, 100 ) A(98)
3       READ ( 5, 100 ) A(97)
4       AC=   A(99)
5       IF(AC.EQ. A(90))GO  TO  8
6       AC=AC*FLOAT( 3600 )
7       A(96)=AC
8       AC=   A(98)
9       IF(AC.EQ. A(90))GO  TO  12
10      AC=AC*FLOAT( 60   )
11      A(95)=AC
12      AC=   A(97)
13      AC=AC+A(96)
14      AC=AC+A(95)
15      A(94)=AC
16      PRINT 200,A(94)
17      STOP
        END
```

Fig. 2 Object program.

An example of use of the language with the object program generated is given in fig 1 and fig 2

3 THE TRANSLATOR

The translator is a program in FORTRAN IV and MAP (Macro-Assembly-Program for the IBM 7090) which simulates an ideal computer on the IBM 7090 of the University of Madrid. This ideal computer accepts cards as input and prints lines as output [1]

The translation is a one-pass process because the instructions are identified by key-words and the addresses manipulated are absolutes

We may describe the actions of the translator in the following way

1 Read in a card (it is a source instruction)
2 Print it on the on-line printer
3 Syntactic analysis of the instruction.
4. If there are no errors, generate the equivalent object instruction and go to step 1
5. If there are any errors, print the type of error and go to step 1

The end of the process is marked out by a special card.

If there are no errors, the object program is executed and the results appear on the on-line printer.

This approach allows the use of the very efficient FORTRAN IV compiler and solves the problem in a straightforward form

Technical information about the translator may be obtained from the mailing address of the authors.

4. CONCLUSION

The described system has been used in the last year by students 12--13 years old, solving elementary problems in Physics and Mathematics

The didactical results [4] are satisfactory and the learning of the language takes about 15 hours The number of runs necessary to obtain a correct program does not exceed four (syntactical correction of errors)

Other implementations of the system have been done in other Universities of the country, in other types of computers

REFERENCES

[1] I. Fernández-Flórez and I. Ramos, Traductor 70/13, Bol. CCUM, 10 (1970) 13

[2] E. García Camarero, La enseñanza de ordenadores en la escuela secundaria, Bol. CCUM, 7 (1969) 5.

[3] E. Garcia Camarero, La máquina didáctica 70/13, Bol. CCUM, 8–9 (1970) 28

[4] E. García Camarero, La enseñanza de ordenadores en la escuela secundaria, Bol. CCUM, 12 (1970) 1

CENTRO DE CALCULO DE LA UNIVERSIDAD DE MADRID

GARCIA CAMARERO, E.

RAMOS SALAVERT, I.

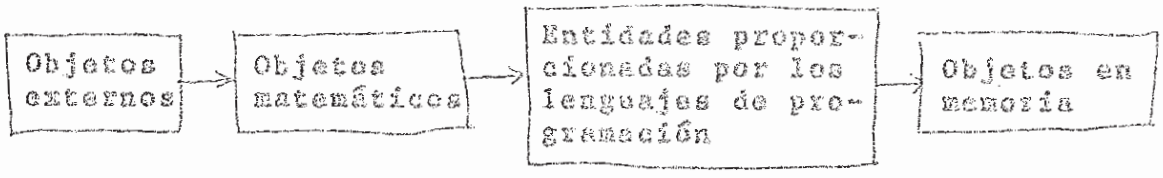
Ensayo sobre la definición automática de tratamientos clínicos

García Camarero, E., Ramos, I. Ensayo sobre la definición automática de tratamientos clínicos. Boletín CCUM. 1971.

ENSAYO SOBRE LA DEFINICION AUTOMATICA DE TRATAMIENTOS CLINICOS

I. Introducción

En el proceso de información por los ordenadores podemos establecer la siguiente clasificación en las representaciones de objetos:



El establecimiento de un modelo se plantea en los tres últimos pasos, y más concretamente en los dos intermedios, ya que el paso a representación en memoria suele hacerse automáticamente mediante un traductor.

Nuestro trabajo intenta dar un modelo de aprendizaje (definido como objeto cateroo) a los niveles:

- objeto matemático: autómata de estados finitos no determinista.
- entidades proporcionadas por los lenguajes de programación: programas.

Se ha tomado un ejemplo concreto para ilustrar el modelo y su implementación.

Definimos el proceso de aprendizaje como el paso de un autómata probabilista (donde todas las opciones son posibles) a un autómata determinista tal que su respuesta es correcta.para los distintos inputs.

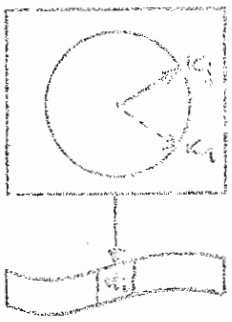
II. Soporte teórico

II.1. Modelo matemático

Definamos previamente el concepto de autómata de estados finitos como objeto matemático de nuestra representación. Entendemos por tal la 5-pla: $A(K, k_o, F, \Sigma, \delta)$ siendo:

K : alfabeto finito de estados.
$K_o \in K$: estado inicial del autómata.
$F \subset K$: conjunto de estados finales del autómata.
Σ : alfabeto de entrada.
δ : $(K \times \Sigma) \to \mathcal{P}(K)$. δ nos realiza la aplicación del producto cartesiano $K \times \Sigma$ sobre el conjunto de las partes de K.

Podríamos hacer una representación gráfica de nuestro autómata como una "caja negra" con un conjunto finito de estados, una cabeza lectora y una cinta de input:



Inicialmente el autómata frente a $a_i \in \Sigma$ y $K_j \subset K$ tiene como posible estado siguiente $K_q \subset K$ (la ignorancia total la representamos por la equiprobabilidad de transición a cualquier estado:

$$\delta (K_j \times a_i) \rightarrow K_q$$

Añadimos también una función de output o de respuesta, definida como:

$$\beta (K_j \times a_i) \rightarrow O_\ell$$

siendo: O : alfabeto de salida

La función β de output realiza la aplicación:

$$\beta : (K \times \Sigma) \rightarrow \mathcal{P}(O)$$

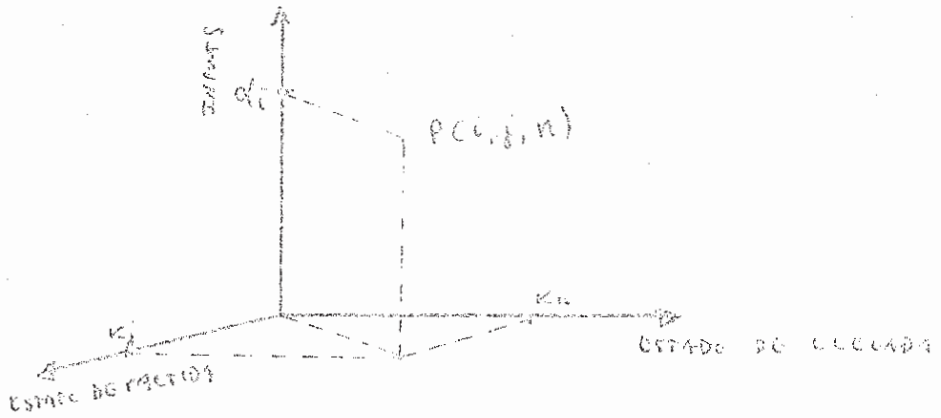
o sea, inicialmente cualquier output es posible partiendo de cualquier estado y llegando cualquier input. El aprendizaje del autómata vendrá dado por el paso de δ y β a funciones:

$$\delta : (K \times \Sigma) \rightarrow K$$

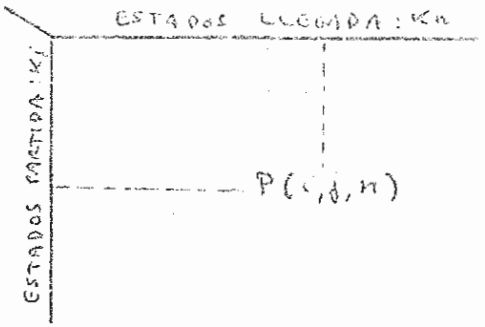
$$\beta : (K \times \Sigma) \rightarrow O$$

En definitiva, por el paso a un autómata determinista,

una representación usual del autómata es mediante sus matrices de transición tridimensionales



o sea, para cada input : α_i



Esta representación matemática del autómata acepta una fácil representación como objeto de un lenguaje de programación sobre una matriz array (card(K) x card(K)) y el proceso de aprendizaje del autómata (sus movimientos en el tiempo) se adapta a un proceso iterativo fácil de implementar en todo lenguaje de programación.

Esto nos lleva al modelo como entidad representada en un lenguaje de programación.

II.2. Representación del modelo en un lenguaje de programación

Sugerimos que la representación del autómata sea la siguiente:

K : alfabeto finito de estados: es una variable J que adquiere como valor: 1, 2, 3,... los sucesivos estados del autómata.

K_0 : estado inicial: un valor concreto de J.

F : subconjunto de valores de J.

Σ : alfabeto de input: variables I que adquieren un valor entero para cada input.

δ : la función de transición del autómata la representamos mediante una matriz tridimensional T(I, J, N).

donde I = input, J = estado de partida del autómata, N = estado de llegada del autómata.

O : función de output es la matriz tridimensional O(I, J, N).

III. Nuestra implementación

Partiendo de las ideas del punto II, damos a continuación un esquema de nuestro proceso.

A1. El estado inicial del autómata (de ignorancia) viene representado por: $T(I, J, N) = \frac{1}{card(K)}$.

A2. Existe un autómata determinista (que juega el papel de

medio externo) que en todo momento informa al autómata que aprende si su movimiento o decisión ha sido correcto o no.

A3. Dependiendo de si el movimiento del autómata ha sido correcto o no, se modifica T(I, J, N); para ello se utilizan las siguientes funciones de premio y castigo.

Funciones de premio y castigo

Función de premio:

Multiplicamos por un factor $K \geq 1$ tal que:

$$T_{t+1}(I, J, N) = K \times T_t(I, J, N)$$

Añadimos los momentos (tiempo del autómata: T_t...) y debe cumplirse:

$$1 \geq T_{t+1}(I, J, N) \geq T_t(I, J, N)$$

Dividiendo por $T_t(I, J, N)$

$$\frac{1}{T_t(I, J, N)} \geq K \geq 1$$

Luego K estará comprendido en el segmento $\left[1, \frac{1}{T_t(I, J, N)}\right]$ por lo que debe tener la forma:

$$K = 1 + \lambda\left[\frac{1}{T_t(I, J, N)} - 1\right]$$

con λ cumpliendo la condición:

$$0 \leq \lambda \leq 1$$

λ marcaría la mayor o menor dureza del castigo.

De esta forma el premio no hace que $T_{t+1}(I, J, N)$ supere la unidad (suma de las probabilidades de transición) y además $T_{t+1}(I, J, N) > T_t(I, J, N)$, hemos bonificado la transición.

Función de castigo:

Tomamos análogamente un factor multiplicativo K':

$$T_{t+1}(I, J, N) = K' \times T_t(I, J, N)$$

y tal que:

$$T_{t+1}(I, J, N) \leq T_t(I, J, N) \leq 1$$

Dividiendo por $T_t(I, J, N)$ y teniendo en cuenta la primera relación obtenemos:

$$K' \leq 1$$

y como las probabilidades han de ser ≥ 0, K' cumple:

$$0 \leq K' \leq 1$$

que es la restricción que debe de cumplir la función de castigo.

La aplicación de una función u otra obliga a toda una distribución de probabilidades de la matriz para que se siga cum-

plicando:

$$\sum_{N=1}^{K} T(I, J, N) = 1$$

IV. Sugerencias sobre aplicación al tratamiento médico

Considerar al tratamiento médico como un modelo de aprendizaje, puede ser prematuro dada la complejidad del problema. Nosotros en el presente trabajo nos hemos limitado a indicar una metodología, es decir, el uso de los autómatas probabilísticos, para tratar problemas de aprendizaje.

De todas formas, quisiéramos también a modo indicativo, ver cómo podría plantearse el problema del tratamiento médico con una metodología similar; aunque, insistimos, su complejidad haría necesario revisarlo y aun acotarlo para que presentara alguna operatividad.

Consideramos al tratamiento médico como una función ϕ que asocia a cada cuadro clínico (o a conjuntos de cuadros clínicos) un tratamiento. Un cuadro clínico puede ser una sucesión de síntomas, es decir, que si llamamos S al conjunto de síntomas (alfabeto de entrada del lenguaje) cada cuadro clínico sería un elemento de S^*, y podríamos indicar con L_S al conjunto de los posibles cuadros clínicos. Por otra parte, podríamos designar por T al conjunto de las indicaciones (alfabeto de salida, o alfabeto del lenguaje de tratamiento), y en T^* estarían los tratamientos, e indicaremos por L_T al conjunto de los tratamientos clínicamente posibles.

El tratamiento médico será una función

$$\phi : L_s \to L_T$$

y tal que subordina otra función ψ definida

$$\psi : L_s \times L_T \to L_T$$

Diremos que un tratamiento es correcto si

$$\psi(x, \alpha) = \Lambda \qquad \begin{matrix} x \in L_s \\ \alpha \in L_T \end{matrix}$$

Mediante nuestro modelo, simularemos la función ϕ, teniendo en cuenta que hemos de premiar las transiciones que hagan $\phi(x, \alpha) = \Lambda$, y castigando las restantes. Los estados del autómata que nos lleven a cumplir esta condición serían los estados finales.

Es natural considerar como estados de salud a aquellos cuadros clínicos que se proyectan sobre la cadena vacía. Es decir, definiremos como estado de salud o normal, a un cuadro clínico $x \in L_s$ tal que $\phi(x) = \Lambda$, de esta forma queda definido $N \in L_s$ como:

$$N = \{x | x \in L_s \ , \ \phi(x) = \Lambda\}$$

LENGUAJES PARA LA ESCRITURA DE TRADUCTORES

Por Isidro RAMOS SALAVERT (*)

RESUMEN

Se ha implementado un sistema para la escritura de traductores sobre dos ordenadores distintos (un IBM-7090 y un UNIVAC-9300) y ha sido usado para implementar varios lenguajes (SIAL/71, PHYSICTRAN, 70-13, PL/1).

El fundamento del sistema es realizar el análisis sintáctico descendente y sin vuelta atrás. Las ideas originales se deben a D. E. Knuth, "Top-Dawn syntax analysis" (International Summer School on computer programming, Copenhagen, 1967) y J. M. Foster "A syntax improving Program" (R.R.E. Mem. 2389, julio 1967).

La diversidad de ordenadores y lenguajes a los que se ha aplicado muestra la gran flexibilidad del diseño, cuya implementación ha abierto la posibilidad de implementar lenguajes en nuestro país.

1. INTRODUCCION

La escritura de traductores representa un considerable esfuerzo de programación que, desde el principio, ha sido un handicap en el desarrollo del software básico de un ordenador. La automatización de partes de este trabajo ha conducido a la construcción de compiladores (1) y a otras técnicas de metaprogramación.

En nuestros días, los aspectos sintácticos de los lenguajes de programación pueden ser elaborados casi automáticamente dando analizadores (2) fácilmente implementables sobre ordenador.

Los aspectos morfológicos pueden ser tratados también en forma metaprogramada (3) y son solamente los aspectos semánticos los que se realizan "a mano" pese a los esfuerzos de formalización de esta última parte de la implementación de traductores (4).

Nuestro trabajo es sobre la implementación en diferentes máquinas (IBM - 7090 y UNIVAC 9300), y la aplicación a diferentes lenguajes: Physictran (5), SIAL/71 (6), PL/1 (7), 70/13 (8), de un sistema para la escritura de traductores (T.W.S.) previamente definido e implementado en Grenoble (I.M.A.G.), por Griffiths et al. (2) siguiendo las ideas de J. M. Foster (9) y D. E. Knuth (10).

Los resultados son satisfactorios y el proceso de implementación de un lenguaje se simplifica considerablemente.

2. BASE TEORICA

El fundamento del sistema es un proceso de análisis sintáctico descendente y sin vuelta atrás.

Para obtener el analizador necesitamos elaborar primero la gramática del lenguaje para que cumpla las cuatro condiciones dadas por D. E. Knuth para su máquina sintáctica, esto implica la reescritura de las reglas de la gramática siguiendo criterios no sintácticos, produciendo, a veces, descripciones sintácticas de los lenguajes un tanto artificiosas. Como contrapartida obtenemos un analizador muy modular que acepta la fácil inclusión de acciones semánticas en él, realizándose la evaluación sintáctica y semántica en paralelo.

La ausencia de vueltas atrás (*backtracking*) en la evaluación sintáctica evita el *backtracking* semántico, costoso y difícil de tratar.

Demos sin demostración las cuatro condiciones que imponemos en la gramática inicial (tipo 2) para obtener una gramática LL (1) en el sentido de D. E. Knuth:

Sea la gramática:

$$G\,[V_T, V_N, P, S]$$

con:

$$V = V_T \cup V_N$$

(*) Centro de Cálculo de la Universidad de Madrid

Ramos, I. Lenguajes para la escritura de traductores. Revista de Automática. Enero-Marzo 1973.

y:

V_T = alfabeto terminal.
V_N = alfabeto no terminal.
P = reglas de producción.
S = axiomas.

1.ª condición.

No hay clases sintácticas recursivas a izquierdas. O sea: si $X \in V_N$ e $Y \in V$ no se cumple la relación binaria R_L (dependencia a izquierdas) definida como:

$$X R_L Y$$

si

$$X \to \Sigma_1 \Sigma_2 \dots \Sigma_n Y \gamma$$

con:

$$X \in V_N,\ \Sigma_1 \in V_N,\ \Sigma_2 \in V_N \quad Y \in V, \quad \gamma \in V^*$$

$$\Sigma_n \in V_N \quad n \geq 0$$

y

$$\Sigma_i \overset{*}{\to} \phi, \quad \forall_i$$

tal que

$$1 \leq i \leq n,$$

entendiendo por R^*_L el cierre transitivo de R_L.

2.ª condición.

Los símbolos terminales que pueden encabezar las diferentes partes derechas de una producción forman conjuntos disjuntos.

O sea, si:

$$X \to Y_1 r_1$$
$$Y_2 r_2$$
$$\vdots$$
$$Y_n r_n$$

con:

$$X \in V_N, \quad Y_i \in V, \quad r_i \in V^* \quad \forall_i \leq n$$

entonces:

$$\text{FIRST}(Y_i) \cap \text{FIRST}(Y_j) = \phi \quad \forall i, j \leq n \ \text{ e } \ i \neq j.$$

siendo:

$$\text{FIRST}(Y_K) = \{x \mid X \in V_T, \quad Y_K R^*_L X\}.$$

3.ª condición.

Si una de las partes derechas de una regla es ϕ (cadena vacía), entonces la aplicación de esta opción no puede ir seguida por un carácter terminal que pertenezca a First (Y_K) para las otras alternativas (con la misma notación que para la 2.ª condición).

4.ª condición.

Ninguna clase sintáctica puede producir la cadena vacía ϕ en más de una forma distinta.

Si nuestra gramática cumple estas cuatro condiciones, esto nos asegura que sólo con el análisis del carácter terminal en mano podemos elegir la alternativa correcta en la aplicación de una regla sintáctica.

Veámoslo con un ejemplo:

Sea *G:*

$$G\{V_T, V_N, P, S\}$$

V_T = {dígito}

V_N = {<entero>}

P = <entero> :: = dígito | <entero> dígito

S : entero

La aplicación de las cuatro condiciones nos conduce a la gramática *LL* (1) débilmente equivalente de la anterior:

P':

<entero> :: = dígito <cont. entero>

<cont. entero> :: = ϕ | dígito <cont. entero>.

Por ser *LL* (1) podemos reescribirla en forma de rutinas recursivas (2):

```
RUTINA      ENTERO
TEST        'dígito'
LLAMAR      CONTENTERO
VOLVER
RUTINA      CONTENTERO
DECIDE      L 1, 'dígito'
SALIR
L 1 TEST    'dígito'
LLAMAR      CONTENTERO
VOLVER
```

Cada regla origina una rutina con punto de entrada: la parte izquierda de la regla. El cuerpo de la rutina está formado por la conversión de cada parte derecha en las siguientes acciones:

si es no terminal: LLAMAR "nombre del no terminal".
si es terminal: TEST "terminal".

Las distintas opciones se seleccionan introduciendo al comienzo de la rutina decisiones sobre los caracteres terminales disjuntos que las pueden encabezar.

La comprobación de que una frase sigue las reglas de la gramática empieza con una llamada al axioma lo que produce la evaluación sintáctica de la frase.

En cada momento de la evaluación sabemos exactamente el punto en el que estamos tanto en la cadena de símbolos terminales como en la gramática: esto permite la inclusión de rutinas semánticas en los puntos correctos (después de la comprobación de subcadenas correctas) y así evaluamos a la vez la sintaxis y la semántica.

En nuestro ejemplo esto dará lugar a la adición de la acción semántica: A_1 después del primer *fesif* y a la A_2 después del segundo, siendo estas acciones:

$A1$: $N \leftarrow$ 'dígito'.

$A2$: $N \leftarrow 10 * N +$ 'dígito'.

Al final del proceso tendremos evaluado en N el valor del entero (que sería la evaluación semántica).

3. LENGUAJES METASINTACTICOS

Distinguimos, sintácticamente, los aspectos morfológicos de los puramente sintácticos con finalidad de optimizar el proceso. Esto produce una separación de los "lenguajes" (metalenguajes) de tratamiento de estos dos aspectos lingüísticos originando:

— lenguaje metamorfológico;
— lenguaje metasintáctico.

Veamos con arreglo a estas ideas, cómo queda el proceso de implementación de un traductor:

Fases.

1. Descripción formalizada de los aspectos morfológicos del lenguaje a implementar usando una gramática *LL* (1):
Sea la gramática:

$$G\ (V_T, V_S, P, S)$$

con:

V_T = (conjunto de caracteres de la máquina).
V_S = (unidades sintácticas elementales y metavariables auxiliares).
P = (conjunto de reglas que describen la estructura de las unidades sintácticas en términos de V_T).
S = unidad sintáctica.

2. Reescritura de la gramática en un lenguaje macrodefinido (que llamaremos lenguaje morfológico) siguiendo las ideas del punto 2. Esto da lugar al preprocesador.

3. Descripción formalizada de los aspectos sintácticos del lenguaje usando también una gramática *LL* (1).

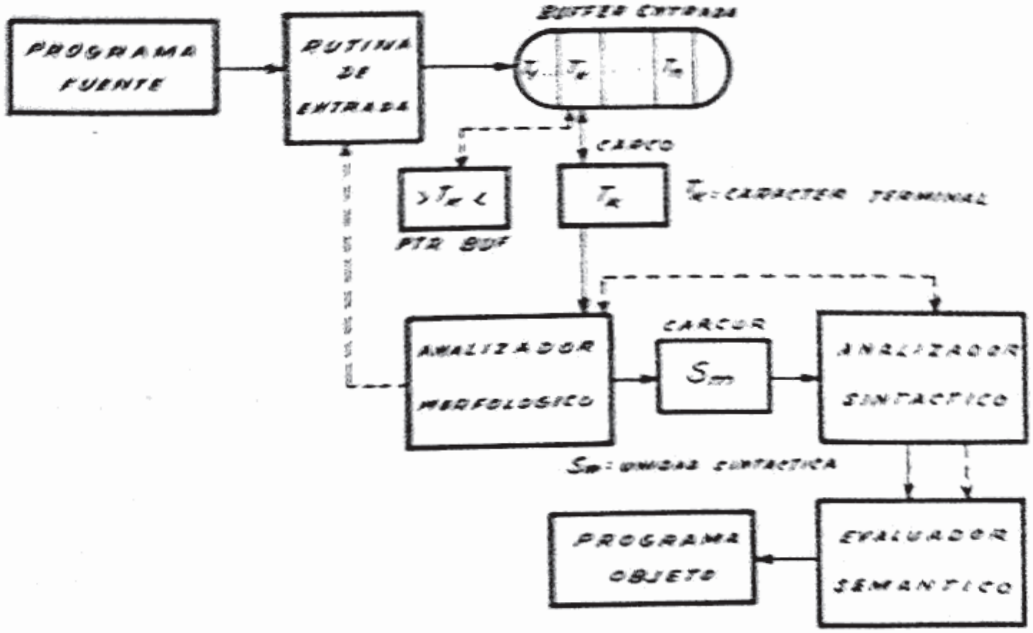


Figura 1.

Sea:

$$G\{V_T, V_N, P, S\}$$

con:

V_T = {unidades sintácticas elementales proporcionales por el preprocesador}.

V_N = {metavariables}.

P = {conjunto de reglas que describen el lenguaje}.

S = programa.

4. Reescritura de la gramática en un nuevo lenguaje macrodefinido (lenguaje sintáctico) siguiendo también las ideas del punto 2. Esto da lugar al analizador sintáctico que evalúa la cadena de unidades sintácticas que le proporciona una a una el preprocesador.

5. Entremezcladas con ambas gramáticas tenemos las correspondientes acciones semánticas escritas en su correspondiente lenguaje metasemántico (macrodefinido y lenguaje ensamblador).

Un diagrama funcional de todo el proceso se puede ver en figura 1.

CARCO = Ventana a través de la cual el preprocesador "ve" la cadena terminal.

CARCUR = Ventana a través de la cual el analizador sintáctico "ve" la cadena de unidades sintácticas.

Dado el paralelismo entre el analizador morfológico y sintáctico es de esperar que a los correspondientes metalenguajes les ocurra lo mismo.

Detallamos a continuación ambos lenguajes.

Lenguaje metamorfológico (macrodefinido).

Macro RUTINA. Define el punto de entrada a la rutina como símbolo externo. Es la primera macro de toda rutina. Ejemplo:

```
RUTINA      ENTERO
```

Define ENTERO como punto de entrada a la rutina que encabeza.

Macro VOLVER. Delimita sintácticamente el fin de toda rutina, siendo por tanto la última macro de toda rutina.

Macro TEST 1. Es un predicado que en caso de no cumplirse emite el mensaje de error correspondiente. Damos una versión en ALGOL:

```
if CARCO ≠ PARAM then ERROR else AVANZA
```

ERROR = procedure de emisión de un error.

AVANZA = procedure que avanza 1 carácter en la cadena terminal.

Ejemplo:

```
TEST 1 *
```

se convertirá en:

```
if CARCO ≠ * them ERROR else AVANZA
```

Macro AVANZA. Avanza 'n' caracteres en la cadena de caracteres terminales.

Así:

```
AVANZA    3
```

Macro DECID 1. Nos sirve para decidir sobre las distintas opciones de una clase sintáctica basándonos en los caracteres terminales que las encabezan. Ejemplo:

```
DECID 1    L 1, *
```

que en ALGOL sería:

```
if CARCO = * then go to L 1
```

Macro PUT. Emite en una variable global la unidad sintáctica detectada. Ejemplo:

```
PUT    "identificador"
```

Incluye además las macros: LLAMAR y SALIR, que veremos a continuación.

Lenguaje metasintáctico.

Además de las macros RUTINA y VOLVER comunes con el anterior, incluye:

Macro LLAMAR. Transfiere el control a la rutina, cuyo nombre es su argumento, guardando la dirección de la instrucción siguiente en una pila de direcciones. Ejemplo:

```
LLAMAR      ENTERO
```

Macro SALIR. Transfiere el control a la dirección contenida en la cumbre de la pila de direcciones decapitándola después. Ejemplo:

```
SALIR
```

Macro TEST. Juega un papel análogo a la TEST 1, pero sobre la cadena de unidades sintácticas. O sea:

```
if CARCUR ≠ PARAM then ERROR else AVANZA
```

(Aquí el avance es sobre la cadena de unidades sintácticas.)

Macro DECIDE. Juega un papel análogo a la DECID 1, pero sobre la cadena de unidades sintácticas. O sea:

```
if CARCUR = PARAM then go to L 1
```

La implementación de estas macros se da en el Apéndice para la IBM-7090.

4. CONCLUSIONES

El sistema descrito ha sido implementado y usado en diferentes ordenadores (IBM-7090 y UNIVAC-9300) y aplicado a diferentes lenguajes (SIAL 171, 70/13, PHYSICTRAN y PL/1), habiendo dado resultados satisfactorios.

La diversidad tanto de las máquinas como de los lenguajes en que ha sido usado muestra la gran flexibilidad del diseño.

5. AGRADECIMIENTOS

A. M. Griffiths y J. C. Boussard (I.M.A.G., Grenoble), que me dieron las ideas y ayuda originales.

A. M. Alfonseca y M. Moya, sin cuya ayuda ambas implementaciones no hubieran sido posibles.

BIBLIOGRAFIA

1. ROSEN, S. "A compiler building system developed by Brooker and Morris". C.A.C.M. July 1964.

2. GRIFFITHS, M., PELTIER, M. "Grammar translation as an aid to compiler production". Etude n.° FF2-0057-2 Centre Scientifique IBM Grenoble (France). 1968.

3. JOHNSON, W. L., et al. "Automatic generation of efficient lexical processors using finite state techniques". C.A.C.M. 11, Dec. 1968.

4. STEEL, T. B. (Jr.) "Formal language description languages for computer programming". Proc. of the IFIP working conference. Vienna. 1964.

5. RAMOS, I. "Construcción de un traductor para un lenguaje especialmente orientado a problemas físicos". Doctoral Thesis. Facultad de Ciencias. Madrid. 1971.

6. ALFONSECA, M. "Modelos electrónicos de procesos de regulación biológica". Doctoral Thesis. E.T.S.I. Telecomunicación. Madrid. 1971.

7. ALFONSECA, M., RAMOS, I. "Un analizador sintáctico para un subconjunto de PL/1".

8. I. FERNANDEZ-FLOREZ, E. CAMARERO, I. RAMOS. "A basic language oriented to secondary schools". IFIP Congress 71. Ljubljana. August 1971.

9. FOSTER, J. M. "A syntax improving program". R.R.E. Memorandum 2362. July 1967.

10. KNUTH, D. E. "Top-down syntax analysis". International Summer school on computer programming. Copenhaguen, August 1967.

APENDICE

```
LLAMAR  MACRO   X
        CLA     RETPT
        ADD     =1
        STO     VRETPT
        CAS     PUNSTA
        TRA     LINEAPI
        NOP
        CLA     **,I
        STCM    RETPT
        TSX     F,4
        FIN
        ENDM    LLAMAR
ENTRAR  MACRO   X
        TRA     F
        ENDM    ENTRAR
SALIR   MACRO
        CLA     RETPT
        STA     **,7
        SWP     =1
        STO     RETPT
        CAS     POSNST
        TRA     **,3
        TRA     **,2
        TRA     LNDOFL
        TRAP    **
        ENDM    SALIR
VOLVER  MACRO
        NADA
        ENDM    VOLVER
BUFTAM  MACRO   X
$       EQU     *
        ENDM    BUFTAM
DECIMF  MACRO   X,Y
        LDF     Y
        CLA     =F
        SUB     CARCOM
        TSX     X
        NOP
        ENDM    DECIMF
RESTA   MACRO   X
        CLL     =F
        LAS     CARCOM
        TRA     **2
        TRA     **,5
        TYLOB   *
        WTOM    1
        MOVE    CONM2
        TEOB    *
        CLA     PTRSUB
        ADD     =1
        STO     PTRSUB
        CLA     CARCOM
        STO     CARCOM+1
        CLA*    PTRSUB
        STO     CARCOM
        ANSW    RESTA
RESET   MACRO   X
        CLA     =X
        CAS     CARCOM
        TRA     **2
        TRA     **,5
        TEOB    *
        WTOM    1
        MOVE    CONM1
        TEOB    *
        TSX     UP,4
        TSX     WRITE,4
        ENDM    RESET
```

CONGRESO NACIONAL AUTOMATICA 1972 - BARCELONA

UN ANALIZADOR SINTACTICO PARA UN SUBCONJUNTO DE PL/I

Por D. Manuel Alfonseca[1] y D. Isidro Ramos[2]

RESUMEN

En este trabajo se expone la construcción de un analizador morfológico y un analizador sintáctico para un subconjunto de PL/I. El sistema utilizado es análisis descendente y sin vuelta atrás, y ha sido implementado en el ordenador IBM/7090 del Centro de Cálculo de la Universidad de Madrid.

SUMMARY

This work is about the implementation of a lexical analyser and a syntax analyser for a subset of PL/I. The system used is top-down syntanx analysis without backtracking; it has been developed for the IBM/7090 computer of the Centro de Cálculo de la Universidad de Madrid.

1 Doctor Ingeniero de Telecomunicación. Centro de Cálculo de la Universidad de Madrid.

2 Doctor en Ciencias Físicas. Centro de Cálculo de la Universidad de Madrid.

Alfonseca, M., Ramos I. Un analizador sintáctico para un subconjunto de PL/I. Actas del Congreso Nacional de Automática. 1972.

I.- INTRODUCCION

Se ha construído un evaluador sintáctico (syntax checker) para un subconjunto de PL/I de acuerdo con las directrices dadas por D.E. Knuth, y J.M. Foster. El proceso ha sido dividido en - dos partes: Analizador morfológico y Analizador sintáctico, con lo cual, y a base de definir metalenguajes apropiados para cada fase, se obtiene una construcción efectiva y a su vez fácilmente metaprogramable en cualquier ordenador que acepte un macroensamblador. Las mismas ideas son fácilmente representables en - lenguajes de alto nivel (FORTRAN, ALGOL, ...) consiguiendose, - además de una normalización un ahorro de esfuerzo de programación.

II.- SOPORTE TEORICO

Nuestro proceso de análisis es descendente y sin vuelta atrás. Las ideas originales se deben a D.E. Knuth, (Ref. 1) y J.M. Foster (Ref. 2).

Una exposición detallada del soporte teórico se halla en el trabajo presentado a este mismo Congreso por I. Ramos (Ref. 3)

III.- ANALIZADOR MORFOLOGICO

1.- NECESIDADES A CUBRIR

La etapa de preproceso, en todo traductor, pretende establecer las unidades sintácticas elementales partiendo de la cadena de caracteres terminales. Se utilizan para ello habitualmente sistemas de transición basados en autómatas de estados finitos, deterministas, lo cual limita la estructura de las unidades sintácticas a descripciones basadas en gramáticas regulares. Esta limitación permite la construcción de preprocesadores muy eficientes, pero en los que hay muchos aspectos resueltos - en una forma ad-hoc impuesta por la restricción anteriormente - citada. Estos aspectos se plantean por una parte a nivel sintáctico (obtención de unidades sintácticas), y por otra a nivel - semántico (asignación de significado: símbolos). El procedimiento que utilizamos (4) resuelve el primero de ellos en forma más flexible sin necesidad de recurrir a tantos procedimientos ad-hoc, lo que proporciona una mayor facilidad en el diseño de preprocesadores a costa de un tiempo de preproceso no excesivamente grande.

Las gramáticas utilizadas para la descripción de los aspectos morfológicos de un lenguaje son de tipo LL(1), proporcionándonos mayor flexibilidad en la descripción de las unidades sintácticas y en la asignación de su significado.

Los aspectos ad-hoc son resueltos en la forma tradicional (construcción de la tabla de símbolos, eliminación de blancos y demás subcadenas sin valor sintáctico, etc.), facilitada por la modularidas del analizador morfológico obtenido.

Presentamos en el punto siguiente la gramática que nos describe los aspectos morfológicos del subconjunto de PL/I considerado.

2.- IMPLEMENTACION

La gramática en la que está basado nuestro preprocesador es la siguiente:

US → SP | PSC | PID | BL US | TINT | CCAR

SP → (PUT(60) |) PUT(28) | + PUT(16) | - PUT(32) | = PUT(11)

PSC → . SC1 | , SC2 | * SC3 | / SC4

SC1 → () PUT(27) | . PUT(10) | ØR

SC2 → () PUT(59) | . PUT(13)

SC3 → () PUT(44) | * PUT(14)

SC4 → () PUT(49) | * CØMENT

ØR → G CØR1 | N CØR2 | L CØR3 | EQ. PUT(29) | ØR. PUT(30) | AND. PUT(31) | CAT. PUT(42)

CØR1 → E. PUT(45) | T. PUT(46)

CØR2 → L. PUT(47) | G. PUT(58) | E. PUT(61) | ØT. PUT(62)

CØR3 → E. PUT(63) | T. PUT(64)

CØMENT → St CØMENT | * /

PID → LETRA CPID

CPID → () A2 | LETRA CPID | CIFRA CPID

BL → b CBL

CBL → b CBL | ()

TINT → CIFRA CICØN

CICØN → () PUT(15) | CIFRA1 CICØN

CCAR → ' SUBC ' PUT(26)

SUBC → St SUBCØN

SUBCØN → () |SUBCØN

LETRA → A|B|...|Z

CIFRA1 → 0|1 ...|b

CIFRA → 0|...|9

En ella, además de la descripción de las unidades sintácticas se añade, en los puntos en los que se detectan estas, la emisión - del símbolo correspondiente (PUT (...)), de acuerdo con la Tabla I (Asignación de códigos a los símbolos correspondientes).

La escritura de esta gramática en el lenguaje morfológico macrodefinido proporciona el preprocesador presentado en el Apéndice A.

IV.- ANALIZADOR SINTACTICO

1.- NECESIDADES A CUBRIR

Nuestro analizador sintáctico resuelve, por ahora, el primer problema que se plantea en la evaluación sintáctica de un - programa PL/I, es decir: indica si es o no correcto. El segundo problema, a saber: siendo correcto el programa, cuál es su significado, (construcción de la C- estructura correspondiente) está sin resolver por el momento. En definitiva, tenemos un comprobador de sintaxis.

La gramática que se ha usado para describir los aspectos sintácticos del lenguaje es también de tipo LL(1), lo que permite una implementación análoga a la del preprocesador, codificada en el lenguaje sintáctico macrodefinido.

2.- IMPLEMENTACION

La sintaxis del subconjunto de PL/I definido es la siguiente:

```
PROGRAM        -->  PROCEDURECL PROGRAMCONT

PROGRAMCONT    -->  ()
                    PROCEDURECL PROGRAMCONT

PROCEDURECL    -->  HEADPART 'PROCEDURE' PPARAMETERL PROCEDUREOP
                    'PCOMMA' SENTENCELIST

HEADPART       -->  ()
                    IDTHING ASSTATEMENT
                    IDTHING 'COLON' HEADPART

ASSTATEMENT    -->  'EQ' EXPRESSION
                    'COMMA' IDTHINGLIST 'EQ' EXPRESSION

IDTHINGLIST    -->  IDTHING
                    IDTHING 'COMMA' IDTHINGLIST

PPARAMETERL    -->  ()
                    PARAMETERL

PARAMETERL     -->  'LBRK' IDENTIFIER IDENTLIST 'RBRK'

IDENTLIST      -->  ()
                    'COMMA' IDENTIFIER IDENTLIST

PROCEDUREOP    -->  ()
                    'RECURSIVE'
                    'OPTIONS' 'LBRK' 'MAIN' 'RBRK'

SENTENCELIST   -->  SENTENCE SENTENCECONT ENDCLAUSE

SENTENCECONT   -->  ()
                    SENTENCE SENTENCECONT

ENDCLAUSE      -->  'END' PIDENTIFIER 'PCOMMA'

PIDENTIFIER    -->  ()
                    IDENTIFIER

SENTENCE       -->  HEADPART SENTEN

SENTEN         -->  'PROCEDURE' PPARAMETERL PROCEDUREOP 'PCOMMA'
                       SENTENCELIST
                    'ENTRY' PPARAMETERL ' PCOMMA'
                    'DECLARE' DECLARATIONL 'PCOMMA'
                    STATEMENT 'PCOMMA'
                    NULLSTATEMEN

DECLARATIONL   -->  DECLARATION DECLARACONT

DECLARACONT    -->  ()
                    'COMMA' DECLARATION DECLACONT

DECLARATION    -->  IDENTIFIER ATTLIST
                    'LBRK' DECLARATIONL 'RBRK' ATTLIST
```

```
ATTLIST      ---->   DIM1 ATTCON

DIM1         ---->   ()
                     'LBRK' BOUNDS1 'RBRK'

BOUNDS1      ---->   BOUN1 BOUNCONT

BOUNCONT     ---->   ()
                     'COMMA' BOUNDS1

BOUN1        ---->   EXPRESSION
                     'STAR'
                     EXPRESSION 'COLON' EXPRESSION

ATTCON       ---->   ()
                     ATT ATTCON

ATT          ---->   ARITHATT
                     'ENTRY'
                     INITIALATT
                     'LABEL' AFTERLABEL
                     STRINGATT 'LBRK' LENGTH 'RBRK'

ARITHATT     ---->   ARITH ARITHCON

ARITHCON     ---->   ()
                     PRECISION

ARITH        ---->   'BINARY'
                     'DECIMAL'
                     'FIXED'
                     'FLOAT'
                     'REAL'
                     'COMPLEX'

PRECISION    ---->   'LBRK' 'TINT' 'RBRK'
                     'LBRK' 'TINT' 'COMMA' POSSING 'TINT' 'RBRK'

AFTERLABEL   ---->   PPARAMETERL

STRINGATT    ---->   'CHARACTER'
                     'BIT'

LENGTH       ---->   'STAR'
                     EXPRESSION

INITIALATT   ---->   'INITIAL' INITPAR

INITPAR      ---->   'LBRK' INITLIST 'RBRK'

INITLIST     ---->   INIT1 INITCØNT

INICØNT      ---->   ()
                     'COMMA' INITLIST
```

```
INIT1        ---->  'LBRK' 'TINT' 'RBRK' INIT2
                    'STAR'
                    INIT3

INIT2        ---->  INIT3
                    INITPAR

INIT3        ---->  IDENTIFIER
                    CHARSTRING
                    POSSING NUMBER

CHARSTRING   ---->  'TERMSTRING' MAYBEB

MAYBEB       ---->  ()
                    'B'

STATEMENT    ---->  IFSTATEMEN
                    UNCONDITIONA

IFSTATEMEN   ---->  IFCLAUSE ST 'ELSE' IFCONT

IFCONT       ---->  ST
                    'PCOMMA'

ST           ---->  HEADPART NULLSTATEMENT
                    STATEMENT

IFCLAUSE     ---->  'IF' EXPRESSION 'THEN'

UNCONDITIONA ---->  BEGINBLOCK
                    GROUP
                    GOTOSTATEMEN
                    CALLSTATEMEN
                    RETURNSTATEM
                    EXITSTATEMEN
                    STOPSTATEMEN
                    STREAMIOST
                    NOTIMPLSTATE

NULLSTATEMEN ---->  'PCOMMA'

BEGINBLOCK   ---->  'BEGIN' 'PCOMMA' SENTENCELIST

GROUP        ---->  'DO' DOCON 'PCOMMA' SENTENCELIST

DOCON        ---->  ()
                    'WHILE' 'LBRK' EXPRESSION 'RBRK'
                    IDTHING 'EQ' SPECLIST

SPECLIST     ---->  SPEC RESTSPEC

RESTSPEC     ---->  ()
                    'COMMA' SPECLIST

SPEC         ---->  EXPRESSION STEPPART WHILPART
```

```
STEPPART      ---->  ()
                     'TO' AUX5
                     'BY' AUX6

AUX5          ---->  EXPRESSION 'BY' EXPRESSION
                     ()

AUX6          ---->  ()
                     EXPRESSION 'TO' EXPRESSION

WHILPART      ---->  ()
                     'WHILE' 'LBRK' EXPRESSION 'RBRK'
                     EXPRESSION
                     EXPRESSION 'WHILE' 'LBRK' EXPRESSION 'RBRK'

GOTOSTATEMEN  ---->  PGOTO IDTHING 'LBRK' EXPRESSION EXPRESSIONCT
                        'RBRK' 'PCOMMA'

RETURNSTATEM  ---->  'RETURN' 'LBRK' EXPRESSION 'RBRK' 'PCOMMA'
                     'RETURN' 'PCOMMA'

EXITSTATEMEN  ---->  'EXIT' 'PCOMMA'

STOPSTATEMEN  ---->  'STOP' 'PCOMMA'

STREAMIOST    ---->  'GET' 'LIST' 'LBRK' GETLIST 'RBRK' 'PCOMMA'
                     'PUT' PUTTHING 'LIST' 'LBRK' PUTLIST 'RBRK'
                        'PCOMMA'

PUTTHING      ---->  ()
                     'PAGE'
                     'LINE' LINENUM
                     'SKIP' SKIPNUM
                     'PAGE' 'LINE' LINENUM

LINENUM       ---->  ()
                     'LBRK' EXPRESSION 'RBRK'

SKIPNUM       ---->  ()
                     'LBRK' 'TINT' 'RBRK'

PUTLIST       ---->  EXPRESSION EXPRESSIONCT

GETLIST       ---->  IDTHING 'COMMA' IDTHINGLIST

EXPRESSIONCT  ---->  ()
                     'COMMA' EXPRESSION EXPRESSIONCT

NOTIMPLSTATE  ---->  'REVERT'
                     'SIGNAL'
                     'FREE'
                     'ALLOCATE'
                     'DELAY'
                     'UNLOCK'
                     'LOCATE'
                     'READ'
                     'REWRITE'
                     'WRITE'
                     'CLOSE'
                     'OPEN'
                     'FORMAT'
```

```
                    'RELEASE'
                    'ON'

EXPRESSION  ---->   EXPF EXPX

EXPX        ---->   ()
                    'OR' EXPF EXPX

EXPF        ---->   EXPE EXPFX

EXPFX       ---->   ()
                    'AND' EXPE EXPFX

EXPE        ---->   EXPD EXPEX

EXPEX       ---->   ()
                    'LT' EXPD EXPEX
                    'LE' EXPD EXPEX
                    'GE' EXPD EXPEX
                    'GT' EXPD EXPEX
                    'NE' EXPD EXPEX
                    'EQ' EXPD EXPEX
                    'NL' EXPD EXPEX
                    'NG' EXPD EXPEX

EXPD        ---->   EXPC EXPDX

EXPDX       ---->   ()
                    'CAT' EXPC EXPDX

EXPC        ---->   EXPB EXPCX

EXPCX       ---->   ()
                    'PLUS' EXPB EXPCX
                    'MINUS' EXPB EXPCX

EXPB        ---->   ()
                    'STAR' EXPA EXPBX
                    'SLASH' EXPA EXPBX

EXPA        ---->   PRIMARY
                    PRIMARY 'POWER' EXPA
                    'PLUS' EXPA
                    'MINUS' EXPA
                    'NOT' EXPA

PRIMARY     ---->   NOBRACKPRIM
                    'LBRK' EXPF EXPX 'RBRK' PRIMARYCON

PRIMARYCON  ---->   CHARSTRING
                    ()

NOBRACKPRIM ---->   IDTHING
                    NUMBER
                    CHARSTRING
```

```
IDTHING          IDENTIFIER IDCON1

IDCON1           ()
                 'LBRK' SUBSLIST 'RBRK'

SUBSLIST         EXPRESSION EXPRESSIONCT

IDENTIFIER       'TERMID'

NUMBER           'TINT' 'POINT' 'TINT' POSSB
                 'TINT' 'POINT' POSSB
                 'TINT' POSSB
                 'POINT' 'TINT' POSSB

POSSB            BTHING
                 'E' POSSIGN 'TINT' BTHING

BTHING           'B'
                 'B' 'I'
                 'I'
                 'BI'

POSSIGN          ()
                 'PLUS'
                 'MINUS'
```

El analizador sintáctico asociado viene incluído en el Apéndice B.

V.- CONCLUSIONES

La implementación ha sido realizada sobre la IBM-7090 del Centro de Cálculo de la Universidad de Madrid. Se han efectuado medidas de tiempo y se han comparado con las expuestas en la bibliografía (Ref. 5), son resultados muy favorables, dado el sistema propuesto. Los resultados fueron los siguientes:

Preprocesador: Relación de tiempos entre el sistema de la Ref. 5 nuestro sistema: 1.56 excluyendo el tiempo de entrada/salida; 0.62 incluyendolo.

Número de caracteres procesados por segundo = 3750 excluyendo el tiempo de entrada/salida; 1500 incluyendolo.

El analizador sintáctico procesa a razón de 2200 unidades sintácticas por segundo.

Teniendo en cuenta que la entrada se hace sobre cinta magnética (729 modelo 2) estos resultados son muy favorables.

VI.- REFERENCIAS

1. KNUTH, D.E., "Top-down syntax analysis". International School on Computer Programming. Copenhague, 1967.

2. FOSTER, J.M., "A syntax improving program", R.R.E. Memorandum 2352, 1967.

3. RAMOS, I., "Lenguajes para la escritura de traductores", este Congreso.

5. JOHNSON, W.L., et al., "Automatic generation of efficient - lexical processors using finite state techniques". CACM 11 (Dec 1968), 805-813.

4. RAMOS, I., "Construcción de un traductor para un lenguaje especialmente adaptado a problemas físicos". Tesis Doctoral Fac. de Ciencias Universidad de Madrid. 1971.

(Total 11)

T A B L A I

SIMBOLO	SIGNIFICADO	SIMBOLO	SIGNIFICADO	SIMBOLO	SIGNIFICADO
0	0	41	R	82	EXIT
1	1	42	.CAT.	83	FIXED
2	2	43	$	84	FLOAT
3	3	44	*	85	GET
4	4	45	.GE.	86	GO
5	5	46	.GT.	87	GOTO
6	6	47	.NL.	88	I
7	7	48	blanco	89	IF
8	8	49	/	90	INITIAL
9	9	50	S	91	LABEL
10	..	51	T	92	LIST
11	=	52	U	93	LINE
12	'	53	V	94	MAIN
13	,.	54	W	95	OPTIONS
14	**	55	X	96	PAGE
15	TINT	56	Y	97	PROCEDURE
16	+	57	Z	98	PUT
17	A	58	.NG.	99	REAL
18	B	59	,	100	RECURSIVE
19	C	60	(	101	RETURN
20	D	61	.NE.	102	SKIP
21	E	62	.NOT.	103	STOP
22	F	63	.LE.	104	THEN
23	G	64	.LT.	105	TO
24	H	65	TERMID	106	WHILE
25	I	66	B	107	FORMAT
26	TERMSTRING	67	BI	108	REVERT
27	.	68	BEGIN	109	SIGNAL
28	)	69	BINARY	110	FREE
29	.EQ.	70	BIT	111	ALLOCATE
30	.OR.	71	BY	112	DELAY
31	.AND.	72	CALL	113	UNLOCK
32	-	73	CHARACTER	114	LOCATE
33	J	74	COMPLEX	115	READ
34	K	75	DECIMAL	116	REWRITE
35	L	76	DECLARE	117	WRITE
36	M	77	DO	118	CLOSE
37	N	78	E	119	OPEN
38	O	79	ELSE	120	RELEASE
39	P	80	END	121	ON
40	Q	81	ENTRY		

```
01000  10011  UNDFL  TCOB    *
02201  10000          WTCB    1
32125  10001          RCHB    COM4
01000  10011          TCOB    *
00     00010          CALL    EXIT
05000  10011
01002  10011
00222  10100
00103                 RUTINA  US          **********
03477  10001          SXA     RET+4,4
00104                 DECID1  LL1,(11,16,28,32,60)
00123                 DECID1  LL2,48
00126                 DECID1  LL3,(27,44,49,59)
00142                 DECID1  LL4,12
00145                 DECID1  LL5,(0,1,2,3,4,5,6,7,8,9)   DECINO  LL5
00203                 DECID1  LL39,65
00206                 DECID1  EXI,43
00211                 ENTRAR  PID
00212          LL1    ENTRAR  SP
00213          LL2    TEST1   48
00235                 ENTRAR  US+1
00236          LL3    ENTRAR  PSC
00237          LL4    ENTRAR  CCAR
00240          LL5    ENTRAR  TINT
00241          LL39   LEER
00246                 ENTRAR  US+1
00247                 VOLVER
00247                 RUTINA  SP          ********
00247                 DECID1  LL6,11
00252                 DECID1  LL7,16
00255                 DECID1  LL8,28
00260                 DECID1  LL9,32
00263                 DECID1  LL38,43
00266                 TEST1   60
00310                 ENTRAR  A1
00311          LL6    TEST1   11
00333                 ENTRAR  A1
00334          LL7    TEST1   16
00356                 ENTRAR  A1
00357          LL8    TEST1   28
00401                 ENTRAR  A1
00402          LL9    TEST1   32
00424                 ENTRAR  A1
00425          LL38   TEST1   43
00447                 ENTRAR  A1
00450                 VOLVER
00450                 RUTINA  PSC         ********
00450                 DECID1  LL10,27
00453                 DECID1  LL11,44
00456                 DECID1  LL12,49
00461                 TEST1   59
00503                 ENTRAR  SC2
00504          LL10   TEST1   27
00526                 ENTRAR  SC1
00527          LL11   TEST1   44
00551                 ENTRAR  SC3
```

TEXT.

```
00552          LL12    TEST1   49
00574                  ENTRAR  SC4
00575                  VOLVER
00575                  RUTINA  SC1             ********
00575                  DECID1  LL13,27
00600                  DECID1  LL14,(17,19,21,23,35,37,38)
00625                  ENTRAR  A1
00626          LL13    TEST1   27
00650                  PUT     10
00654                  ENTRAR  RET
00655          LL14    ENTRAR  OR
00656                  VOLVER
00656                  RUTINA  SC2             ********
00656                  DECID1  LL15,27
00661                  ENTRAR  A1
00662          LL15    TEST1   27
00704                  PUT     13
00710                  ENTRAR  RET
00711                  VOLVER
00711                  RUTINA  SC3             ********
00711                  DECID1  LL16,44
00714                  ENTRAR  A1
00715          LL16    TEST1   44
00737                  PUT     14
00743                  ENTRAR  RET
00744                  VOLVER
00744                  RUTINA  SC4             ********
00744                  DECID1  LL18,44
00747                  ENTRAR  A1
00750          LL18    TEST1   44
00772                  ENTRAR  COMENT
00773                  VOLVER
00773                  RUTINA  OR              ********
00773                  DECID1  LL19,17
00776                  DECID1  LL17,19
01001                  DECID1  LL21,21
01004                  DECID1  LL20,23
01007                  DECID1  LL22,35
01012                  DECID1  LL23,37
01015                  TEST1   38
01037                  TEST1   41
01061                  TEST1   27
01103                  PUT     30
01107                  ENTRAR  RET
01110          LL17    TEST1   19
01132                  TEST1   17
01154                  TEST1   51
01176                  TEST1   27
01220                  PUT     42
01224                  ENTRAR  RET
01225          LL19    TEST1   17
01247                  TEST1   37
01271                  TEST1   20
01313                  TEST1   27
01335                  PUT     31
01341                  ENTRAR  RET
```

TEXT.

```
01342          LL20    TEST1   23
01364                  ENTRAR  COR1
01365          LL21    TEST1   21
01407                  TEST1   40
01431                  TEST1   27
01453                  PUT     29
01457                  ENTRAR  RET
01460          LL22    TEST1   35
01502                  ENTRAR  COR3
01503          LL23    TEST1   37
01525                  ENTRAR  COR2
01526                  VOLVER
01526                  RUTINA  COR1            ********
01526                  DECID1  LL24,21
01531                  TEST1   51
01553                  TEST1   27
01575                  PUT     46
01601                  ENTRAR  RET
01602          LL24    TEST1   21
01624                  TEST1   27
01646                  PUT     45
01652                  ENTRAR  RET
01653                  VOLVER
01653                  RUTINA  COR2            ********
01653                  DECID1  LL25,35
01656                  DECID1  LL26,23
01661                  DECID1  LL27,21
01664                  TEST1   38
01706                  TEST1   51
01730                  TEST1   27
01752                  PUT     62
01756                  ENTRAR  RET
01757          LL25    TEST1   35
02001                  TEST1   27
02023                  PUT     47
02027                  ENTRAR  RET
02030          LL26    TEST1   23
02052                  TEST1   27
02074                  PUT     58
02100                  ENTRAR  RET
02101          LL27    TEST1   21
02123                  TEST1   27
02145                  PUT     61
02151                  ENTRAR  RET
02152                  VOLVER
02152                  RUTINA  COR3            ********
02152                  DECID1  LL28,21
02155                  TEST1   51
02177                  TEST1   27
02221                  PUT     64
02225                  ENTRAR  RET
02226          LL28    TEST1   21
02250                  TEST1   27
02272                  PUT     63
02276                  ENTRAR  RET
02277                  VOLVER
```

```
TEXT.

   02277                    RUTINA   TINT          ********
   02277                    LLAMAR   CIFRA
   02311                    ENTRAR   CICON
   02312                    VOLVER
   02312                    RUTINA   CICON          ********
   02312                    DECID1   LL29,(0,1,2,3,4,5,6,7,8,9,48)
   02353                    PUT      15
   02357                    ENTRAR   RET
   02360              LL29  LLAMAR   CIFRA1
   02372                    ENTRAR   CICON
   02373                    VOLVER
   02373                    RUTINA   CIFRA          ********

069
 0 15660  1CCC1             CLA      CARCO
 0 15657  1CCC1             STO      CARCO-1
   02375                    AVANZA   1
   02405                    SALIR
   02416                    VOLVER
   02416                    RUTINA   CIFRA1         ********
   02416                    DECID1   LL31,48

070
 0 15660  1CCC1             CLA      CARCO
 0 15657  1CCC1             STO      CARCO-1
   02423                    AVANZA   1
   02433                    SALIR
   02444              LL31  TEST1    48
   02466                    SALIR
   02477                    VOLVER
   02477                    RUTINA   CCAR          ********
   02477                    TEST1    12
   02521                    LLAMAR   SUBC
   02533                    TEST1    12
   02555                    PUT      26
   02561                    ENTRAR   RET
   02562                    VOLVER
   02562                    RUTINA   SUBC          ********
   02562                    DECID1   LL41,65
   02565                    DECID1   LL30,12
   02570                    AVANZA   1
   02600                    ENTRAR   SUBC
   02601              LL41  LEER
   02606                    ENTRAR   SUBC
   02607              LL30  SALIR
   02620                    VOLVER
   02620                    RUTINA   PID           ********

077
 0 31506  1CCC1             STZ      CLETRA
 0 15711  1CCC1             STZ      FLAG
   02622                    LLAMAR   LETRA
 1 00014  1CCC0             AXT      12,1

078
 0 32261  1CCC1             CLA      =060606C6C6C60
```

```
TEXT.

 1 31536  1CCC1          STO     VEC+12,1
 1 41001  1CC11          TIX     *-1,1,1
 1 00014  1CCCC          AXT     12,1
 0 15657  1CCC1          CLA     CARCO-1
 1 31536  1CCC1          STO     VEC+12,1
 1 01001  1CC11          TXI     *+1,1,-1
   02644                 ENTRAR  CPID
   02645                 VOLVER
   02645                 RUTINA  CPID          ********
   02645                 DECID1  LL32,(0,1,2,3,4,5,6,7,8,9)
   02703                 DECID1  LL33,(11,12,16,27,28,32,43,44,48,49,59,60)

181
 0 31506  1CCC1          CLA     CLETRA
 0 32176  1CCC1          ADD     =1

182
 0 31506  1CCC1          STO     CLETRA
   02752                 LLAMAR  LETRA
 0 15657  1CCC1          CLA     CARCO-1
 1 31536  1CCC1          STO     VEC+12,1
 1 01001  1CC11          TXI     *+1,1,-1
   02767                 ENTRAR  CPID
 0 32176  1CCC1  LL32    CLA     =1
 0 15711  1CCC1          STO     FLAG
 0 31506  1CCC1          CLA     CLETRA
 0 32176  1CCC1          ADD     =1

183
 0 31506  1CCC1          STO     CLETRA
   02775                 LLAMAR  CIFRA
 0 15657  1CCC1          CLA     CARCO-1
 1 31536  1CCC1          STO     VEC+12,1
 1 01001  1CC11          TXI     *+1,1,-1
   03012                 ENTRAR  CPID
   03013         LL33    ENTRAR  A2
   03014                 VOLVER
   03014                 RUTINA  LETRA         ********
   03014                 DECID1  LL35,(17,18,19,20,21,22,23,24,25)
   03047                 DECID1  LL35,(33,34,35,36,37,38,39,40,41)
   03102                 DECID1  LL35,(50,51,52,53,54,55,56,57)
   03132                 TEST1   0
   03154                 SALIR

189
 0 15660  1CCC1  LL35    CLA     CARCO
 0 15657  1CCC1          STO     CARCO-1
   03167                 AVANZA  1
   03177                 SALIR
   03210                 VOLVER
   03210                 RUTINA  COMENT        ********
   03210                 DECID1  LL40,65
   03213                 DECID1  LL36,44
   03216                 AVANZA  1
   03226                 ENTRAR  COMENT
   03227          LL36   TEST1   44
```

```
EXT.

   03251                    DECIC1  LL37,49
   03254                    ENTRAR  COMENT
   03255            LL37    TESTI   49
   03277                    ENTRAR  US+1
   03300            LL40    LEER
   03305                    ENTRAR  COMENT
   03306                    VOLVER
   03306                    RUTINA  A1              ********

193
 0 15664  1CCC1             CLA     CARCUR
 0 15663  1CCC1             STC     CARCUR-1
 0 15657  1CCC1             CLA     CARCO-1
 0 15664  1CCC1             STO     CARCUR
   03312                    ENTRAR  RET
   03313                    VOLVER
   03313                    RUTINA  A2              ********
 0 15711  1CCC1             CLA     FLAG
 0 03322  1CCC1             TZE     PPCL
   03315                    PUT     65
   03321                    ENTRAR  RET

194
 1 00002  1CCCC  PPCL       AXT     2,1
 2 00015  1CCCC             AXT     13,2
 0 32301  1CCC1             CLA     =7B17
 0 01005  1CC11             STD     *+5
 2 31537  1CCC1  LOOP       LDQ     VEC+13,2
 0 00035  1CCCC             RQL     29
 0 CCCC6  1CCCC             LLS     6
 2 01001  1CC11             TXI     *+1,2,-1
 2 03326  1CCC1             TXH     LOOP,2,7
 1 15710  1CCC1             SLW     LTR+2,1
 0 32302  1CCC1             CLA     =1B17
 0 41003  1CC11             STD     *-3
 1 03326  1CCC1             TIX     LOOP,1,1
 0 31506  1CCC1             CLA     CLETRA

195
 0 03364  1CCC1             TZE     LLL2
 2 31506  1CCC1             LAC     CLETRA,2
 2 31511  1CCC1             CLA     TABAUX-1,2
 0 31507  1CCC1             STA     COLON
 0 00022  1CCCC             ARS     18
 0 31510  1CCC1             STA     NPCL
   03346                    LLAMAR  SCAN
 0 31511  1CCC1             CLA     CODIGO
 0 32223  1CCC1             SUB     =65
 0 03431  1CCC1             TNZ     LLL6

196
 0 03431  1CCC1             TRA     LLL6
 0 15657  1CCC1  LLL2       CLA     CARCO-1
 0 32262  1CCC1             SUB     =18
 0 03376  1CCC1             TZE     LLL3
 0 32214  1CCC1             SUB     =3
```

```
XT.

 03404   10CC1           TZE     LLL4
 32215   10CC1           SUB     =4
 03412   10CC1           TZE     LLL5
 32223   10CC1   LLL10   CLA     =65
 31511   10CC1           STO     CODIGO
 03431   10CC1           TRA     LLL6
 15664   10CC1   LLL3    CLA     CARCUR
 32257   10CC1           SUB     =15
 03422   10CC1           TZE     LLL7
 32177   10CC1           SUB     =11
 03422   10CC1           TZE     LLL7
 03373   10CC1           TRA     LLL10
 15664   10CC1   LLL4    CLA     CARCUR
 32257   10CC1           SUB     =15

7
 03425   10CC1           TZE     LLL8
 32211   10CC1           SUB     =12
 03425   10CC1           TZE     LLL8
 03373   10CC1           TRA     LLL10
 15664   10CC1   LLL5    CLA     CARCUR
 32257   10CC1           SUB     =15
 03427   10CC1           TZE     LLL9
 32211   10CC1           SUB     =12
 03427   10CC1           TZE     LLL9
 32271   10CC1           SUB     =39
 03427   10CC1           TZE     LLL9
 03373   10CC1           TRA     LLL10
 32303   10CC1   LLL7    CLA     =66
 31511   10CC1           STO     CODIGO
 03431   10CC1           TRA     LLL6
 32304   10CC1   LLL8    CLA     =78
 03423   10CC1           TRA     LLL7+1
 32305   10CC1   LLL9    CLA     =88
 03423   10CC1           TRA     LLL7+1

8
 15664   10CC1   LLL6    CLA     CARCUR
 15663   10CC1           STO     CARCUR-1
 31511   10CC1           CLA     CODIGO
 15664   10CC1           STO     CARCUR
 03435                   ENTRAR  RET
 03436                   VOLVER
 03436                   RUTINA  SCAN            *********
 31507   10CC1           CLA     COLON
 31510   10CC1           ADD     NPCL
 32176   10CC1           SUB     =1
 01003   10011           STA     *+3
 32223   10CC1           CLA     =65
 31511   10CC1           STO     CODIGO
 00000   10CCC           AXC     **,4
 01001   10011           TXI     *+1,4,-1
 01013   10011           SXD     *+11,4
 31507   10CC1           LAC     COLON,1
 31656   10CC1           CLA     IIPPCL-1,1
 15706   10CC1           SUB     LTR
```

```
C 01006  1CC11          TNZ     *+6
1 31743  1CCC1          CLA     IIPPCL+52,1

99
C 15707  1CCC1          SUB     LTR+1
C 01003  1CC11          TNZ     *+3
1 32030  1CCC1          CLA     ICCD-1,1
C 31511  1CCC1          STO     CODIGO
1 01001  1CC11          TXI     *+1,1,-1
1 41011  1CC11          TXH     *-9,1,**
  03462                 SALIR
  03473                 VOLVER
  03473                 RUTINA  RET          *********
CCCO     CCC10          CALL    CLOCK(B)
4 04000  1CC11
C 01003  10011
C 04261  1C1CO

CO
C 32170  1CCC1
4 CCCCO  1CCCC          AXT     **,4
C 32164  1CCC1          CLA     S
C 32170  1CCC1          FAD     B
C 32167  1CCC1          FSB     A
C 32164  1CCC1          STO     S
4 CCCC1  1CCCO          TRA     1,4
  03505                 VOLVER
  03505                 RUTINA  PROGRA       ********
  03505                 LLAMAR  PROCCL
  03517                 ENTRAR  PROCON
  03520                 VOLVER
  03520                 RUTINA  PROCON       ********
  03520                 DECIDE  L1,(65,97)
  03526                 SALIR
  03537          L1     LLAMAR  PROCCL
  03551                 ENTRAR  PROCON
  03552                 VOLVER
  03552                 RUTINA  PROCCL       ********
  03552                 LLAMAR  HEADPA
  03564                 TEST    97
  03604                 LLAMAR  PPARAL
  03616                 LLAMAR  PROCOP
  03630                 TEST    13
  03650                 ENTRAR  SENLIS
  03651                 VOLVER
  03651                 RUTINA  HEADPA       ********
  03651                 DECIDE  L2,65
  03654                 SALIR
  03665          L2     LLAMAR  IDTHIN
  03677                 ENTRAR  AUX1
  037CO                 VOLVER
  03700                 RUTINA  AUX1         ********
  037CO                 DECIDE  L3,10
  03703                 ENTRAR  ASSTAT
  03704          L3     TEST    10
  03724                 ENTRAR  HEADPA
```

TEXT.

```
   03725                VOLVER
   03725                RUTINA  ASSTAT        ********
   03725                DECIDE  L4,59
   03730                TEST    11
   03750                ENTRAR  EXPRES
   03751          L4    TEST    59
   03771                LLAMAR  IDTHLI
   04003                TEST    11
   04023                ENTRAR  EXPRES
   04024                VOLVER
   04024                RUTINA  IDTHLI        ********
   04024                LLAMAR  IDTHIN
   04036                ENTRAR  AUX2
   04037                VOLVER
   04037                RUTINA  AUX2          ********
   04037                DECIDE  L5,59
   04042                SALIR
   04053          L5    TEST    59
   04073                ENTRAR  IDTHLI
   04074                VOLVER
   04074                RUTINA  PPARAL        ********
   04074                DECIDE  L6,60
   04077                SALIR
   04110          L6    ENTRAR  PARAML
   04111                VOLVER
   04111                RUTINA  PARAML        ********
   04111                TEST    60
   04131                LLAMAR  IDENTI
   04143                LLAMAR  IDENLI
   04155                TEST    28
   04175                SALIR
   04206                VOLVER
   04206                RUTINA  IDENLI        ********
   04206                DECIDE  L7,59
   04211                SALIR
   04222          L7    TEST    59
   04242                LLAMAR  IDENTI
   04254                ENTRAR  IDENLI
   04255                VOLVER
   04255                RUTINA  PROCOP        ********
   04255                DECIDE  L8,100
   04260                DECIDE  L9,95
   04263                SALIR
   04274          L9    TEST    95
   04314                TEST    60
   04334                TEST    94
   04354                TEST    28
   04374                SALIR
   04405          L8    TEST    100
   04425                SALIR
   04436                VOLVER
   04436                RUTINA  SENLIS        ********
   04436                LLAMAR  SENTEN
   04450                LLAMAR  SENCON
   04462                ENTRAR  ENDCLA
   04463                VOLVER
```

TEXT.

```
   04463                RUTINA  SENCON        ********
   04463                DECIDE  L10,80
   04466                LLAMAR  SENTEN
   04500                ENTRAR  SENCON
   04501          L10   SALIR
   04512                VOLVER
   04512                RUTINA  ENDCLA        ********
   04512                TEST    8C
   04532                LLAMAR  PIDENT
   04544                TEST    13
   04564                SALIR
   04575                VOLVER
   04575                RUTINA  PIDENT        ********
   04575                DECIDE  L11,65
   04600                SALIR
   04611          L11   ENTRAR  IDENTI
   04612                VOLVER
   04612                RUTINA  SENTEN        ********
   04612                LLAMAR  HEADPA
   04624                ENTRAR  SENT
   04625                VOLVER
   04625                RUTINA  SENT          ********
   04625                DECIDE  L12,13
   04630                DECIDE  L13,81
   04633                DECIDE  L14,97
   04636                DECIDE  L15,76
   04641                ENTRAR  STATEM
   04642          L15   TEST    76
   04662                LLAMAR  DECLAL
   04674                TEST    13
   04714                SALIR
   04725          L14   TEST    97
   04745                LLAMAR  PPARAL
   04757                LLAMAR  PROCOP
   04771                TEST    13
   05011                ENTRAR  SENLIS
   05012          L12   ENTRAR  NULSTA
   05013          L13   TEST    81
   05033                LLAMAR  PPARAL
   05045                TEST    13
   05065                SALIR
   05076                VOLVER
   05076                RUTINA  DECLAL        ********
   05076                LLAMAR  DECLAR
   05110                ENTRAR  DECCON
   05111                VOLVER
   05111                RUTINA  DECCON        ********
   05111                DECIDE  L16,59
   05114                SALIR
   05125          L16   TEST    59
   05145                LLAMAR  DECLAR
   05157                ENTRAR  DECCON
   05160                VOLVER
   05160                RUTINA  DECLAR        ********
   05160                DECIDE  L17,60
   05163                LLAMAR  IDENTI
```

EXT.

```
05175                ENTRAR  ATTLIS
05176          L17   TEST    60
05216                LLAMAR  DECLAL
05230                TEST    28
05250                ENTRAR  ATTLIS
05251                VOLVER
05251                RUTINA  ATTLIS          ********
05251                LLAMAR  DIM1
05263                ENTRAR  ATTCON
05264                VOLVER
05264                RUTINA  DIM1            ********
05264                DECIDE  L18,60
05267                SALIR
05300          L18   TEST    60
05320                LLAMAR  BOUND1
05332                TEST    28
05352                SALIR
05363                VOLVER
05363                RUTINA  BOUND1          ********
05363                LLAMAR  BOUN1
05375                ENTRAR  BOUNCO
05376                VOLVER
05376                RUTINA  BOUNCO          ********
05376                DECIDE  L19,59
05401                SALIR
05412          L19   TEST    59
05432                ENTRAR  BOUND1
05433                VOLVER
05433                RUTINA  BOUN1           ********
05433                DECIDE  L20,44
05436                LLAMAR  EXPRES
05450                ENTRAR  AUX3
05451          L20   TEST    44
05471                SALIR
05502                VOLVER
05502                RUTINA  AUX3            ********
05502                DECIDE  L21,10
05505                SALIR
05516          L21   TEST    10
05536                ENTRAR  EXPRES
05537                VOLVER
05537                RUTINA  ATTCON          ********
05537                DECIDE  L22,(59,28,13)
05550                LLAMAR  ATT
05562                ENTRAR  ATTCON
05563          L22   SALIR
05574                VOLVER
05574                RUTINA  ATT             ********
05574                DECIDE  L23,(73,70)
05602                DECIDE  L24,81
05605                DECIDE  L25,91
05610                DECIDE  L26,90
05613                ENTRAR  ARIATT
05614          L26   ENTRAR  INIATT
05615          L25   TEST    91
05635                ENTRAR  AFTLAB
```

EXT.

```
  05636          L24     TEST    81
  05656                  SALIR
  05667          L23     LLAMAR  STRATT
  05701                  TEST    60
  05721                  LLAMAR  LENGTH
  05733                  TEST    28
  05753                  SALIR
  05764                  VOLVER
  05764                  RUTINA  ARIATT          ********
  05764                  LLAMAR  ARITH
  05776                  ENTRAR  ARICON
  05777                  VOLVER
  05777                  RUTINA  ARICON          ********
  05777                  DECIDE  L27,60
  06002                  SALIR
  06013          L27     ENTRAR  PRECIS
  06014                  VOLVER
  06014                  RUTINA  ARITH           ********
  06014                  DECIDE  L28,74
  06017                  DECIDE  L29,75
  06022                  DECIDE  L30,99
  06025                  DECIDE  L31,83
  06030                  DECIDE  L32,84
  06033                  TEST    69
  06053                  SALIR
  06064          L32     TEST    84
  06104                  SALIR
  06115          L31     TEST    83
  06135                  SALIR
  06146          L30     TEST    99
  06166                  SALIR
  06177          L29     TEST    75
  06217                  SALIR
  06230          L28     TEST    74
  06250                  SALIR
  06261                  VOLVER
  06261                  RUTINA  PRECIS          ********
  06261                  TEST    60
  06301                  TEST    15
  06321                  ENTRAR  AUX4
  06322                  VOLVER
  06322                  RUTINA  AUX4            ********
  06322                  DECIDE  L33,59
  06325                  TEST    28
  06345                  SALIR
  06356          L33     TEST    59
  06376                  LLAMAR  POSSIC
  06410                  TEST    15
  06430                  TEST    28
  06450                  SALIR
  06461                  VOLVER
  06461                  RUTINA  AFTLAB          ********
  06461                  ENTRAR  PPARAL
  06462                  VOLVER
  06462                  RUTINA  STRATT          ********
  06462                  DECIDE  L34,70
```

EXT.

```
06465                TEST    73
06505                SALIR
06516          L34   TEST    70
06536                SALIR
06547                VOLVER
06547                RUTINA  LENGTH        ********
06547                DECIDE  L35,44
06552                ENTRAR  EXPRES
06553          L35   TEST    44
06573                SALIR
06604                VOLVER
06604                RUTINA  INIATT        ********
06604                TEST    9C
06624                ENTRAR  INIPAR
06625                VOLVER
06625                RUTINA  INIPAR        ********
06625                TEST    60
06645                LLAMAR  INILIS
06657                TEST    28
06677                SALIR
06710                VOLVER
06710                RUTINA  INILIS        ********
06710                LLAMAR  INIT1
06722                ENTRAR  INICON
06723                VOLVER
06723                RUTINA  INICON        ********
06723                DECIDE  L118,59
06726                SALIR
06737          L118  TEST    59
06757                ENTRAR  INILIS
06760                VOLVER
06760                RUTINA  INIT1         ********
06760                DECIDE  L119,60
06763                DECIDE  L120,44
06766                ENTRAR  INIT3
06767          L120  TEST    44
07007                SALIR
07020          L119  TEST    60
07040                TEST    15
07060                TEST    28
07100                ENTRAR  INIT2
07101                VOLVER
07101                RUTINA  INIT2         ********
07101                DECIDE  L121,60
07104                ENTRAR  INIT3
07105          L121  ENTRAR  INIPAR
07106                VOLVER
07106                RUTINA  INIT3         ********
07106                DECIDE  L122,65
07111                DECIDE  L123,26
07114                LLAMAR  POSSIG
07126                ENTRAR  NUMBER
07127          L123  ENTRAR  CHASTR
07130          L122  ENTRAR  IDENTI
07131                VOLVER
07131                RUTINA  CHASTR        ********
```

TEXT.

```
07131                TEST    26
07151                ENTRAR  MAYBEB
07152                VOLVER
07152                RUTINA  MAYBEB          ********
07152                DECIDE  L4C,66
07155                SALIR
07166          L4C   TEST    66
07206                SALIR
07217                VOLVER
07217                RUTINA  STATEM          ********
07217                DECIDE  L41,89
07222                ENTRAR  UNCOND
07223          L41   ENTRAR  IFSTAT
07224                VOLVER
07224                RUTINA  IFSTAT          ********
07224                LLAMAR  IFCLAU
07236                LLAMAR  ST
07250                DECIDE  L129,79
07253                SALIR
07264          L129  TEST    79
07304                ENTRAR  IFCONT
07305                VOLVER
07305                RUTINA  IFCONT          ********
07305                DECIDE  L42,13
07310                ENTRAR  ST
07311          L42   TEST    13
07331                SALIR
07342                VOLVER
07342                RUTINA  ST              ********
07342                DECIDE  LL,65
07345                ENTRAR  STATEM
07346          LL    LLAMAR  HEADPA
07360                TEST    13
07400                SALIR
07411                VOLVER
07411                RUTINA  IFCLAU          ********
07411                TEST    89
07431                LLAMAR  EXPRES
07443                TEST    104
07463                SALIR
07474                VOLVER
07474                RUTINA  UNCOND          ********
07474                DECIDE  L43,68
07477                DECIDE  L44,77
07502                DECIDE  L45,(98,85)
07510                DECIDE  L46,(86,87)
07516                DECIDE  L47,103
07521                DECIDE  L48,72
07524                DECIDE  L49,82
07527                DECIDE  L50,101
07532                ENTRAR  NCIMST
07533          L50   ENTRAR  RETSTA
07534          L49   ENTRAR  EXISTA
07535          L48   ENTRAR  CALSTA
07536          L47   ENTRAR  STOSTA
07537          L46   ENTRAR  GOTOST
```

TEXT.

```
07540          L45     ENTRAR  STRIOS
07541          L44     ENTRAR  GROUP
07542          L43     ENTRAR  BEGBLO
07543                  VOLVER
07543                  RUTINA  NULSTA          ********
07543                  TEST    13
07563                  SALIR
07574                  VOLVER
07574                  RUTINA  BEGBLO          ********
07574                  TEST    68
07614                  TEST    13
07634                  ENTRAR  SENLIS
07635                  VOLVER
07635                  RUTINA  GROUP           ********
07635                  TEST    77
07655                  LLAMAR  DOCON
07667                  TEST    13
07707                  ENTRAR  SENLIS
07710                  VOLVER
07710                  RUTINA  DOCON           ********
07710                  DECIDE  L51,106
07713                  DECIDE  L52,65
07716                  SALIR
07727          L52     LLAMAR  IDTHIN
07741                  TEST    11
07761                  ENTRAR  SPELIS
07762          L51     TEST    106
10002                  TEST    60
10022                  LLAMAR  EXPRES
10034                  TEST    28
10054                  SALIR
10065                  VOLVER
10065                  RUTINA  SPELIS          ********
10065                  LLAMAR  SPEC
10077                  ENTRAR  RESSPE
10100                  VOLVER
10100                  RUTINA  RESSPE          ********
10100                  DECIDE  L53,59
10103                  SALIR
10114          L53     TEST    59
10134                  ENTRAR  SPELIS
10135                  VOLVER
10135                  RUTINA  SPEC            ********
10135                  LLAMAR  EXPRES
10147                  LLAMAR  STEPAR
10161                  ENTRAR  WHIPAR
10162                  VOLVER
10162                  RUTINA  STEPAR          ********
10162                  DECIDE  L54,105
10165                  DECIDE  L55,71
10170                  SALIR
10201          L55     TEST    71
10221                  ENTRAR  AUX6
10222          L54     TEST    105
10242                  ENTRAR  AUX5
10243                  VOLVER
```

EXT.

```
10243                RUTINA  AUX6          ********
10243                LLAMAR  EXPRES
10255                DECIDE  L56,(106,59,13)
10266                TEST    105
10306                ENTRAR  EXPRES
10307          L56   SALIR
10320                VOLVER
10320                RUTINA  AUX5          ********
10320                LLAMAR  EXPRES
10332                DECIDE  L57,(106,59,13)
10343                TEST    71
10363                ENTRAR  EXPRES
10364          L57   SALIR
10375                VOLVER
10375                RUTINA  WHIPAR        ********
10375                DECIDE  L58,106
10400                DECIDE  L115,(16,32,62,60,26,15,27,65)
10430                SALIR
10441          L58   TEST    106
10461                TEST    60
10501                LLAMAR  EXPRES
10513                TEST    28
10533                SALIR
10544          L115  LLAMAR  EXPRES
10556                DECIDE  L58,106
10561                SALIR
10572                VOLVER
10572                RUTINA  GOTOST        ********
10572                LLAMAR  PGOTO
10604                LLAMAR  IDTHIN
10616                TEST    13
10636                SALIR
10647                VOLVER
10647                RUTINA  PGOTO         ********
10647                DECIDE  L59,87
10652                TEST    86
10672                TEST    105
10712                SALIR
10723          L59   TEST    87
10743                SALIR
10754                VOLVER
10754                RUTINA  CALSTA        ********
10754                TEST    72
10774                LLAMAR  IDENTI
11006                DECIDE  L130,13
11011                TEST    60
11031                LLAMAR  EXPRES
11043                LLAMAR  EXPRCT
11055                TEST    28
11075          L130  TEST    13
11115                SALIR
11126                VOLVER
11126                RUTINA  RETSTA        ********
11126                TEST    101
11146                ENTRAR  AUX7
11147                VOLVER
```

EXT.

```
11147                RUTINA  AUX7          ********
11147                DECIDE  L60,60
11152                TEST    13
11172                SALIR
11203          L60   TEST    60
11223                LLAMAR  EXPRES
11235                TEST    28
11255                TEST    13
11275                SALIR
11306                VOLVER
11306                RUTINA  EXISTA        ********
11306                TEST    82
11326                TEST    13
11346                SALIR
11357                VOLVER
11357                RUTINA  STOSTA        ********
11357                TEST    103
11377                TEST    13
11417                SALIR
11430                VOLVER
11430                RUTINA  STRIOS        ********
11430                DECIDE  L61,98
11433                TEST    85
11453                TEST    92
11473                TEST    60
11513                LLAMAR  GETLIS
11525                TEST    28
11545                TEST    13
11565                SALIR
11576          L61   TEST    98
11616                LLAMAR  PUTTHI
11630                TEST    92
11650                TEST    60
11670                LLAMAR  PUTLIS
11702                TEST    28
11722                TEST    13
11742                SALIR
11753                VOLVER
11753                RUTINA  PUTTHI        ********
11753                DECIDE  L124,96
11756                DECIDE  L125,93
11761                DECIDE  L126,102
11764                SALIR
11775          L126  TEST    102
12015                ENTRAR  SKINUM
12016          L125  TEST    93
12036                ENTRAR  LINNUM
12037          L124  TEST    96
12057                DECIDE  L125,93
12062                SALIR
12073                VOLVER
12073                RUTINA  LINNUM        ********
12073                DECIDE  L127,60
12076                SALIR
12107          L127  TEST    60
12127                LLAMAR  EXPRES
```

EXT.

```
12141                TEST    28
12161                SALIR
12172                VOLVER
12172                RUTINA  SKINUM          ********
12172                DECIDE  L128,60
12175                SALIR
12206          L128  TEST    60
12226                TEST    15
12246                TEST    28
12266                SALIR
12277                VOLVER
12277                RUTINA  PUTLIS          ********
12277                LLAMAR  EXPRES
12311                ENTRAR  EXPRCT
12312                VOLVER
12312                RUTINA  GETLIS          ********
12312                LLAMAR  IDTHIN
12324                TEST    59
12344                ENTRAR  IDTHLI
12345                VOLVER
12345                RUTINA  EXPRCT          ********
12345                DECIDE  L62,59
12350                SALIR
12361          L62   TEST    59
12401                LLAMAR  EXPRES
12413                ENTRAR  EXPRCT
12414                VOLVER
12414                RUTINA  NOIMST          ********
12414                DECIDE  L63,121
12417                DECIDE  L64,109
12422                DECIDE  L65,120
12425                DECIDE  L66,110
12430                DECIDE  L67,107
12433                DECIDE  L68,111
12436                DECIDE  L69,119
12441                DECIDE  L70,112
12444                DECIDE  L71,118
12447                DECIDE  L72,113
12452                DECIDE  L73,117
12455                DECIDE  L74,114
12460                DECIDE  L75,116
12463                DECIDE  L76,115
12466                TEST    108
12506                SALIR
12517          L76   TEST    115
12537                SALIR
12550          L75   TEST    116
12570                SALIR
12601          L74   TEST    114
12621                SALIR
12632          L73   TEST    117
12652                SALIR
12663          L72   TEST    113
12703                SALIR
12714          L71   TEST    118
12734                SALIR
```

EXT.

```
12745          L70     TEST    112
12765                  SALIR
12776          L69     TEST    119
13016                  SALIR
13027          L68     TEST    111
13047                  SALIR
13060          L67     TEST    107
13100                  SALIR
13111          L66     TEST    110
13131                  SALIR
13142          L65     TEST    120
13162                  SALIR
13173          L64     TEST    109
13213                  SALIR
13224          L63     TEST    121
13244                  SALIR
13255                  VOLVER
13255                  RUTINA  EXPRES          ********
13255                  LLAMAR  EXPF
13267                  ENTRAR  EXPX
13270                  VOLVER
13270                  RUTINA  EXPX            ********
13270                  DECIDE  L77,30
13273                  SALIR
13304          L77     TEST    30
13324                  LLAMAR  EXPF
13336                  ENTRAR  EXPX
13337                  VOLVER
13337                  RUTINA  EXPF            ********
13337                  LLAMAR  EXPE
13351                  ENTRAR  EXPFX
13352                  VOLVER
13352                  RUTINA  EXPFX           ********
13352                  DECIDE  L78,31
13355                  SALIR
13366          L78     TEST    31
13406                  LLAMAR  EXPE
13420                  ENTRAR  EXPFX
13421                  VOLVER
13421                  RUTINA  EXPE            ********
13421                  LLAMAR  EXPD
13433                  ENTRAR  EXPEX
13434                  VOLVER
13434                  RUTINA  EXPEX           ********
13434                  DECIDE  L79,64
13437                  DECIDE  L80,58
13442                  DECIDE  L81,63
13445                  DECIDE  L82,47
13450                  DECIDE  L83,45
13453                  DECIDE  L84,29
13456                  DECIDE  L85,46
13461                  DECIDE  L86,61
13464                  SALIR
13475          L86     TEST    61
13515                  LLAMAR  EXPD
13527                  ENTRAR  EXPEX
```

EXT.

```
13530          L85     TEST    46
13550                  LLAMAR  EXPD
13562                  ENTRAR  EXPEX
13563          L84     TEST    29
13603                  LLAMAR  EXPD
13615                  ENTRAR  EXPEX
13616          L83     TEST    45
13636                  LLAMAR  EXPD
13650                  ENTRAR  EXPEX
13651          L82     TEST    47
13671                  LLAMAR  EXPD
13703                  ENTRAR  EXPEX
13704          L81     TEST    63
13724                  LLAMAR  EXPD
13736                  ENTRAR  EXPEX
13737          L80     TEST    58
13757                  LLAMAR  EXPD
13771                  ENTRAR  EXPEX
13772          L79     TEST    64
14012                  LLAMAR  EXPD
14024                  ENTRAR  EXPEX
14025                  VOLVER
14025                  RUTINA  EXPD          ********
14025                  LLAMAR  EXPC
14037                  ENTRAR  EXPDX
14040                  VOLVER
14040                  RUTINA  EXPDX         ********
14040                  DECIDE  L87,42
14043                  SALIR
14054          L87     TEST    42
14074                  LLAMAR  EXPC
14106                  ENTRAR  EXPDX
14107                  VOLVER
14107                  RUTINA  EXPC          ********
14107                  LLAMAR  EXPB
14121                  ENTRAR  EXPCX
14122                  VOLVER
14122                  RUTINA  EXPCX         ********
14122                  DECIDE  L88,16
14125                  DECIDE  L89,32
14130                  SALIR
14141          L89     TEST    32
14161                  LLAMAR  EXPB
14173                  ENTRAR  EXPCX
14174          L88     TEST    16
14214                  LLAMAR  EXPB
14226                  ENTRAR  EXPCX
14227                  VOLVER
14227                  RUTINA  EXPB          ********
14227                  LLAMAR  EXPA
14241                  ENTRAR  EXPBX
14242                  VOLVER
14242                  RUTINA  EXPBX         ********
14242                  DECIDE  L9C,44
14245                  DECIDE  L91,49
14250                  SALIR
```

XT.

```
14261          L91     TEST    49
14301                  LLAMAR  EXPA
14313                  ENTRAR  EXPBX
14314          L90     TEST    44
14334                  LLAMAR  EXPA
14346                  ENTRAR  EXPBX
14347                  VOLVER
14347                  RUTINA  EXPA            ********
14347                  DECIDE  L92,62
14352                  DECIDE  L93,16
14355                  DECIDE  L94,32
14360                  LLAMAR  PRIMAR
14372                  ENTRAR  AUX8
14373          L94     TEST    32
14413                  ENTRAR  EXPA
14414          L93     TEST    16
14434                  ENTRAR  EXPA
14435          L92     TEST    62
14455                  ENTRAR  EXPA
14456                  VOLVER
14456                  RUTINA  AUX8            ********
14456                  DECIDE  L95,14
14461                  SALIR
14472          L95     TEST    14
14512                  ENTRAR  EXPA
14513                  VOLVER
14513                  RUTINA  PRIMAR          ********
14513                  DECIDE  L96,60
14516                  ENTRAR  NOBRPR
14517          L96     TEST    60
14537                  LLAMAR  EXPF
14551                  LLAMAR  EXPX
14563                  TEST    28
14603                  ENTRAR  PRICON
14604                  VOLVER
14604                  RUTINA  PRICON          ********
14604                  DECIDE  L97,26
14607                  SALIR
14620          L97     ENTRAR  CHASTR
14621                  VOLVER
14621                  RUTINA  NOBRPR          ********
14621                  DECIDE  L98,26
14624                  DECIDE  L99,65
14627                  ENTRAR  NUMBER
14630          L99     ENTRAR  IDTHIN
14631          L98     ENTRAR  CHASTR
14632                  VOLVER
14632                  RUTINA  IDTHIN          ********
14632                  LLAMAR  IDENTI
14644                  ENTRAR  IDCON1
14645                  VOLVER
14645                  RUTINA  IDCON1          ********
14645                  DECIDE  L101,60
14650                  SALIR
14661          L101    TEST    60
14701                  LLAMAR  SUBLIS
```

EXT.

```
14713                TEST    28
14733                SALIR
14744                VOLVER
14744                RUTINA  SUBLIS          ********
14744                LLAMAR  EXPRES
14756                ENTRAR  EXPRCT
14757                VOLVER
14757                RUTINA  IDENTI          ********
14757                TEST    65
14777                SALIR
15010                VOLVER
15010                RUTINA  NUMBER          ********
15010                DECIDE  L106,27
15013                TEST    15
15033                ENTRAR  AUX9
15034   L106         TEST    27
15054                TEST    15
15074                ENTRAR  POSSB
15075                VOLVER
15075                RUTINA  AUX9            ********
15075                DECIDE  L107,27
15100                ENTRAR  POSSB
15101   L107         TEST    27
15121                ENTRAR  AUX10
15122                VOLVER
15122                RUTINA  AUX10           ********
15122                DECIDE  L108,15
15125                ENTRAR  POSSB
15126   L108         TEST    15
15146                ENTRAR  POSSB
15147                VOLVER
15147                RUTINA  POSSB           ********
15147                DECIDE  L109,78
15152                ENTRAR  BTHING
15153   L109         TEST    78
15173                LLAMAR  POSSIG
15205                TEST    15
15225                ENTRAR  BTHING
15226                VOLVER
15226                RUTINA  BTHING          ********
15226                DECIDE  L110,67
15231                DECIDE  L111,88
15234                DECIDE  L116,66
15237                SALIR
15250   L116         TEST    66
15270                ENTRAR  AUX11
15271   L111         TEST    88
15311                SALIR
15322   L110         TEST    67
15342                SALIR
15353                VOLVER
15353                RUTINA  AUX11           ********
15353                DECIDE  L112,88
15356                SALIR
15367   L112         TEST    88
15407                SALIR
```

```
EXT.

   15420                        VOLVER
   15420                        RUTINA  POSSIC          ********
   15420                        DECIDE  L113,16
   15423                        DECIDE  L114,32
   15426                        SALIR
   15437                L114    TEST    32
   15457                        SALIR
   15470                L113    TEST    16
   15510                        SALIR
   15521                        VOLVER
   15521                        RUTINA  LECT            * * *

79
4 15570   1CCC1              SXA     IN,4
CC00      CCC10              CALL    CLOCK(S3)
4 04000   1CC11
C 01003   1CC11
C 17323   101C0
C 32171   1CCC1
C 01000   1CC11              TCOA    *
C 01202   1CCCC              RTDA    2
C 32132   1CCC1              RCHA    COM18
C 01000   1CC11              TCOA    *

80
C 00042   1CCC1              TEFA    EXI
C 01000   1CC11              TCOB    *
C 02201   1CCCC              WTDB    1
C 32130   100C1              RCHB    COM17
C 01000   1CC11              TCOB    *
CC00      CCC10              CALL    CLOCK(S4)
4 04CC0   1CC11
C 01003   1CC11
C 17335   101C0
C 32172   1CCC1
C 32173   1CCC1              CLA     S5
C 32172   1CCC1              FAD     S4
C 32171   1CCC1              FSB     S3
C 32173   1CCC1              STO     S5
1 00120   1CCCC              AXT     80,1
2 00015   1CCCC              AXT     13,2
4 00006   1CCCC              AXT     6,4
2 15702   1CCC1              LDQ     BUFF+13,2
C 32212   1CCC1              CLA     =0

81
C 00006   1CCCC              LGL     6
1 31656   1CCC1              STO     IBUF+80,1
1 01001   1CC11              TXI     *+1,1,-1
4 41004   100 11             TIX     *-4,4,1
2 41007   1CC11              TIX     *-7,2,1
4 00002   1CCCC              AXT     2,4
C 15702   1CCC1              LDQ     BUFF+13
C 32212   1CCC1              CLA     =0
C 00006   1CCCC              LGL     6
1 31656   1CCC1              STO     IBUF+80,1
```

UN EJEMPLO DE RECURSIVIDAD: HANOI

Por I. Ramos

1- INTRODUCCION

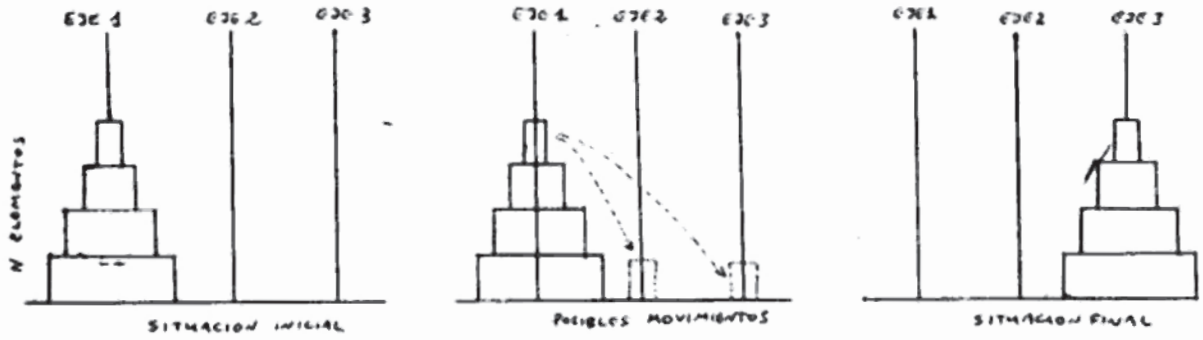
El juego de las "Torres de Hanoi" se presta a una fácil ejemplificación de los procesos recursivos. Se dice que ciertos monjes del Tíber una torre de 60 elementos de un sitio a otro y que cuando hayan terminado de trasladarla, el mundo terminará. Como veremos a continuación esto no es motivo de preocupación para nadie dada la imposibilidad temporal de realizarlo.

Explicación del juego: Se dispone de N piezas circulares de radio decreciente insertadas en un eje en forma de tronco de cono. Tenemos además otros dos ejes en que insertar las piezas. El fin del juego es trasladar la torre inicial de un eje a otro teniendo en cuenta que:

1) En cada movimiento solo movemos una pieza de un eje a otro (la única accesible en la cumbre de una torre).

2) Nunca puede haber una pieza mayor sobre otra menor en un mismo eje.

Veamos una figura:



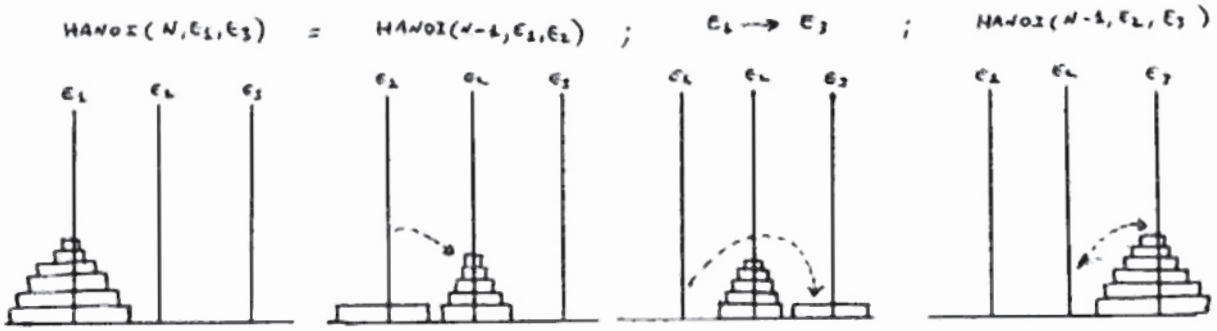
Veamos a continuación una solución trivial al problema y fácilmente implementable sobre ordenador.

2- ALGORITMO PROPUESTO

Representaremos la acción de trasladar N elementos desde un eje: E_1 a otro E_3 mediante la función de tres argumentos: HANOI (N,E_1,E_3). El algoritmo está basado en la siguiente afirmación: "El trasladar una torre de N elementos de E_1 a E_3 es equivalente a trasladar una torre -

Ramos, I. Un ejemplo de recursividad: HANOI. Boletín CCUM. nº 24. 1974.

de N-1 elementos de E_1 a E_2 seguido de trasladar 1 pieza de E_1 a E_3 - y seguido de trasladar una torre de N-1 elementos de E_2 a E_3" o gráficamente:



O sea:

$$HANOI\ (N,E_1,E_3) = HANOI\ (N-1,E_1,E_2)$$
$$.E_1 \longrightarrow E_3$$
$$.\ .\ HANOI\ (N-1,E_2,E_3)$$

Si codificamos los ejes por 1,2,3 , dados dos cualesquiera de ellos: x,y (origen y fin) el intermedio será 6-x-y, con lo que la función de traslado (que llamamos HANOI) entre dos ejes cualesquiera será:

$$HANOI\ (N,x,y) = HANOI\ (N-1,x,6-x-y)$$
$$x \longrightarrow y$$
$$HANOI\ (N-1,6-x-y,y)$$

lo que acepta una rápida implementación en ALGOL-60 en forma de procedimiento recursivo:

```
Procedure  HANOI (N,x,y);
     begin  HANOI (N-1,x,6-x-y);
            WRITE (x,y);
            HANOI (N-1,6-x-y,y);
     end
```

con lo que una forma de obtener impresos los movimientos sucesivos - sería llamar a esta procedure con los parámetros adecuados

```
HANOI (4,1,3);
```

Desarrollemos este ejemplo.

			MOVIMIENTOS
HANOI (3,1,3)	HANOI (2,1,2)	HANOI (1,1,3)	1-3
		1 ⟶ 2	1-2
		HANOI (1,3,2)	3-2
	1 — 3		1-3
	HANOI (2,2,3)	HANOI (1,2,1)	2-1
		2 ⟶ 3	2-3
		HANOI (1,1,3)	1-3

Donde hacemos la hipótesis obvia de que HANOI (1,x,y) = x-y ya que trasladar una torre de altura 1 de x a y es realizar el movimiento de 1 pieza de x a y. Esta afirmación es equivalente a suponer que HANOI (0,x,y) es la acción nula puesto que:

HANOI (1,x,y) =	HANOI (0,x,6-x-y)	nada
	x ⟶ y	x-y
	HANOI (0,6-x-y,y)	nada

que es la solución que hemos tomado nosotros en ALGOL:

```
begin  procedure  HANOI (N,x,y); value N,x,y; integer N,x,y;
                 begin if N ≠ 0 then begin HANOI (N-1,x,6-x-y);
                                           ESCRIBIR (x,y,N);
                                           HANOI (N-1,6-x-y-y);
                                       end

                 end

          HANOI (3,1,3) :

end
```

este programa produciría la salida dada en el ejemplo anterior.

3- IMPLEMENTACION EN ALGOL

La implementación en Algol es inmediata. Nosotros por motivos - de una presentación más cuidada de la salida de la máquina, así como - por obtener ejemplos para distintos valores de N hemos añadido más cosas al programa. Una versión de éste la damos a continuación

PROGRAMA

```
HAN                                                        10/26/73
                    ALCOR-ILLINOIS 7090 COMPILER

 1          'BEGIN' 'INTEGER' I,M.,'INTEGER''ARRAY' LX,LY(/1..10/).,
 2                   'INTEGER' K,MM.,
 3                   'PROCEDURE' HANOI(N,X,Y).,'INTEGER' N,X,Y.,
 4                                             'VALUE' N,X,Y.,
 5                  'BEGIN' 'IF' N 'NQ' 0 'THEN'
 6             'BEGIN'           HANOI(N-1,X,6-X-Y).,
 7                               ESCRIBIR(X,Y,N).,
 8                               HANOI(N-1,6-X-Y,Y).,
 9             'END'.,
10                  'END'.,
11                   'PROCEDURE' ESCRIBIR(X,Y,N).,'INTEGER' N,X,Y.,
12                                'VALUE' N,X,Y.,
13                   'BEGIN' 'INTEGER'K.,      M.=M+1., MM.=MM+1.,
14                       'IF' MM 'EQ' 2**I-1'OR' M'EQ' 10 'THEN' 'BEGIN'
15                               LX(/M/).=X ., LY(/M/).=Y .,
16                               FORMAT(6,'('10X,10(F2.0,1H-,F2.0,3X)')').,
17                               PRINTF(LX(/1/),LY(/1/),LX(/2/),LY(/2/),
18                                      LX(/3/),LY(/3/),LX(/4/),LY(/4/),
19                                      LX(/5/),LY(/5/),LX(/6/),LY(/6/),
20                                      LX(/7/),LY(/7/),LX(/8/),LY(/8/),
21                                      LX(/9/),LY(/9/),LX(/10/),LY(/10/)).,
22                               M.=0.,
23                               'FOR' K.=1'STEP' 1 'UNTIL'10 'DO'
24                               LX(/K/).=LY(/K/).=0.,
25                                                        'END'
26                                                    'ELSE''BEGIN'
27                               LX(/M/).=X ., LY(/M/).=Y .,'END'
28                   'END'.,
29                  'FOR' I.=2 'STEP' 1 'UNTIL'10 'DO' 'BEGIN'
30                  FORMAT(6,'('1H1,50X,14H*** HANOI N = ,F2.0  ,4H ***,')').,
31                  PRINTF(I).,
32                  FORMAT(6,'('///,10X,17HPUNTO DE SALIDA =,F2.0
33                                 ,18HPUNTO DE LLEGADA =,F2.0/
34                  10X,35H-----------------------------------')').,
35                  PRINTF( 1, 3).,
36                       MM.=M.=0.,  'FOR' K.=1 'STEP'1 'UNTIL' 10 'DO'
37                           LX(/K/).=LY(/K/).=0.,
38                  HANOI(I, 1, 3).,                      'END'.,
39             'END'.,
40             'FINIS'
```

Y la salida para N = 9 , E_1 = 1 , E_3 = 3 , es la siguiente:

SALIDA nº 9

*** HANOI N = 9. ***

PUNTO DE SALIDA =1.PUNTO DE LLEGADA =3.

1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.
3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.
1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.
2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.
3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.
3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.
3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	3.-2.
1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	1.-2.
3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.
2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.
2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.
2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.
1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.
2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	3.-2.	1.-3.	1.-2.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.
2.-1.	2.-3.	1.-3.	2.-1.	3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	3.-1.
2.-1.	3.-2.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.	1.-3.	2.-1.
3.-2.	3.-1.	2.-1.	2.-3.	1.-3.	1.-2.	3.-2.	1.-3.	2.-1.	2.-3.
1.-3.	0.-0.	0.-0.	0.-0.	0.-0.	0.-0.	0.-0.	0.-0.	0.-0.	0.-0.

4- CONCLUSIONES

El número de movimientos que se realizan es función de N. Concretamente 2^n-1. Lo que se obtiene fácilmente sin más que considerar el árbol de generación de movimientos dado en el ejemplo. En el caso de los monjes Tibetanos el nº de movimientos es pues $2^{60}-1 \quad 10^{18}$. Es pues una leyenda tranquilizadora.

Existen muchas soluciones a este problema clásico en Teoría de Juegos, la propuesta creemos que es breve y elegante.

Debo agradecer a C. Pair el haberme sensibilizado a este problema.

SISTEMA DE CREACION Y MANTENIMIENTO DE BANCOS DE INFORMACION

(PARTE I)

POR

ISIDRO RAMOS SALAVERT

CALIXTO JORGE LOZANO PUEYO

ALFONSO MUÑOZ HERRERO

PUBLICADO EN

REVISTA DE AUTOMATICA

AÑO VI - NUM. 22

Octubre-noviembre-diciembre 1974

Ramos, I., Lozano, C. Muñoz, A. Sistema de Creación y Mantenimiento de Bancos de Información. Revista de Automática, 1974 (Parte I), 1975 (Partes II y III).

SISTEMA DE CREACION Y MANTENIMIENTO [1] DE BANCOS DE INFORMACION [2]

(PARTE I)

Por Isidro RAMOS SALAVERT (*)
Calixto Jorge LOZANO PUEYO (**)
Alfonso MUÑOZ HERRERO (**)

ABSTRACT

Se desarrolla un sistema de creación, recuperación y gestión de bancos de información formado por dos lenguajes y los traductores asociados, totalmente transparente al tipo de información tratada: numérica, alfabética y de palabras clave. El sistema presenta las facilidades de los I R S tradicionales más otras propias, como son: independencia con relación al tipo de información, lenguajes próximos a los habituales de programación, flexibilidad, etc , lo que nos ha llevado a acuñar el nuevo término de Banco de Información

1 INTRODUCCION

Uno de los problemas capitales con el que se enfrenta la sociedad de hoy es la recuperación de información, dada la enorme cantidad y variedad de documentos producidos, acertadamente calificada de polución de la información Las soluciones clásicas: revista de revistas, etc , se han visto desbordadas por la avalancha informativa y pese a la tecnología más flexible en cuanto al soporte de esta información: microfichas, microfilmes, etc , no se ha conseguido resolver el problema

El ordenador aparece como una solución, debido a su enorme capacidad de almacenamiento, velocidad de operación y fiabilidad Existen numerosos sistemas dedicados a esta tarea: ITIRC (IBM), MISTRAL (CII), etc., con una clara polarización hacia la recuperación de documentos y que, a nuestro entender, abordan el problema de una forma parcial (tipo de información tratada) y relativamente poco informática: estructura de los lenguajes de interrogación alejada de los lenguajes habituales de programación

Ha sido nuestro propósito, al realizar este trabajo, el hacer desaparecer estas dos limitaciones imponiendo a nuestro sistema las siguientes características:

a) Crear un sistema de recuperación de información completamente transparente en cuanto al tipo de información tratada: numérica, alfabética y palabras clave. Es decir, los objetos a tratar en nuestro sistema pueden caracterizarse por estos tres tipos de características, realizándose el acceso a ellos en función de estas con igual potencialidad, sin privilegiar, pues, ninguna de ellas

b) Los lenguajes que participan en el proceso:

— lenguaje de creación del banco de información y
— lenguaje de interrogación y actualización del banco de información,

presentan las características usuales de los lenguajes de programación: instrucciones declarativas, intrucciones de control e instrucciones operativas, con las implicaciones de comodidad de escritura de programas y facilidad de aprendizaje

(1) En el desarrollo de este trabajo, realizado en el Instituto de Informática, ha colaborado el Centro de Cálculo de la Universidad de Madrid.

(2) Este trabajo, ganador del Premio Nacional de Informática C.C.S. 1974, se reproduce por cortesía del Centro de Cálculo de Sabadell.

(*) Instituto de Informática de Madrid Centro de Cálculo de la Universidad de Madrid.

(**) Instituto de Informática de Madrid

c) Mantenemos las características clásicas de este tipo de sistemas más las dos anteriores Esto hace que el sistema sea aplicable tanto a la recuperación de documentos como al mantenimiento de bancos con información heterogénea: control de *stocks,* nóminas, personal, etc.

Creemos haber conseguido nuestros objetivos El sistema es operativo en estos momentos sobre UNIVAC-1108, pero, dados los criterios de implementación:

— Programación estructurada
— Independencia con respecto al ordenador
— Lenguajes de alto nivel utilizados en su implementación es perfectamente trasladable a cualquier otro ordenador y lenguajes

II PRESENTACION DEL TRABAJO

Hemos creado un sistema de recuperación de objetos caracterizados por:

— Un objeto puede tener un cierto numero de características de distintos tipos Una característica puede estar presente o ausente; si está presente admite un valor cuya representación depende del tipo de característica Si está ausente, ésto aparece reflejado en la representación escogida del objeto a efectos de acceso
— El numero de caracteristicas es potencialmente infinito aunque cada objeto venga caracterizado por un número finito de ellas

El almacenamiento de las informaciones (objetos) se realizan en memoria secundaria en un fichero secuencial (ya que el tratamiento se ha de realizar para todos y cada uno de los objetos) Esto no es exactamente verdad más que en una primera aproximación Ya veremos en IV 2.2 que el usuario sólo *ve* aquellos objetos del banco que con arreglo a su programa de interrogación le pueda interesar.

Cada objeto viene representado por un registro de *N* palabras (a fijar por el usuario) en el que aparecen:

— El objeto
— Informaciones asociadas con él.
— Representación del objeto en la forma más adecuada a efectos de su tratamiento: recuperación, modificación, actualización, supresión, etc
Esto equivale a caracterizar el objeto, a efectos de acceso, por las características que considere oportunas el usuario del banco

Esta representación de objetos la realiza automáticamente el intérprete de creación del banco, teniendo en cuenta el programa de creación del usuario (punto IV)

El acceso al banco consiste, para ciertos objetos, en modificar los valores de ciertas caracteristicas, añadir (o sea, hacer presente) o suprimir ciertas caracteristicas, así como recuperar objeto completo o parte en función de relaciones entre características definidas por el usuario en su programa de interrogación; éste se ejecuta aplicándolo sucesivamente a *todos* los objetos del banco

El programa de interrogación consiste en una serie de instrucciones separadas por ; y terminando con un signo de interrogación.

Una instrucción puede ser:

— Una llamada a un subprograma estándar.
— Una instrucción condicional del tipo:

si ε *entonces* J_1 *si no* J_2 *finsi*

Donde ε es una expresión booleana (definida sobre las características) y J_1, J_2 una colección de instrucciones (entre las cuales pueden encontrarse otras condiciones):

— Una transferencia de control.

Así, pues el trabajo consta de las siguientes partes:

1 *Lenguaje de creación del banco de información* en el que el usuario define su banco en cuanto a talla de los objetos, características que los van a hacer accesibles, valores de éstas, etc (punto IV 2).

2 *Traductor de los programas de creación* hemos escogido un intérprete basado en un autómata de estados finitos (punto IV 3).

3 *Lenguaje de interrogación del banco* en el que el usuario escribe los programas de interrogación del banco: consulta, modificación, etc (punto V 2).

4 *Traducción de los programas de interrogación* hemos construido un compilador con la finalidad de optimizar los tiempos de explotación de los programas que no necesitan ser retraducidos cada vez (punto V 3).

La puesta en marcha de un trabajo de esta envergadura implica una representación y almacenamiento de los objetos que optimice memoria y tiempo de acceso Es en el

punto III donde abordamos este problema con detalle

Además, y para facilitar la realización del proyecto, así como su comprensión, hemos recurrido a las siguientes técnicas:

1 *Programación estructurada* — Ambos traductores han sido implementados según esta técnica de programación, lo que implica:

— Fácil implementación
— Modularidad
— Legibilidad
— Extensibilidad
— Fácil modificación

Hemos huido de forma intencionada y explícita de los organigramas por considerarlos ineficientes e ilegibles Sólo en casos muy particulares han sido utilizados para describir acciones concretas Esta actividad se definió como política de trabajo de grupo desde un principio, el resultado ha sido una elevada eficiencia en la implementación, así como las características antes enunciadas

2 *Lenguajes de implementación de alto nivel* — Lo que nos ha permitido:

— Rapidez en la implementación
— Fácil depuración de los programas
— Legibilidad de éstos (dada su estructura lógica explícita)

El ALGOL-60, lenguaje idóneo para representar algoritmos, cumple este papel Además de proporcionarnos procedimientos recursivos tan necesarios en los procesos de traducción Esto no excluye cualquier otro tipo de lenguaje: PL/I, COBOL, FORTRAN, ENSAMBLADOR, ya que como veremos los métodos utilizados son independientes de las características particulares del lenguaje de implementación

3 *Lenguaje objeto de alto nivel* — Podíamos realizar consideraciones análogas al punto anterior

No obstante hemos de indicar que un P.O en lenguaje máquina sería más eficiente Podría ser una posible ampliación del trabajo

Nosotros hemos escogido como lenguaje objeto FORTRAN V

De una forma muy esquemática podíamos representar el proceso de creación y uso del banco en la forma siguiente (fig 1):

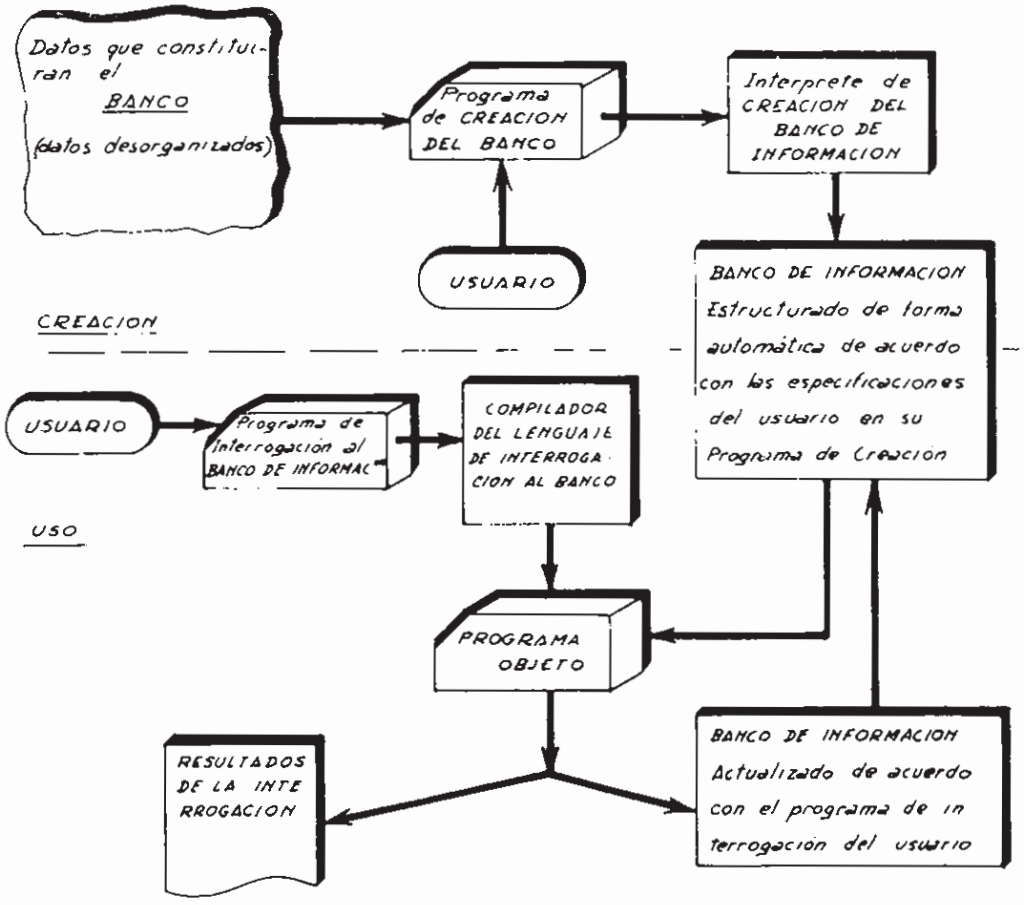


Figura 1

III REPRESENTACION Y ALMACENAMIENTO DE LOS OBJETOS

Pepresentación — El banco de información contendrá una colección de objetos sobre los que queremos realizar tratamientos: consulta, actualización, impresión, etcétera.

En nuestro trabajo hemos hecho la siguiente distinción:

Objetos. — Colección amorfa de informaciones

Representación de los objetos. — Creamos la imagen adecuada de los objetos a efectos de su acceso posterior de acuerdo con las especificaciones de los usuarios (realizadas en su programa de creación del banco)

Un objeto en el banco constará de estos dos aspectos El usuario trabajará siempre sobre la representación de los objetos, definida inicialmente en el programa de creación, pero actualizada de forma dinámica por el usuario en sus programas de interrogación (si es que así lo desea)

mática a efectos de actualización, modificación, etc

Así, pues, concretando, un objeto vendrá representado en el banco por un registro de *N* palabras (definido por el usuario en el momento de la creación) Estas *N* palabras se distribuirán en la forma siguiente:

N_1: Palabras a efectos de identificación externa del objeto (definidas por el usuario) En el caso de que el objeto sea un *abstract* de un libro o artículo podrá ser éste o una parte En cualquier otro caso podrá ser la información que crea oportuna

N_2: Palabras para la representación del objeto a efectos de acceso Depende de la caracterización del objeto por el usuario y de las necesidades impuestas por la representación escogida En nuestro caso, y dadas las características de los tratamientos a realizar:

- — acceso rápido,
- — modificación aleatoria (actualización o supresión),
- — poca memoria para representar el objeto,

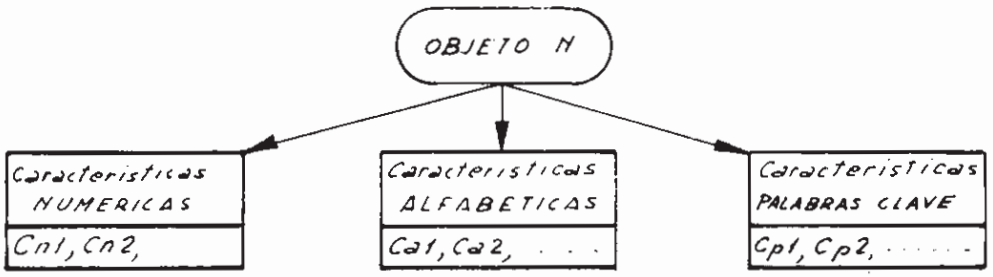


Figura 2

Un objeto podrá representarse esquemáticamente mediante una colección de características:

Esta representación, como hemos dicho, no es estática, sino dinámica, en función de los intereses del usuario que podrá añadir, modificar, suprimir características (de cualquiera de los tres tipos) en el orden que crea más oportuno. El numero, tipo y valor de las características por objeto es variable: estática (momento de la creación) y dinámicamente (momento del uso del banco), así como el número de objetos que forman el banco No creemos necesario insistir sobre la flexibilidad y potencia de esta idea. Un objeto podrá ser cualquier cosa: un libro, una pieza de un almacén, un empleado de una entidad, un alumno de un centro docente, etc, que podrá ser tratado de forma auto-

hemos escogido una tabla de acceso disperso ligada que detallaremos en el punto III.1 3.

N_3: El resto de las *N* palabras organizadas en forma de dos listas libres ligadas de talla de elemento, tres y cuatro palabras respectivamente Estas dos listas constituyen el espacio libre del que dispone el usuario y que autorizan la talla variable de la representación de los objetos (a efectos de añadir y suprimir características)

Obviamente, $N = N_1 + N_2 + N_3$ (N_2 y N_3 son variables, pero $N_2 + N_3 = N - N_1$)

Toda característica, a efectos de tratamiento interno, vendrá representada por un código numérico (asignado de forma automática en el mometno de creación del banco) y un tipo (numérica, alfabética, palabra clave) La correspondencia entre estas in-

formaciones se realizará mediante la tabla de símbolos directa que asocia:

— Identificador de característica → numero de código asociado y tipo que veremos en el punto IV 2 1 1

Existe otra tabla de símbolos inversa (creada en el mismo momento que la anterior) que efectua la correspondencia:

— Numero de código → identificador de característica, que veremos en el punto IV.2 1 2

Ambas tablas, producidas a la creación del banco, son necesarias en el momento de la compilación de un programa de interrogación, lo que implica que deben aparecer en el banco de información (por ejemplo, al principio) para poder ser usadas por el compilador

Las dos, dadas las características dinámicas de las informaciones a almacenar, son también tablas de acceso disperso ligadas

Pero volvemos a nuestra descripción de objetos en el banco de información (fig 3).

Las N_2 palabras serán utilizadas en la forma siguiente:

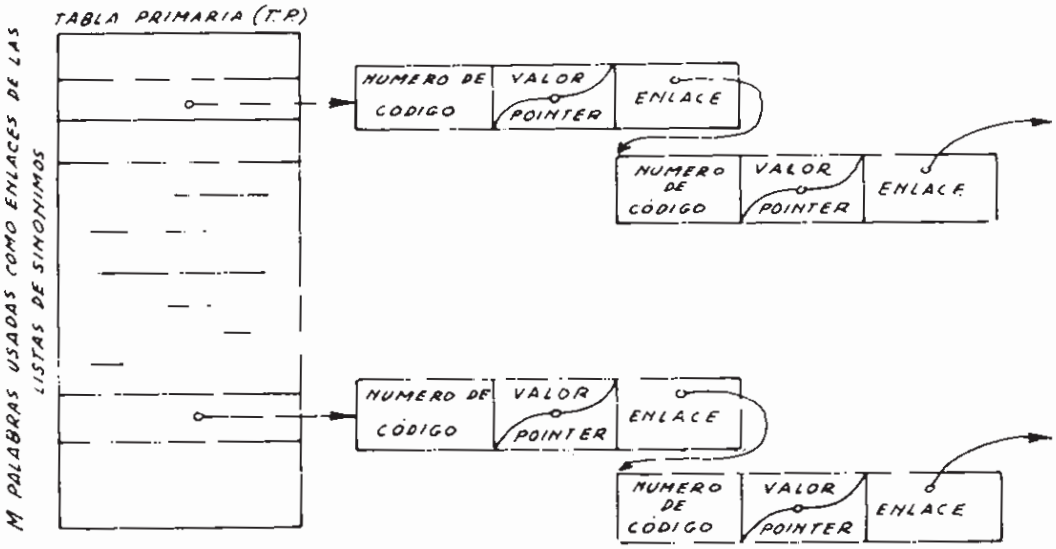


Fig 3 — Tabla de acceso disperso

El proceso de acceso a una característica lo esquematizaremos como la función (fig 4):

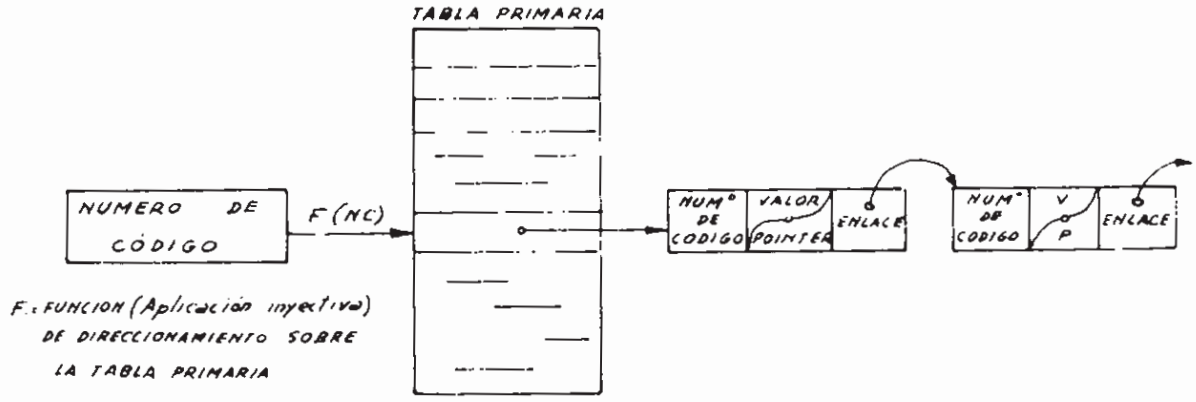


Figura 4

Como el almacenamiento es secuencial (memoria secuencial), el aspecto que presentaría un objeto sería:

— KEY Es el número de orden del objeto dentro del fichero
— Descripción del objeto Es una identificación para efectos externos e incluso podría ser una parte de un abstracto
— Pointer Cont Abstracto Enlace con el resto del abstracto
— CLL Es la cabeza de la lista libre

Un objeto tiene asignado un numero de palabras determinado, pero siempre fijo (*N*)

Si no tiene especificadas todas las características que puede físicamente contener habrá espacio libre, y este espacio libre se organiza en grupos de tres palabras y están todos los grupos enlazados, con organización de lista ligada

— Tabla primaria de características Es una tabla hasta que por medio del numero de código nos da la dirección del grupo de tres elementos que identifica la característica.

— CLDC Es la cabeza de la lista libre de los elementos que contienen el reflejo de una característica en el abstracto o el valor alfanumérico de una característica Estos elementos de grupos de cuatro pueden ir enlazados definiendo el vector de posición de una palabra en un abstracto o su valor alfanumérico (fig 5)

El tamaño de los registros y de cada campo será el siguiente:

KEY Una palabra con un número entero	1 W
Descripcion objeto Alfanumérico	25 W
Pointer Cont Abst Una palabra con un número entero	1 W
CLL Cabeza de lista libre, una palabra con un número entero	1 W
Tabla primaria Direcciones de las características	20 W
Número código Una palabra con un número entero	1 W
Valor Pointer Una palabra con un número real	1 W
Enlace Una palabra con un número entero	1 W
CLDC Una palabra con un número entero	1 W
Posición valor Alf Una palabra alfanumérico o entero	1 W
Enlace Una palabra con un número entero	1 W

Como podemos tener como máximo, por ejemplo, por cada objeto 57 caract, el conjunto de caracteristicas tendrá una longitud de 171 W, y el conjunto de valores alfabéticos o posiciones tendrá una longitud de 228 W Lo cual hace que la longitud del registro total sea de 448 W (1/4 de pista de

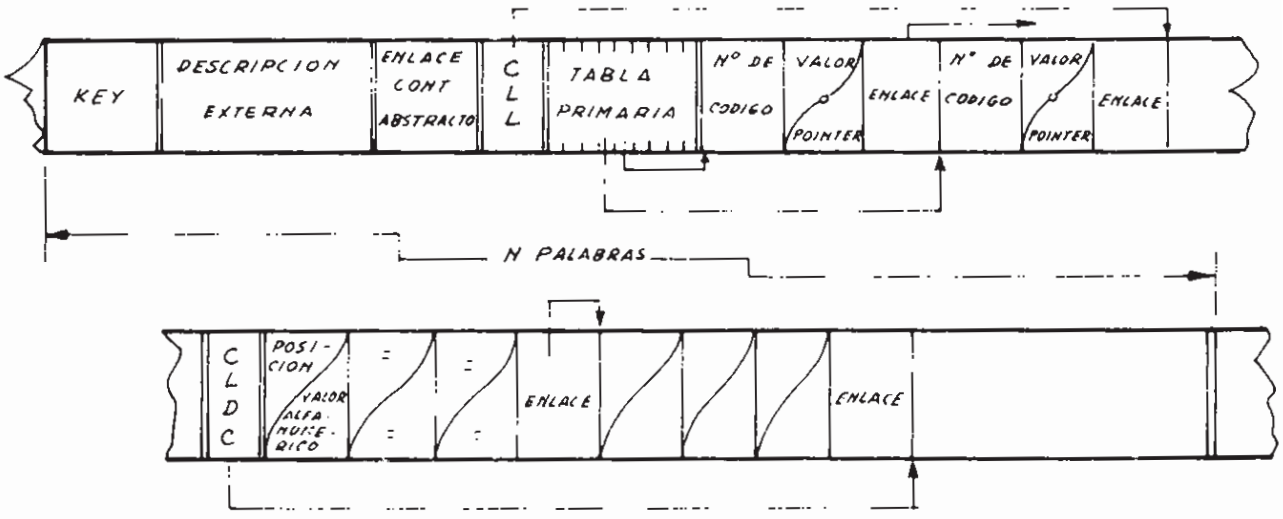


Figura 5

La continuación del abstracto se pondrá en un área al final del fichero, siendo los registros de longitud variable El formato del área será el siguiente (fig 6):

Fastrand), que es el dispositivo periférico de almacenamiento de información utilizado en nuestro caso

La facilidad de dividir el objeto en dos

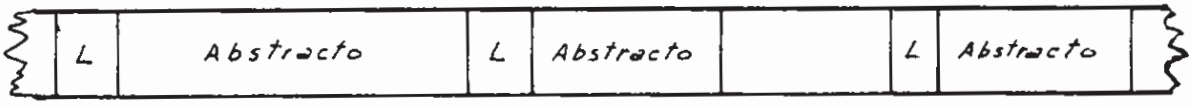


Figura 6

partes (por ejemplo, los abstractos) desaparecería caso de usar una memoria auxiliar tipicamente secuencial: por ejemplo, una cinta magnética. Esto obligaría a poner todo el objeto en las N_1 palabras, lo que no supone ninguna limitación.

Con este tipo de estructuración de la información se permite el poder cambiar características de los distintos tipos en un mismo objeto.

Como para cada tipo de características el tratamiento es diferente, entonces habrá que pasar por las TDS para saber el tipo, y dependiendo de ésta, hacer el tratamiento que corresponda. Por tanto, vamos a ver los distintos tratamientos que se van a hacer con una característica para los distintos tipos de características.

III 1 1 *Tipos de características: numéricas, alfabéticas y palabras clave*

La representación de estas características es la siguiente:

Todas ellas aparecen como un triplete en la representación del objeto (fig. 7):

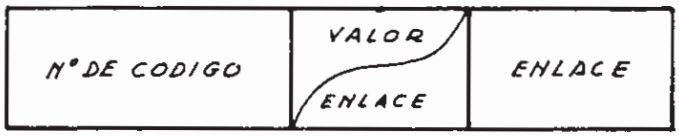


Figura 7.

Primera palabra

El numero de código es el asignado en la tabla de simbolos directa

Segunda palabra

Segun el tipo de la característica contendrá:

Valor. Caso de caracteristicas numéricas, su valor se puede representar en una palabra.

Enlace: Caso de características alfabéticas y palabras clave. Su valor es de talla variable (cadenas de longitud cualquiera) lo que obliga a una representación idónea

Tercera palabra

Tiene en cuenta la estructura ligada de la tabla escogida (*Hash* ligada).

Característica alfabética — Contiene el enlace de la lista de cuadrupletas que representan su valor (cadena de caracteres) (figura 8):



Figura 8.

Característica palabra clave — Podemos definir el valor de una palabra clave como:

— Su presencia o ausencia de un texto.
— Las posiciones en que aparece

Por tanto, ENLACE será:
— *Nil* (la característica no aparece en el objeto)
— *Dirección* del primer cuadruplete, que contiene las posiciones en que aparece en el texto (fig. 9):

NOTA: *Nil* representa la ausencia de enlace (será cualquier cosa que no sea una dirección)

Así, pues, hemos representado todos los tipos de características.

El hecho de que el valor de una palabra clave sea su presencia o ausencia en un texto y caso de presencia sus posiciones a lo largo del texto, nos permite recuperar textos análogamente a los sistemas clásicos de Recuperación de la Información.

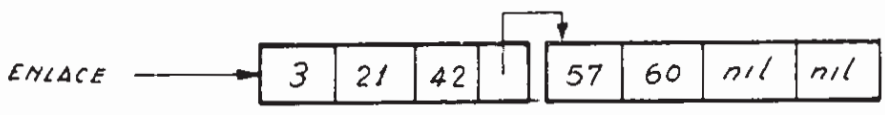


Figura 9

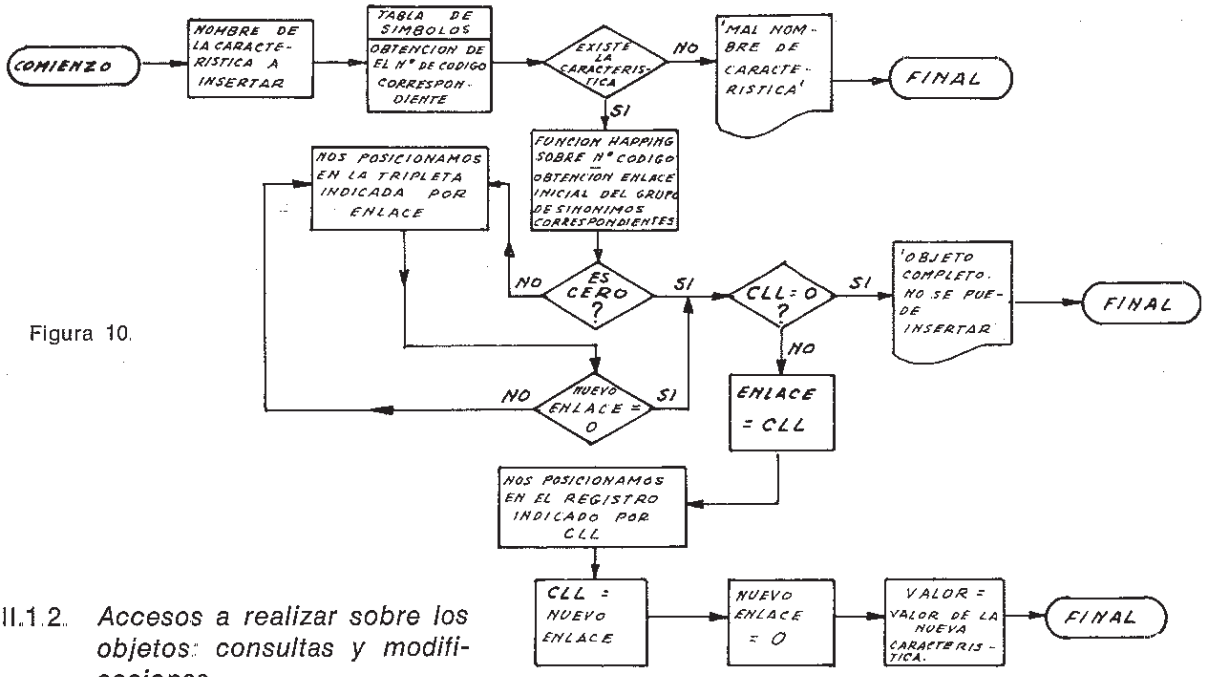


Figura 10.

III.1.2. *Accesos a realizar sobre los objetos: consultas y modificaciones.*

Accesos a realizar sobre las características de tipo numérico (o alfabético).

La información que ya hemos visto organizada va a ser objeto de diversos planteamientos, tales como Consulta para comparación con otra información, localización para extraer esa información o simplemente una modificación de la información y hasta su supresión si llega al caso.

Veamos ahora los algoritmos básicos de todos estos tratamientos. Sustancialmente van a ser:

— Inserción de una nueva característica. Se supone que estamos ya posicionados en el *objeto* en que queremos insertar la nueva característica (fig. 10).

— Búsqueda y localización de una característica. Lo mismo que antes, supomos que estamos posicionados en el registro u objeto a tratar (fig. 11).

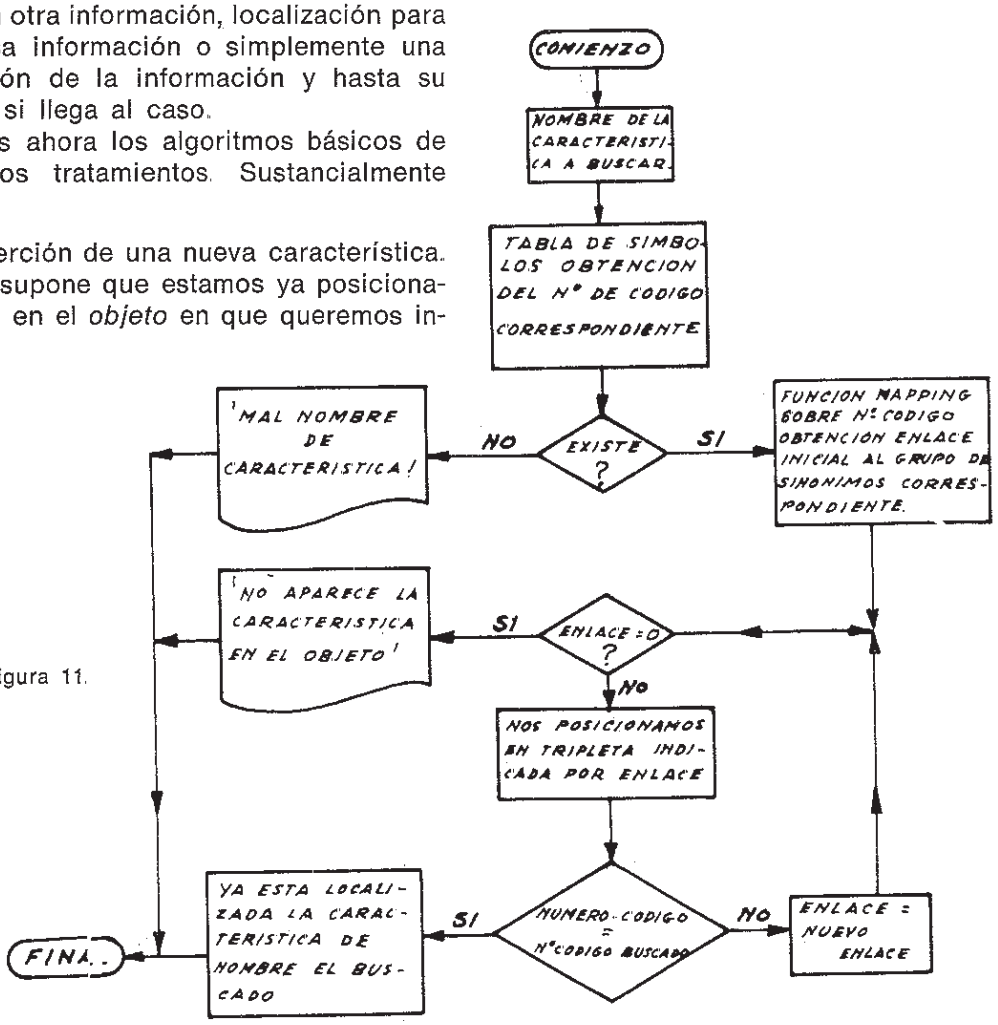


Figura 11.

— La modificación de una característica se efectuará sencillamente llevando a cabo la busqueda de dicha característica, y una vez localizada, modificando el campo *valor* de la información obtenida

— La eliminación de una característica

El algoritmo de eliminación sería de la forma siguiente:

— Supongamos que estamos en el *objeto* que nos interesa tratar En primer lugar se lleva a cabo una busqueda de la característica a eliminar (fig 12)

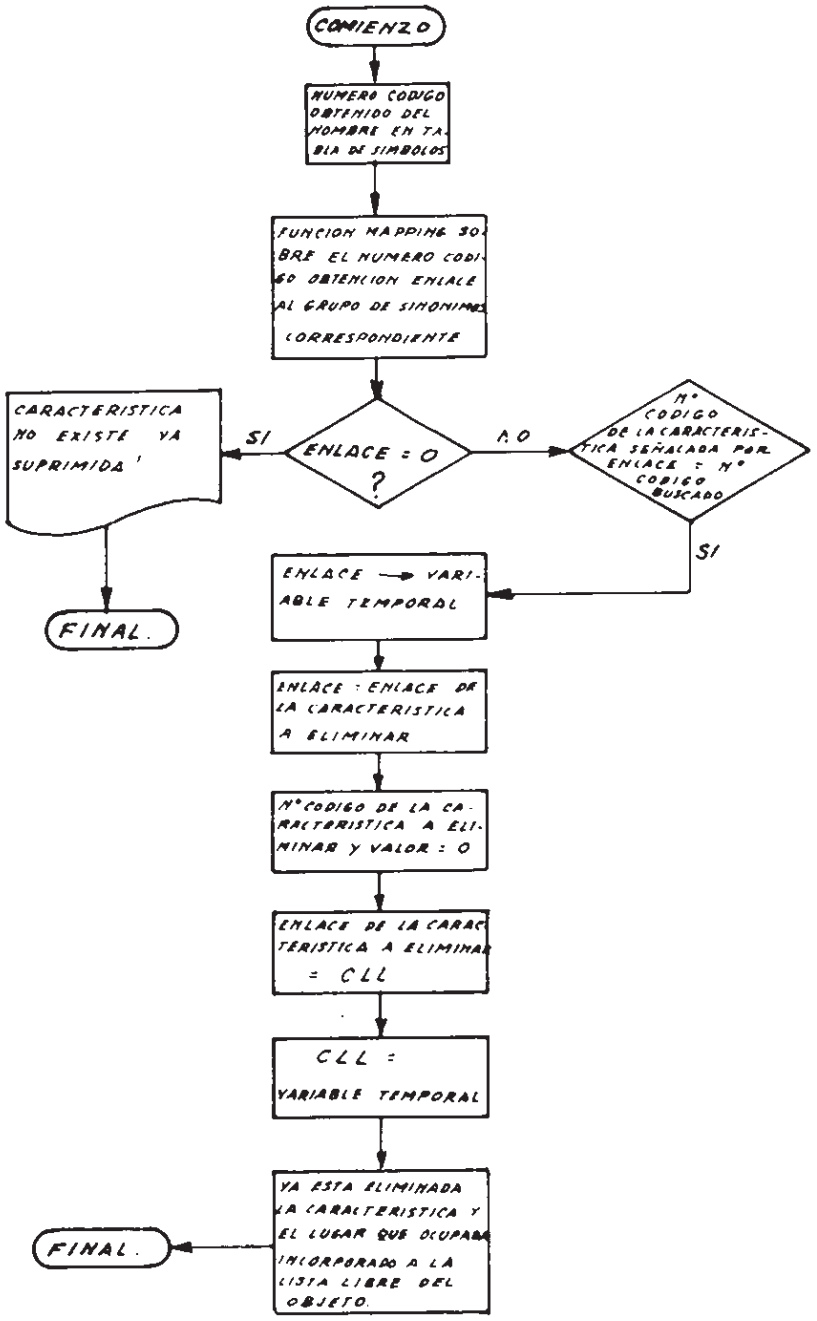


Figura 12

Sobre estos algoritmos básicos se va a basar todo el proceso a realizar con la información almacenada en nuestro fichero. En el fondo sólo vamos a buscar información para componerla, modificarla, eliminarla o extraerla para exteriorizarla

En la actualización del valor de una característica se tiene en cuenta el tipo de

ésta. Una supresión de una característica hace que las palabras que la representaban vuelvan a las respectivas listas libres

Evidentemente, las características del tipo palabra clave sólo se consultan

Accesos a realizar sobre la información de tipo alfabético

Vamos a ver los algoritmos básicos de los tratamientos que se van a realizar sobre la información de tipo alfabético:

— Inserción de una nueva característica
Se supone que estamos ya posicionados en el objeto en que queremos insertar la nueva característica:

INSTHA:

A1 Obtención del número de código de la característica
A2 Si no existe la característica entonces escribir 'Mal nombre Característica', ir a A14
A3 *K* = función de mapping (número de código) por la tabla primaria
A4 Recorrer los grupos de sinónimos hasta encontrar el último (enlace = 0)
A5 Si CLL = 0 entonces escribir No se puede insertar, ir a A14
A6 ENLACE = CLL
A7 CLL = NUEVO ENLACE del grupo de tres donde vamos a meter la nueva característica
A8 NUEVO ENLACE = 0
A9 VALOR O POINTER = CLDC
A10 Si CLDC = 0 entonces escribir No se puede insertar, ir a A14
A11 DC (CLDC) = literal
A12 Si el número de caracteres de literal es mayor que 18, entonces:
CLDC = NUEVO ENLACE de DC
IR a A11
A13 NUEVO ENLACE de DC = 0
A14 Fin

— La modificación de una característica alfabética supone una busqueda en el objeto de donde está localizada, y una vez hecho esto, acceder a DC donde esté su valor, y lo modificamos con el literal de la siguiente forma:

MODIFA:

A1 DC (Pointer) = literal
A2 Si el número de caracteres de literal es mayor que 18 y el pointer del grupo de cuatro de DC no es cero, ir a A1
A3 Si CLDC = 0 entonces escribir 'No se puede modificar', ir a A6
A4 Coger un grupo de cuatro de la lista libre. Pointer = CLDC
A5 Ir a A1
A6 Fin

— La eliminación de una características alfabética supone una busqueda de la característica a eliminar, y una vez hecho esto, acceder a OBJET y encadenar el grupo de tres a la lista libre y acceder con el Pointer de OBJET a DC y encadenar todos los grupos de cuatro palabras asignados a la caracteristica a la lista libre:

SUPTHA:

A1 ENLACE ANTERIOR OBJET = ENLACE OBJET
A3 CLL = ENLACE ANTERIOR OBJET
A2 ENLACE OBJET = CLL
A4 CLDC = Pointer de OBJET
A5 Recorrer DC (CLDC) hasta el pointer cero
A6 Pointer cero = antiguo CLDC
A7 Fin

Accesos a realizar sobre la información de palabras clave

— La inserción de una palabra clave una vez hallado su vector de posiciones con respecto al abstracto y formando el vector como un literal, se hará de la misma forma que para las alfabéticas (INSTHP)
— La eliminación de una palabra clave se hace de la misma forma que las de tipo alfabético.

III 1 3 *Estructura de la información adecuada*

La información del banco de información viene organizada de una manera secuencial en cuanto a los objetos o elementos del archivo Dentro de cada elemento, la información de las características se ha organizado en forma de tabla Hash ligada, para un acceso más rápido y mejor a pesar de que la ocupación de espacio va a ser un poco mayor debido a los enlaces y a la tabla primaria de la tabla Hash Ya hemos dicho que como máximo habrá *M* características en cada objeto

También se ha usado la estructura de tabla Hash ligada en la formación de la *tabla de símbolos*, que contiene todos los posibles nombres de características para asignarles un numero de código.

La tabla primaria de la tabla de símbolos va a clasificar los nombres de característica, por su inicial, en 26 grupos Cada grupo contendrá todos los nombres de característica que empiecen por la misma letra

Convendría procurar que los nombres de característica más usuales estuvieran lo más cerca posible del comienzo de su correspondiente lista de sinónimos, para mejorar la

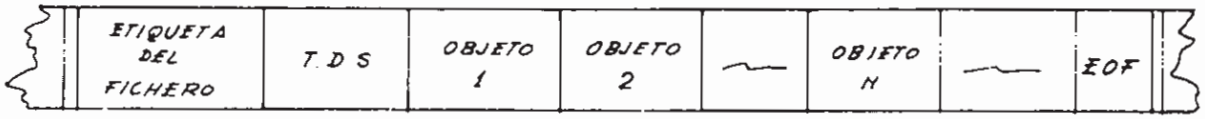
eficiencia al disminuir el tiempo de búsqueda de cada identificador de característica

La *tabla de símbolos* se coloca al principio del fichero, y será lo primero que se lea, manteniéndose en memoria durante todo el tiempo que dure el proceso, mejorando así el tiempo de acceso

No insistimos más sobre estas estructuras de información: las tablas de acceso disperso ligadas (o tablas Hash ligadas) que suponemos conocidas y que aparecen ampliamente tratadas en la bibliografía informática

III 2 *Almacenamiento de los objetos*

Los objetos se van a almacenar en un fichero secuencial, en un dispositivo de acceso directo, pero siempre con una organización secuencial pura, puesto que un tratamiento del fichero se va a extender a todos y cada uno de los objetos.



La T.D.S contiene todos los posibles nombres de característica que pueden tener los objetos del fichero, juntamente con su correspondiente número de código y su tipo

Cada objeto tiene asignado un numero fijo de palabras por el usuario, o sea, que vamos a tener registros de longitud fija, y cada objeto puede tener un abstracto al final del fichero de longitud variable

Cada vez que se procesa un objeto se lleva a memoria el registro completo, y si es modificado se reescribe otra vez en el mismo lugar que ocupaba, para ello cada objeto lleva asociada una KEY o numero de orden del objeto dentro del fichero

En nuestro caso, y como hemos dicho, el soporte del banco es el tambor magnético FASTRAND del UNIVAC-1108. Otros soportes son perfectamente posibles: discos, cintas magnéticas, etc

* * *

NOTA: El desarrollo de este trabajo continuará en el próximo número

BIBLIOGRAFIA

PAIR, C: "Les structures de l'information" Ales, julio 1971

SISTEMA PARA LA CREACION Y MANTENIMIENTO DE BANCOS DE INFORMACION

POR

I RAMOS
Calixto LOZANO
A MUÑOZ

PUBLICADO EN:

REVISTA DE INFORMATICA Y AUTOMATICA

AÑO VIII - NUM 23 - 1975

SISTEMA PARA LA CREACION Y MANTENIMIENTO DE BANCOS DE INFORMACION

(PARTE II)

Por I. RAMOS (*)
Calixto LOZANO (**)
A. MUÑOZ (**)

IV. LENGUAJE DE CREACION DEL BANCO E INTERPRETE ASOCIADO

IV.1. *Presentación del problema. Ejemplo.*

Vamos a suponer una aplicación real del sistema y veremos cómo van a incidir en el ejemplo las distintas partes de nuestro sistema.

Supongamos que queremos construir un banco de información que incluya todos los modelos actualmente en uso de ordenadores electrónicos. Podríamos así recuperar más tarde subconjuntos de este banco de información que cumplan con una serie de condiciones entre sus características.

Podrían ser características *numéricas* de los objetos de este banco:

— Ciclo de memoria.

— Ciclo de instrucción.

— Tamaño (en K) de la memoria.

— Numero de procesadores.

— Número de líneas que se le pueden conectar.

— Precio y alquiler, etc.

De la misma manera, podríamos tener características de tipo *alfabético* tales como:

— Organización de la memoria: 'PALABRA' o 'BYTE'.

— Marca, por ejemplo: 'UNIVAC', 'HONEYWELL BULL', 'CONTROL DATA'.

— Modelo, o sea: '1108', '370-145', '7090', 'CIBER', 'IRIS-80'.

— Tipo del byte: 'SEXTETO' u 'OCTETO'.

Y admitiríamos también características de tipo palabra-clave, como las siguientes:

— COMPATIBLE o no.

— Todos los procesadores: COBOL, FORTRAN, ALGOL, PL1, etc.

— Las características que llevan su valor implícito en su presencia o ausencia: T. REAL, T. COMPARTIDO, MULTIPROCESO, etc.

— El sistema debe ser capaz de crear el banco de información a partir de sentencias de asignación tales como:

```
MARCA   = CONTROL DATA CORP';
MODELO  = 'CIBER';
SIZEME  = 256;
ORGMEM  = BYTES ; COBOL; FORTRAN; ALGOL: PL1;
PRECIO  = 200 000.000;
```

A partir de estas necesidades vamos a ver cómo va a ser el lenguaje de creación del banco de información.

IV.2. *Lenguaje de creación del banco.*

Hemos intentado que el lenguaje que ha de servir para llevar a cabo la creación del banco de información sea lo más sencillo de manejar y, al mismo tiempo, ofrezca una potencia aceptable en cuanto a facilidades de programación. Un programa escrito en este lenguaje es examinado por un intérprete que analiza la corrección de la sintaxis de lo escrito y va ejecutando las sentencias a medida que se comprueba su sintaxis.

(*) Instituto de Informática. Centro de Cálculo de la Universidad de Madrid.
(**) Instituto de Informática

IV.2.1. *Definición informal del lenguaje de creación.*

Se pueden distinguir dos fases en un programa de creación:

— Una fase de DECLARACIONES donde se especifican los tamaños de los objetos, de la información asociada a ellos de cara a su identificación externa, el nombre del fichero y los nombres de TODAS las características posibles para ese banco.

— Una fase de DESCRIPCION de los objetos que componen el banco. Esta fase se divide en tantos párrafos como objetos tiene el banco en su creación. En esta fase se le asigna a cada objeto unos valores para unas determinadas características que posee.

Vamos a ver ahora más detenidamente cada una de estas dos fases.

IV.2.1.1. *Declaraciones.*

El programa empieza con la palabra FICHERO seguida del nombre del fichero escrito entre comillas.

Inmediatamente después escribiremos una de las dos palabras CREACION o ACTUALIZACION, según que el fichero exista ya y sea una nueva etapa en su construcción o que lo vayamos a crear con esta pasada y no exista todavía. Sea un ejemplo:

```
FICHERO 'ALUMNOS  CREACION, o bien
FICHERO  LIBROS  ACTUALIZACION
```

Esta diferenciación es debida a que si es actualización hay que leer el fichero ya existente y modificar, si es necesario, sus tablas de símbolos y su matriz booleana de características por objeto, así como tomar la dirección de grabación donde se ha de incluir el siguiente objeto, mientras que si es creación hay que inicializar todo el espacio necesario, tablas y matriz booleana, partiendo de cero.

Si aparece la palabra ACTUALIZACION no hace falta definir ni la talla de los objetos ni la longitud de la información descriptiva alfanumérica asociada a cada objeto de cara a su identificación externa.

En caso de que sea CREACION habría que definir: TALLA OBJETOS *n*; INFORMACION OBJETOS *m*, siendo siempre $m < n$. *n* y *m* son dos enteros que definen: *n* = longitud en palabras del registro en que se almacena un objeto; *m* = longitud de su descripción externa identificativa, que será la que normalmente necesitemos recuperar.

A continuación, ya sean programas de creación o de actualización, se definen TODAS las variables que se van a usar en el programa, o sea, todas las posibles CARACTERISTICAS de los objetos a que se va a hacer referencia. Dichas declaraciones se efectúan de la forma:

```
CARACTERISTICAS <tipo> <serie de características
                del mismo tipo>;
                <tipo> <serie de características
                del mismo tipo>;
                . . . . . .
                <tipo> <serie de características
                del mismo tipo>;
```

El tipo puede ser NUMERICAS, ALFABETICAS o PALABRAS-CLAVE. Los nombres de característica pueden ser de cualquier longitud, aunque el intérprete sólo cogerá los seis primeros caracteres y por esos será reconocida y tratada la característica.

Una serie de características es un conjunto de nombres de característica separados por comas. Así, una declaración sería:

```
CARACTERISTICAS ALFABETICAS NOMBRE,APEL1,
                  APEL2,CALLE;
                NUMERICAS EDAD NHIJOS;
                PALABRAS-CLAVE TORNERO,
                  FRESADOR,MAESTRO,ALBAÑIL;
                NUMERICAS TELEF ;
                ALFABETICAS LUGNAC, LUGRES;
```

Se puede, como se ve, declarar características de un tipo en varias etapas, subsanando así la posibilidad de olvidos, pues en caso de que nos olvidemos de haber declarado una característica de un tipo que ya ha sido declarado, volveríamos a declarar de nuevo el tipo incluyendo ya la característica, como se ve en el ejemplo con la característica TELEF. y con LUGNAC y LUGRES.

Esta posibilidad tiene aún una ventaja: como la tabla de símbolos inversa se va creando de una manera secuencial según van apareciendo las características, y en el mismo orden se les asigna número de código, es conveniente declarar, en primer lugar, las características más usadas o que se prevea que van a ser más usadas, acelerando de esta manera su localización, al aparecer en los primeros lugares de la tabla de símbolos inversa y en los grupos de sinónimos en los primeros lugares también, si se trata de otras tablas.

IV.2.1.2. *Asignaciones en cada objeto.*

Una vez finalizada la parte de declaraciones y empezada la fase de asignaciones, ya no se pueden hacer más declaraciones. La fase de asignaciones tiene un párrafo para cada objeto. Un párrafo estaría organizado como sigue:

— Comienza por la palabra OBJETO.

— A continuación sigue un número entero que es la clave numérica del OBJETO y serviría para su direccionamiento.

— A continuación se codificará entre comillas la descripción externa del objeto, no pudiendo la longitud de ésta exceder de la longitud asignada en INFORMACION OBJETOS *m*.

Por tanto, el número de caracteres de la descripción externa, *n c*, será $n\,c \leq m.6$, ya que caben seis caracteres en una palabra y *m* viene dado en PALABRAS.

Detrás de la descripción externa hay que poner un PUNTO.

```
Ej : OBJETO 31   TURISMO ROLLS-ROYCE 1973
     5 PLAZAS, BERLINA, 600 CV. '.
```

Antes del punto, y después de la descripción, puede venir o no (es opcional) un abstracto o descripción del objeto por palabras clave, cuya presencia o ausencia y el orden o concatenación de unas con otras son parte esencial en la individualización del objeto. Si existe tal abstracto se codificará mediante la palabra ABSTRACTO, seguida del conjunto de palabras-clave que lo componen, separadas por blancos y encerradas entre comillas. Ejemplo:

```
OBJETO 32  TURISMO BENTLEY 1973, 5 PLAZAS,
BERLINA 600 CV  ABSTRACTO TAPICE MARRON
     LUNAS COLOR RADIO CASSET ANTENA
```

Ahora vienen las asignaciones propiamente dichas. Una asignación le asigna un valor a una característica del objeto. Habrá tres tipos de asignaciones, un tipo para cada tipo de característica. Veámoslo:

— Para características numéricas:

```
Característica = [característica = [ ]] número
```

Lo cual permite hacer asignaciones múltiples del tipo:

```
PRECIO = NHORAS = CANTID = DESCUE = 0;
```

Número es cualquier número real o entero:

— Para características alfabéticas:

```
Característica = [característica = [ ]] literal
```

```
Ejemplo:  TITULO = AUTOR = EDITORIAL =      ;
```

— Para características de tipo palabra-clave.

Sólo es necesaria la presencia de la palabra; de esta manera se asigna la presencia de ella:

```
Ejemplo: LITERA; CIENCI; NOVELA; FICCIO;
```

Se pueden codificar las palabras-clave, lo mismo que los nombres de característica, con más de seis caracteres, pero el sistema sólo toma los seis primeros caracteres, así que es mejor en cuanto a rapidez de tratamiento crear los nombres de característica y palabras-clave con sólo seis caracteres, creando un diccionario externo para nuestro uso que nos indique el significado de cada nombre, diccionario que, en cuanto nos hallemos familiarizados con el banco de datos, nos sobrará:

— Detrás de la última asignación del objeto y señalando el final del párrafo dedicado a éste, se codifica un *.

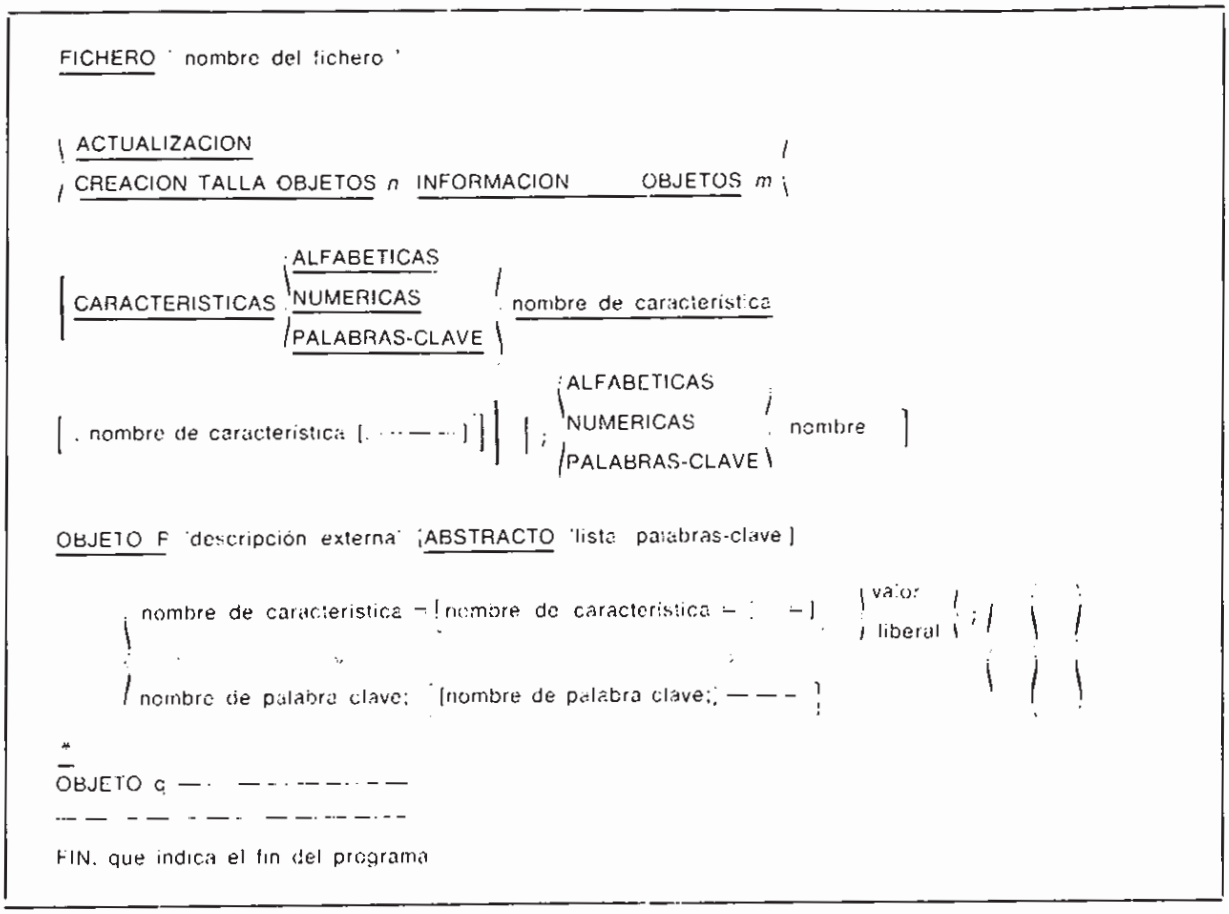
Un párrafo puede ser como el que sigue:

```
OBJETO 27 TURISMO SEAT 127 2 PUERTAS, TRAC
   DELANTERA
ABSTRACTO COLOR BLANCO TAPIC ROJA LUNAS
   VERDES NEUMAT RADIAL
PRECIO = 147.000; NASIEN = NCILIN = 4; POTDIN = 52;
   CINTOS;
CILIND = 903; ENTINI = 50 000; PLAZ18 = 7 000;
   PLAZ24 = 5.100; RADIO;
DENOM = URO127 ; TIPO = BERLINA ; CARROC =
   'AUTOPORTANTE ;
CAPMAL = 364;*
```

Tras toda la sucesión de objetos, a cada uno de los cuales corresponde un párrafo de programa de creación y señalando la terminación del programa aparecerá la palabra reservada FIN.

Veamos ahora un pequeño resumen o modo de recordatorio del lenguaje de creación:

ESTRUCTURA DE UN PROGRAMA DE CREACION



IV 2 2. *Definición formal del lenguaje de creación*

El lenguaje de creación del banco de información viene definido formalmente por la siguiente gramática de tipo 3, segun la clasificación de Chomsky, si consideramos como terminales algunos elementos que podrían ser considerados en un primer paso como clases sintácticas, tales como:

— Literal (cualquier cadena de caracteres incluidos entre comillas)
— Entero (cualquier cadena de dígitos)
— Identificador (un nombre de hasta seis caracteres)
— Numero (cualquier numero entero o real)

Si aceptamos estas cuatro clases sintácticas como terminales, la gramática puede ser, como hemos visto arriba, de tipo 3 ó regular y será la siguiente

1 <PROG> ::= FICHERO <RP>
2 <RP> ::= LITERAL <DECL>
3 <DECL> ::= CREACION <DEC1>
4 <DECL> ::= ACTUALIZACION <DEC5>
5 <DEC1> ::= TALLA OBJETOS <DEC2>
6 <DEC2> ::= ENTERO <DEC3>
7 <DEC3> ::= INFORMACION OBJETOS <DEC4>
8 <DEC4> ::= ENTERO <DEC5>
9 <DEC5> ::= CARACTERISTICAS <TIPO>
10 <TIPO> ::= ALFABETICAS <LISTA>
11 <TIPO> ::= NUMERICAS <LISTA>
12 <TIPO> ::= PALABRAS-CLAVE <LISTA>
13 <LISTA> ::= IDENTIFICADOR <RID>
14 <RID> ::= , <LISTA>
15 <RID> ::= ; <TIPO>
16 <RID> ::= . <EJEC>
17 <EJEC> ::= OBJETO <CEJEC>
18 <CEJEC> ::= ENTERO <INFR>
19 <INFR> ::= LITERAL <ABS>
20 <ABS> ::= ABSTRACTO <CABS>
21 <ABS> ::= , <CASG>
22 <CABS> ::= LITERAL <CASG1>
23 <CASG1> ::= , <CASG>
24 <CASG> ::= IDENTIFICADOR <RHASG>
25 <RHASG> ::= = <CRH>
26 <RHASG> ::= , <OTHAS>

27 <CRH> ::= NUMERO <CLASG>

28 <CRH> ::= LITERAL <CLASG>

29 CRH ::= IDENTIFICADOR <RHASG>

30 CLASG ::= ; <OTHAS>

31 OTHAS ::= IDENTIFICADOR <RHASG>

32 OTHAS ::= * <COBJ>

33 COBJ ::= OBJETO <CEJEC>

34 COBJ ::= FIN

Ya tenemos asi definida la gramática que va a generar el lenguaje de creación Apoyándonos en esta gramática vamos a construir el analizador sintáctico y el generador de acciones semánticas Ambas funciones se van a llevar a cabo de una manera simultánea y no vamos a generar un código objeto, sino que se van a generar directamente acciones, siendo ésta la filosofia de los INTERPRETES

IV.3 *Intérprete asociado al lenguaje de creación*

Obtenida ya la gramática que genera el lenguaje de creación se nos presenta el problema de construir un programa que acepte como datos una cadena de caracteres, que constituye el programa de creación, le analice sintácticamente, y si la sintaxis es correcta, genera una serie de acciones que hacen exactamente lo que hemos querido indicar en la codificación del programa fuente Este programa va a tener caracteristicas de INTERPRETE, pues no va a producir un código objeto, sino que va a efectuar directamente las acciones requeridas por el programa fuente y va generando éstas al mismo tiempo que va analizando la sintaxis del programa

IV.3.1 *El intérprete como un autómata de estados finitos*

Como es sabido, se puede asociar a una gramática de tipo 3 o regular un autómata de estados finitos deterministico capaz de reconocer cualquier palabra generada por dicha gramática.

Si definimos un autómata tal que el conjunto de estados se corresponda exactamente con el conjunto de clases sintácticas definidas en la gramática, el conjunto de los *inputs* se corresponde exactamente con el conjunto de los terminales de la gramática, el estado inicial del autómata se corresponde con el axioma de la gramática y creamos dos estados nuevos: *a*) estado FINAL, y *b*) estado de ERROR, con el diagrama de transiciones que a continuación expresamos, hemos construido el autómata reconocedor del lenguaje generado por la gramática que nos ocupa

La materialización del autómata como programa se ha llevado a cabo de la forma:

— Cada estado del autómata se convierte en la etiqueta de un 'módulo' de programación

— Cada módulo de programación constará de una bifurcación a otro módulo de programación, dependiendo de que se detecte o no el *input* que corresponde al arco correspondiente del diagrama de transiciones

En este sentido, hay que hacer una aclaración: hemos visto que entre los INPUTS hay dos clases determinadas, una a la que pertenecen palabras terminales o palabras reservadas y otra a la que pertenecen los cuatro tipos de 'falsos terminales', o sea: identificadores, numeros (reales o enteros), enteros y literales

Naturalmente, cuando intentemos preguntar por la existencia de una palabra reservada determinada, la pregunta será distinta de la que tenemos que hacer cuando preguntamos por uno de los 'falsos terminales'.

La distinción a la hora de chequear terminales se efectua con las rutinas de testeo de terminales, que son:

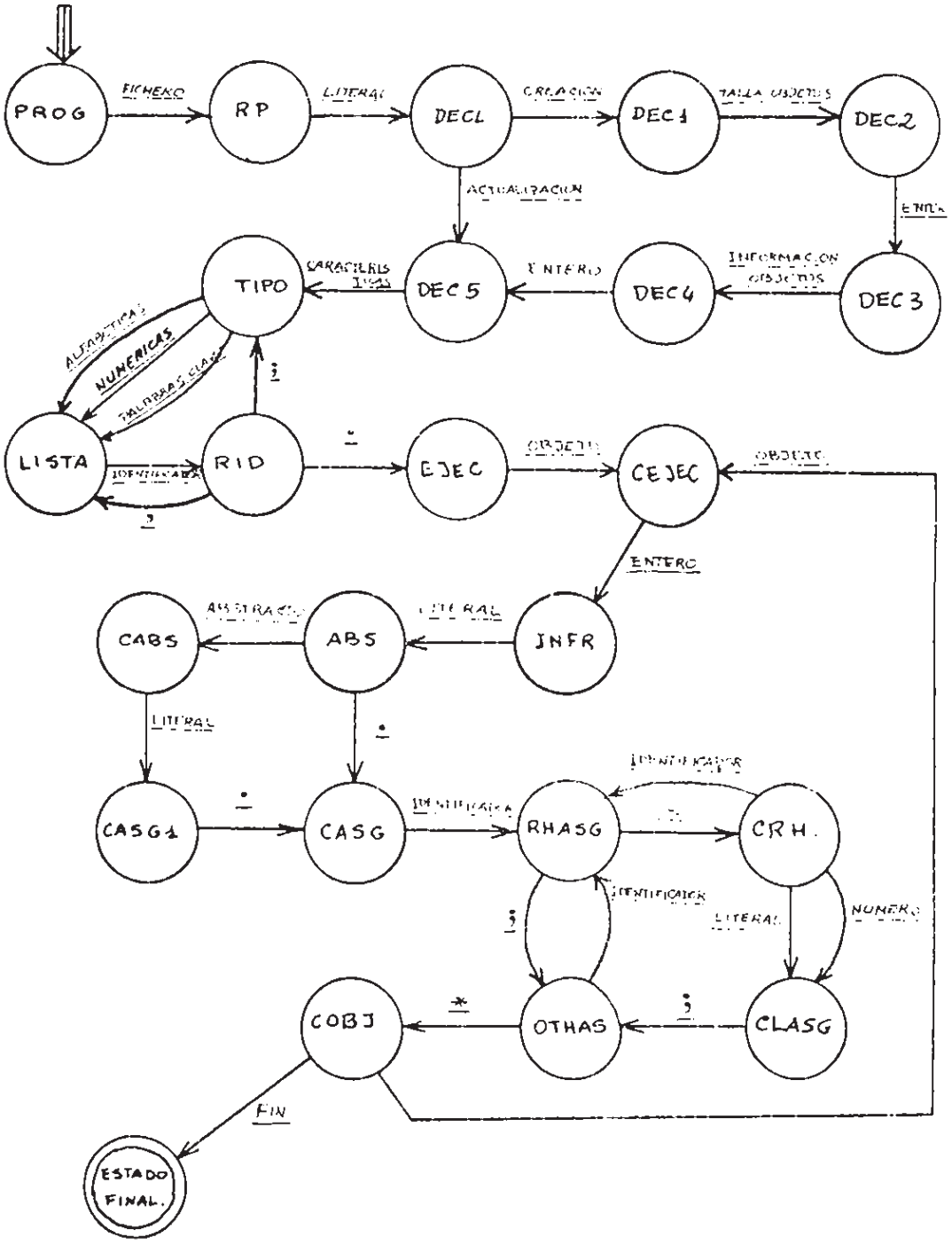
ES ('carácter') pregunta si el carácter sobre el que estamos es el que sirve de parámetro a la rutina Es un procedimiento booleano (*verdad* si el carácter-parámetro es igual que el carácter-chequeado, y *falso* si no) y no avanza al siguiente carácter, sino que permanece donde estaba

TEST ('cadena') comprueba la existencia en la cadena de *input* de la cadena de caracteres indicada como parámetro del procedimiento, emitiendo un mensaje de error en caso de que no exista la cadena-parámetro en el texto chequeado, y posicionándose en el primer carácter no blanco encontrado en el *input* después del chequeo, tanto si ha encontrado la cadena buscada como si no, asi que por lo menos avanza un número de posiciones en el texto igual a la longitud de la cadena buscada.

CHECK ('falso terminal') Sirve para testar la existencia de un falso terminal El pa-

DIAGRAMA DE TRANSICIONES DEL AUTOMATA

Al estado de ERROR se va si no se reconoce ningún terminal de los definidos para el estado en que se encuentra el autómata en un momento dado de su reconocimiento



rámetro entre comillas será una de las siguientes palabras:

LITERAL, si lo que buscamos es un literal
ENTERO, si lo que buscamos es un número entero
NUMERO si lo que buscamos es un número entero o real
IDENTIFICADOR si lo que buscamos es un nombre de característica

Después de reconocer un falso terminal, la rutina *check* salta todos los blancos que encuentra en el texto y nos deja posicionados en el primer carácter no blanco

Cada módulo de programación se compone, pues, de un testeo de elementos terminales o falsos terminales, de una bifurcación u otro módulo de programación y además de un conjunto de acciones semánticas

IV.3.2. *Acciones semánticas asociadas al intérprete.*

Las acciones que va a realizar el intérprete son, en realidad, el resultado de darle a la codificación en lenguaje fuente el sentido que el programador quería y la materialización en conjuntos de instrucciones ejecutables que efectúen exactamente lo que el programador deseaba.

Cada 'módulo' de programación lleva asociado un conjunto de acciones semánticas que se plasma en llamadas a la 'rutina de acciones semánticas', que nosotros llamamos SEMANT. Una llamada a SEMANT se materializa como SEMANT (*N*), siendo *N* un parámetro que nos indica el punto de entrada de SEMANT a donde queremos ceder el control. Así, SEMANT (10) indicaría que queremos entrar en el punto de SEMANT indicado por la etiqueta L10.

Las expresadas acciones tienen como objeto, lo mismo que el programa fuente, tres misiones principales:

— Preparar todo lo necesario para llevar a cabo y efectuar la construcción de las TABLAS de símbolos, directa e inversa.

— Construcción de la matriz booleana CARACTERISTICAS/OBJETOS y del mecanismo de direccionamiento de los objetos en el soporte físico.

— Construcción de las representaciones de los objetos, que son las que van a quedar grabadas formando el banco de información, conteniendo toda la información numérica, alfabética y de palabras-clave.

Vamos a ver ahora detenidamente cada uno de estos tres puntos

IV.3.2.1. *Construcción de las tablas de símbolos.*

Se van a construir con la parte declarativa del programa de creación del fichero. Van a contener todas las características susceptibles de ser manejadas en el banco que se va a crear.

Por medio de la TABLA DIRECTA podremos conocer el número de código de una variable, conociendo su nombre, y por medio de la TABLA INVERSA podremos conocer el nombre y el tipo de una variable sabiendo su número de código.

IV.3.2.1.1. *Tabla de símbolos inversa.*

Va a constar de dos vectores de 300 elementos de longitud cada uno, NOMINV y TIPINV.

NOMINV va a tener 300 elementos, cada uno de los cuales estará compuesto de seis caracteres, que completan una palabra y aquí podremos almacenar el nombre de las variables.

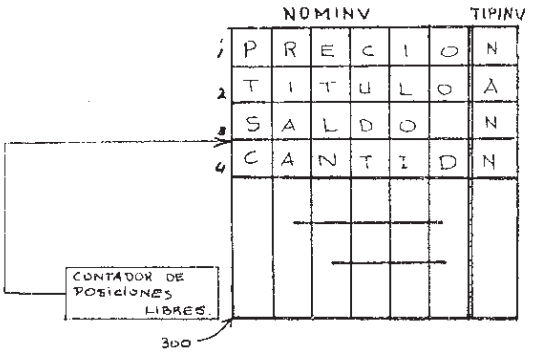
TIPINV será formado por elementos de un carácter y expresará el TIPO de la característica:

A si es de tipo alfabético
N si es de tipo numérico
P si es de tipo de palabra-clave

El número de código de la variable será el número que nos indica la posición de la variable dentro de los vectores NOMINV y TIPINV.

Según se van leyendo las declaraciones se van insertando las variables declaradas en esta tabla, por simple orden cronológico y, por consiguiente, se les va asignando un numero de código que depende del momento en que han sido declaradas. Se va actualizando en cada inserción un contador que nos indicará el primer lugar libre de la tabla, apto para recibir una nueva inserción.

El acceso a esta tabla es directo por el número de código (así, el numero de código de precio será 1, al estar en el primer lugar de la tabla el de TITULO, 2; el de SALDO, 3, y el de cantidad, 4) y secuencial por el nombre de la variable.



IV.3.2.1.2. *Tabla de simbolos directa.*

Va a ser una tabla de tipo HASH abierta. La entrada a la tabla se efectua por el nombre de la característica, al cual se le aplica una función HASH que genera una entrada en una TABLA PRIMARIA, que a su vez nos da una entrada en la tabla hash, correspondiente al grupo de sinónimos respecto a la

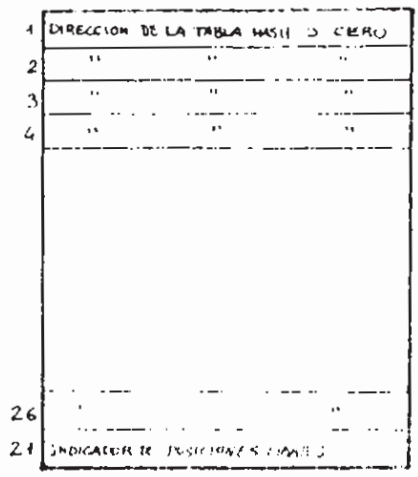
función mencionada a que pertenece la característica. Esta función de HASH va a tomar el nombre de la característica, y se va a fijar en el carácter inicial; esta inicial podrá ser una de las 26 letras del alfabeto; pues bien, si es una A, la entrada a la tabla primaria se efectuará en el primer lugar, si es una B, en el segundo, y si fuera una Z, en el último lugar. De aquí se desprende que la tabla primaria va a tener 26 entradas.

La tabla primaria tendrá el siguiente contenido:

Si el contenido de alguna de las entradas fuera cero (inicialmente lo son todos) significa que todavía no hay en la tabla de símbolos ninguna característica perteneciente al grupo de sinónimos determinado por la función hash para esa entrada.

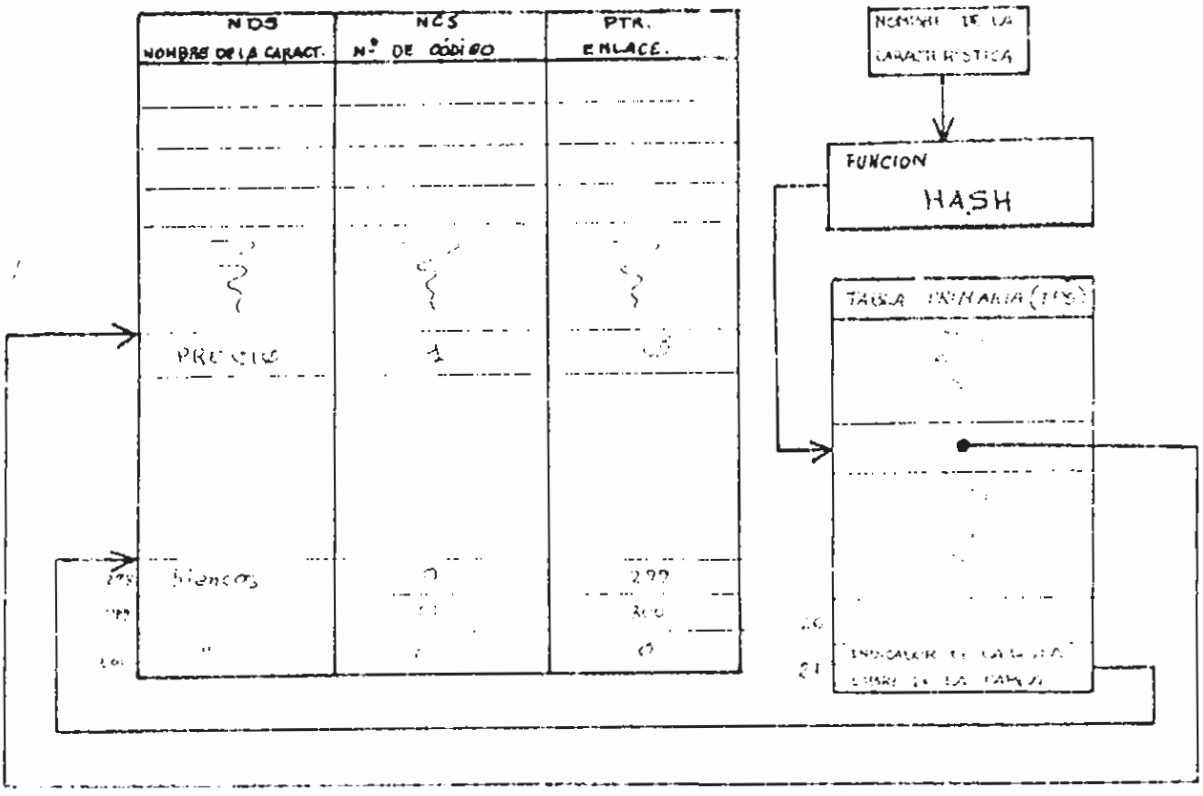
La tabla hash va a constar de tres vectores de 300 elementos cada uno.

Uno va a contener los nombres de las características, otro el numero de código correspondiente y el tercero va a ser el enlace con el siguiente lugar de la tabla ocupado por un sinónimo de la característica en cuestión.



Los sitios vacios de la tabla van enlazados en una 'lista libre' cuyo primer elemento viene indicado por la 27.ª entrada de la tabla primaria.

Veamos cómo estaria organizada la tabla hash:



Inicialmente toda la tabla hash será lista libre y estará ligada toda ella, siendo TPS (27) = 1.

Vamos a ver ahora cómo se van construyendo las dos tablas de símbolos, porque se van construyendo simultáneamente. El proceso sería:

A1 Se inicializan las dos tablas, haciendo:

- TPS rellena con CEROS, excepto TPS(27), que estará conteniendo un 1 (indicando el primer elemento de la tabla hash)
- NDS relleno a blancos. Será un conjunto de nom-

bres de seis caracteres cada uno, inicialmente a blancos

— NCS relleno a CEROS Serán los futuros números de código de las características

— PTR inicialmente señalarán a la siguiente fila de la tabla, serán:

— PTR (I) — I + 1 para I = 1, 2 299
PTR (300) = 0 Señalando el fin de la lista

— NOMINV rellena a blancos

— TIPINV reilena a blancos también

— Hacemos NCOD = 0

A2 Por cada declaración de nueva característica se hará:

— Se acude a NOMINV y hacemos NCOD = NCOD + 1 en NOMINV (NCOD) incluimos el nombre de la característica y su tipo en TIPINV (NCOD)

— Obteniendo así el número de código aplicamos al nombre de característica la función de hashing y obtenemos la entrada a la tabla primaria

— Si el contenido de la entrada fuera CERO, cogemos el primer elemento de la lista libre y lo encadenamos a dicha entrada desencadenándolo de la lista libre y modificando TPS(27) Incluimos entonces el nombre en NDS y el número de código (NCOD) en NCS y hacemos PTR — 0

Si no fuera cero el contenido de la entrada recorremos el grupo de sinónimos hasta hallar su final y operamos como hemos hecho antes, cogiendo el primer elemento de la lista libre para contener el nombre y número de código de la nueva característica

IV 2 2 *Construcción del mecanismo de direccionamiento secuencial de los objetos*

El proceso de consulta al banco de datos se realiza para todos y cada uno de los objetos en forma secuencial Ya en la presentación del trabajo comentamos que esto no era exactamente verdad, puesto que, en el momento en que la talla del banco fuera grande conduciría a un tiempo de respuesta prohibitivo (el tiempo de respuesta es proporcional al numero de objetos tratados)

Por tanto, el mecanismo de direccionamiento secuencial de los objetos sólo "ve" aquellos que de acuerdo con el programa de interrogación del usuario puedan interesarle

Para definir este direccionamiento recurrimos a dos mecanismos:

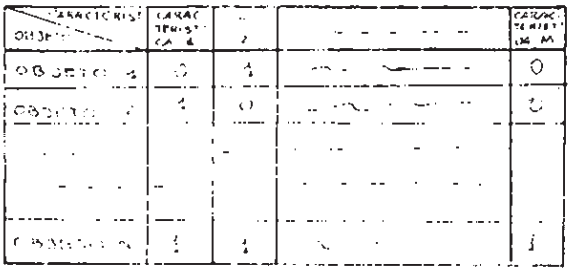
— Matriz booleana (objetos/características).

— Vector booleano del programa (programa/característica)

IV 2 2 1 *Matriz booleana*

El intérprete de creación del banco construye una matriz booleana que nos indica las características que posee cada objeto, así como los objetos que poseen una característica dada, en la forma:

matriz booleana MB [1 : NOBJTS, 1 : NCARACT]



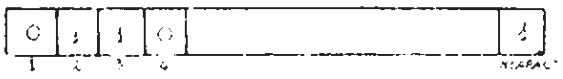
Esta matriz booleana juega el papel del fichero inverso típico de las aplicaciones de I.R

IV 2 2 2 *Algoritmos de direccionamiento*

Todo programa de interrogación es evaluado durante el análisis léxico-gráfico dando origen a un vector booleano que nos indica las características en el programa de interrogación:

vector booleano V B [1 : NCARACT]

PROGRAMA?



El subconjunto del banco de información que puede interesarle al usuario estará formado por aquellos objetos que tengan al menos una característica de las que aparecen en este vector booleano Este subconjunto lo podemos definir como:

$$BD = \text{OBJETOS} \quad \overline{\text{(AUBUC} \quad \text{UR)}}$$

siendo:

BD = banco de información que interesa al usuario

A, B, C, R = caracteristicas presentes en el programa (expresadas en VB)

I = NOT

U = OR inclusive

= complemento conjuntista (definido sobre el banco de información total)

Desde un punto de vista operativo esto quiere decir:

El mecanismo de direccionamiento secuencial de los objetos (a la ejecución) nos hace pasar del $OBJETO_{K-1}$ al $OBJETO_K$.

El $OBJETO_K$ interesará al usuario si se cumple que la variable booleana B (inicializada a FALSE) es verdad después del proceso:

for i = 1 step 1 until NCARCT do B: = BV VB [i] ∧ MB [K,i]

Si B es verdad el objeto será leído y tratado, no siéndolo en caso contrario.

Esto implica que tenemos que modificar el mecanismo de lectura visto en el punto: Estructura del programa objeto en el sentido:

IF (B) READC ( )

y generar el valor de B en el programa objeto.

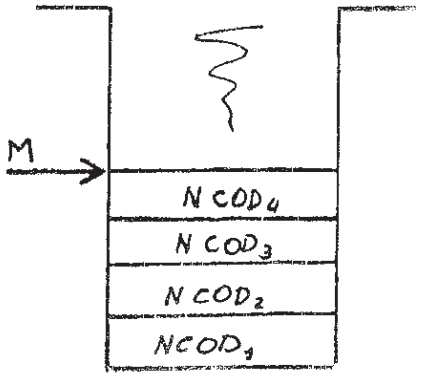
Otro mecanismo más adecuado de direccionamiento de los objetos, que tenga en cuenta la lógica del programa de interrogación, está siendo estudiado, pero por ahora consume un tiempo enorme, no resultando práctico.

IV.3.2.3. *Construcción de la representación de los objetos.*

Ya hemos visto en III.1 cómo se representaban los objetos que constituyen el Banco de Información. La construcción de esa representación es llevada a cabo por la rutina de acciones semánticas en las llamadas que se le hacen desde la FASE DE DESCRIPCION del Banco. Para cada OBJETO que encontramos en dicha fase de descripción habrá que crear un registro organizado con una estructura que acepte la información que nos proporciona la codificación-fuente. El registro y su estructura ya lo vimos en III.1. Las acciones semánticas lo que harán es ir rellenando los campos de ese registro con los valores dados en la fase de descripción.

IV.3.2.3.1. *Tratamiento de las informaciones numéricas.*

Las informaciones numéricas pueden ser sólo de dos tipos: el número del OBJETO o clave y valores de asignación de variables de tipo numérico. Para el primero, lo que se hace es simplemente darle ese valor al campo *Key* del registro. Para el resto, o sea, para las asignaciones y debido a que están permitidas las asignaciones múltiples del tipo PRECIO = COSTOS = PRECAM = 15; nos vamos a apoyar en una pila. Dicha pila va a ir conteniendo en cada asignación las variables que van a tomar un mismo valor. Vamos a ver lo que va a contener la pila y cómo se va a tratar:



Por cada variable que aparezca en una asignación se va a incluir en la pila su número de código y se va a incrementar el indicador de la cabeza de la pila, M, en una unidad. Una vez encontrado el valor a asignar se procede a darles a todas las variables insertas en la pila ese valor, decapitando en una unidad la pila en cada paso (M = M-1) hasta que la pila se queda vacía (M = 0). El proceso a seguir en cada caso sería:

A1. Aplicar la función de hash al elemento M de la pila. Hemos obtenido así la entrada en la tabla primaria del objeto.

A2. Buscar entre el grupo de sinónimos de la tabla hash del objeto si está la variable expresada por el número de código que tenemos. Si llegamos al final del grupo de sinónimos sin haber encontrado la variable (*), cogemos el primer elemento de la lista libre, indicado por CLL, lo desencadenamos de la lista libre, actualizando CLL y lo encadenamos al grupo de sinónimos en el que nos estamos moviendo. Si, por el contrario, nos encontramos con que existe ya el sitio reservado para esa variable, estamos ya situados; no necesitamos seguir y entonces (a partir de aquí el proceso es el mismo en ambos casos) almacenamos el valor en su sitio.

A3. Hacer M = M — 1. Si M = 0, hemos acabado, y si no, volver a A1.

(*) 0 si no existe aún grupo de sinónimos y la tabla primaria contiene un ∅ como dirección del grupo

IV 3 2 3 2 *Tratamiento de las informaciones alfanuméricas*

La primera información alfanumérica que nos encontramos es el literal 'descripción externa', definido ya en IV 2 1 2, y lo vamos a almacenar en el campo DESCP

Si existe ABSTRACTO tenemos otra información alfanumérica y ésta se guardaría en una zona de memoria para al final del tratamiento del objeto grabarlo en una ZONA DE ABSTRACTOS, junto con su longitud

El resto de la información de tipo alfanumérico que nos podemos encontrar nos proviene de las asignaciones a variables de este tipo de un literal (TITULO = 'EL CID')

También en este tipo de información se permite la asignación múltiple, y el modo de tratarla es exactamente igual que en el caso de información numérica, salvo en el momento de almacenar la información Veámoslo:

En información de tipo numérico un gránulo que iba a contener una variable constaba de:



Donde en información de tipo numérico va almacenado el valor, ahora vamos a almacenar el POINTER a la dirección del DEPOSITO DE CONTENIDOS, donde vamos a insertar la información alfanumérica Esta inserción se hará a partir de la dirección del DEPOSITO DE CONTENIDOS (DC) que señala CLDC

Como DC está dividido en gránulos encadenados de tres palabras (18 caracteres) de información y una de enlace, el literal puede ser todo lo largo que permita la lista libre del depósito de contenidos, encadenándose los gránulos que hagan falta El enlace del ultimo gránulo asociado a una variable será ⊘ y habrá que actualizar CLDC tras cada nueva inserción

IV 3 2 3 3 *Tratamiento de las informaciones de palabras-clave*

Para tratar las palabras clave se necesita crear el vector de posiciones de cada una de ellas dentro del texto del abstracto, para lo cual cada vez que aparezca una variable de este tipo en la fase de descripción, hallaremos su vector de posiciones y lo escribiremos en el DEPOSITO DE CONTENIDOS La obtención del vector de posiciones es como sigue:

— Se obtiene el abstracto del objeto en cuestión a partir del campo 'Pointer contenido abstracto' Se investiga lo mismo que para las características alfanuméricas, el lugar del depósito de contenidos a partir del cual puede ser empezado a almacenar la información asociada con la característica

— Cada palabra del gránulo va a contener un entero, que va a representar una de las posiciones de la variable en el texto y se halla examinando el abstracto, palabra por palabra y comparando éstas con el nombre de la variable

Por ejemplo, sea el abstracto de un objeto:

ABSTRACTO FISICA NUCLEAR FISICA AIRE ELECTRONICA FISICA

El vector de posiciones de la palabra FISICA es (1, 3, 6)

El vector de posiciones puede ser tan largo como permita la lista libre del depósito de contenidos, pero lo vamos a limitar a 12 posiciones, ya que es muy raro que en un abstracto se repita tantas veces una palabra clave y también por facilidad a la hora de comparar los vectores de posición de diversas palabras-clave

El proceso de creación de la representación de una palabra-clave en el objeto será:

A1 El mismo proceso que para los alfanuméricos hasta el momento en que se ha de rellenar, con el valor que se le ha asignado a la variable alfanumérica, el depósito de contenidos

A2 Una vez hallado el vector de posiciones, transcribir las posiciones distintas de ⊘ al depósito de contenidos, utilizando los gránulos que sean necesarios (un gránulo para cada tres o menos posiciones) y encadenando esta lista de gránulos desde el sitio que corresponde a la dirección del contenido de la variable en la tabla hash del objeto

Después de haber visto cómo se representan los objetos, hemos de aclarar que, previamente a todos los procesos que hemos

descrito, cada vez que se va a empezar a describir un objeto hay que inicializar de la siguiente forma:

— Descripción del objeto a blancos

— Pointer contenido abstracto a CERO

— CLL a 21

— Tabla primaria (los 20 primeros de OBJET) a CERO

— El resto del objeto hasta el campo CLDC, a CEROS
La longitud del vector OBJET es de 191 elementos de una palabra cada uno (*)

— CLDC a UNO

— DC (depósito de contenidos) a blancos (**)

(*) Ahora hay que encadenar la lista libre inicial, que será todo este campo. El encadenamiento será:
I ← CLL;
1 : OBJET (I + 2) = I + 3;
I ← I + 3;
Si I + 2 > 191 entonces se acabó el encadenamiento hago OBJET (I + 2) = ∧; y si no, vuelve a 1

(**) Hay que encadenar ahora la lista libre del depósito de contenidos, que inicialmente será todo DC. Se sigue un procedimiento análogo al seguido en (*)

BIBLIOGRAFIA

C. PAIR: "Les structures de l'information". Alès 1971

SISTEMA PARA LA CREACION Y MANTENIMIENTO DE BANCOS DE INFORMACION

POR

I. RAMOS

CALIXTO LOZANO

A. MUÑOZ

PUBLICADO EN:

REVISTA DE INFORMATICA Y AUTOMATICA

AÑO VIII. - NUM. 24 - 1975

SISTEMA PARA LA CREACION Y MANTENIMIENTO DE BANCOS DE INFORMACION

(PARTE III)

Por I RAMOS (*)
CALIXTO LOZANO (**)
A MUÑOZ (**)

V LENGUAJE DE USO DEL BANCO Y COMPILADOR ASOCIADO

V 1 *Presentación del Problema Ejemplo*

Nos encontramos con el problema ahora de disponer de un lenguaje que nos permita expresar de una manera clara y precisa lo que queremos hacer con los objetos en las operaciones de manejo del banco, tales como:

— Consulta.
— Modificación.
— Inserción
— Eliminación

Deberemos poder disponer de instrucciones de tipo condicional y de tipo operativo para efectuar procesos como el que indicamos a continuación

Continuando con el ejemplo del banco de ordenadores electrónicos visto en el punto IV 1 y que veremos también en V.2 3, una consulta típica al banco podría ser:

> Si precio < 100 000 000 y año > 1971 y COMPATIBLE, IBM-360. entonces si ciclo-memoria < 1 entonces escribir (abstracto del objeto). sino si organización de memoria = 'PALABRA' y tamaño-memoria > 75. entonces escribir (descripción del objeto). sino finsi finsi finsi

Con estas necesidades veremos cómo ha de ser el lenguaje apto para expresar tales procesos.

V 2 *Lenguaje de interrogación del banco.*

Toda la información que tenemos almacenada en el banco va a ser tratada con vistas a la recuperación; para expresar el tratamiento a que va a ser sometida necesitaremos crear un lenguaje que nos permita escribir una codificación clara, legible y de fácil traducción, y que, además, exprese todas las facilidades necesarias

Un tratamiento de recuperación se define para un elemento del fichero, y se itera para todos los elementos que lo componen.

El tratamiento de un elemento va a ser preguntar por una serie de características de los objetos, compararlos con unos valores, expresiones u otras características o contadores, y dependiendo del resultado de dichas comparaciones efectuaremos unas acciones determinadas en el objeto Estas acciones deberán poder ser: imprimir características, abstracto, descripción, objeto o todo el objeto completo, o bien eliminar características del objeto, agregar al objeto nuevas características o modificar características ya existentes

El lenguaje interrogación tendrá que soportar todas estas facilidades. Sus instrucciones van a ser de tres tipos: condicionales, incondicionales y de proceso (llamadas a subprograma), segun vamos a ver en la descripción informal siguiente:

V 2 1 *Descripción informal.*

Constituye el Manual de Uso del Lenguaje, y es como sigue:

Instrucciones de llamadas a subprogramas

Instrucción AGREGAR

Su forma será [etiqueta]:

AGREGAR ((IDENT$_1$, EXPR$_1$) (IDENT$_2$, EXPR$_2$))

IDENT$_i$ serán nombres de características de los tres tipos posibles

(*) Instituto de Informática Centro de Cálculo de la Universidad de Madrid.
(**) Instituto de Informática

$EXPR_i$ serán:

Para características numéricas, expresiones aritméticas.

Para características alfabéticas, literales.

Para palabras clave, la palabra ABSTRACTO.

Lo que hace la instrucción para características numéricas es buscar un sitio en el objeto para la característica de nombre $IDENT_i$ y asignarla el valor dado por la evaluación de la expresión $EXPR_i$.

Ejemplo:

AGREGAR ((PRECIO, 50))

agrega al objeto la característica PRECIO con valor de 50.

Lo que hace la instrucción para características alfabéticas es buscar un sitio en el objeto para la característica de nombre $IDENT_i$ y darle como valor el literal asociado.

Ejemplo:

AGREGAR ((EDITORIAL, 'AE'))

agrega al objeto la característica editorial, poniendo un *pointer* a DC donde estará el valor AE.

Lo que hace la instrucción para características de palabras clave es buscar un sitio en el objeto para la palabra clave de nombre $IDENT_i$ y hallar su vector de posiciones con respecto al abstracto y ponerlo en DC.

Cada pareja IDENT, EXPR viene encerrada entre paréntesis y separados por una coma sus dos elementos.

Puede haber un número ilimitado de parejas.

Si el objeto está lleno y no se puede agregar la característica recibiremos un mensaje.

Ejemplo:

AGREGAR ((NETO, PRECIO-RAPPEL) (EDITORIAL, 'AE') (SOCIEDAD, ABSTRACTO))

Instrucción MODIFICAR.

Su forma será [etiqueta]:

MODIFICAR ($PARAM_1$, $EXPR_1$) . $PARAM_n$, $EXPR_n$))

Los parámetros $PARAM_i$ pueden ser nombres de característica numérica o alfabética o contadores.

Si la característica es numérica o es un contador, $EXPR_i$ es una expresión aritmética.

Si la característica es alfabética, $EXPR_i$ es un literal.

Lo que hace la instrucción para características numéricas es buscar la característica dentro del objeto, y se modifica su valor poniendo en su lugar la evaluación de la expresión $EXPR_i$.

Si $PARAM_i$ es un contador, la acción que desencadena la instrucción es modificar el valor del contador con el resultado de la evaluación de la expresión $EXPR_i$.

Ejemplo:

MODIFICAR ((PRECIO, PRECIO+20 * PRECIO/100) (W01, W01 + 1))

Si $PARAM_i$ es una característica alfabética se busca la característica dentro del objeto y se modifica su valor con el del literal.

Ejemplo:

MODIFICAR ((AUTOR, 'J. L. AROSA'))

Si quisiéramos modiifcar una característica que no contiene el OBJETO, tendríamos un mensaje de error.

Instrucción ELIMINAR.

Su forma será [etiqueta]:

ELIMINAR ($IDENT_1$, , $IDENT_n$)

$IDENT_i$ son nombres de características de los tres tipos.

Si la característica es numérica, la acción que se lleva a cabo es eliminar del OBJETO la característica y añadir el nuevo espacio obtenido en OBJET a la lista libre.

Si la característica es alfabética o de palabra clave, la acción que se lleva a cabo es eliminar del objeto la característica y añadir el espacio obtenido de OBJET a su lista libre, y el espacio obtenido en DC a su lista libre.

Ejemplo:

ELIMINAR (PRECIO, AUTOR, SOCIEDAD)

Instrucción CREAR.

Su forma será [etiqueta]:

CREAR ($CONTADOR_1$, $EXPR_1$) . ($CONTADOR_n$, $EXPR_n$))

$CONTADOR_i$ son contadores.

$EXPR_i$ son expresiones aritméticas.

La acción que realiza esta instrucción es inicializar contadores, y se puede utilizar también para modificarlos.

Ejemplo:

CREAR ((W01, 0) (W02, W01) (W03, 1))

Instrucción IMPRIMIR

Su forma será una de las siguientes, con o sin etiquetas:

— IMPRIMIR ("literal"): Donde literal es cualquier cadena que no contenga la comilla La longitud máxima del literal va a ser de 56 caracteres, para mayor facilidad en la generación del código objeto Esta instrucción imprime el literal en una línea de impresión

— IMPRIMIR (expresión): Donde expresión es una expresión aritmética que se evaluará y se imprimirá su valor en una línea de impresión

— El caso más sencillo de expresión es una característica que puede ser numérica o alfabética; si es alfabética se cogerá su valor de DC y se imprimirá

— IMPRIMIR ( ): Indica que lo que queremos imprimir es el campo de descripción asociado al objeto

— IMPRIMIR (@ TOTREG): Indica que se imprima la totalidad del objeto, su campo de descripción, todas sus características (su nombre y su valor), excepto las de palabras clave

— IMPRIMIR ($ ABSTRACTO): Indica que se imprima el abstracto

Si queremos imprimir el nombre de una característica lo deberemos poner como un literal

Ejemplo:

```
IMPRIMIR ('PRECIO');
IMPRIMIR (PRECIO);
```

Esto hará que se imprima la palabra PRECIO y el valor de la característica PRECIO

Instrucción CONDICIONAL

Su forma general será [etiqueta]:

```
Si expresión lógica ENTONCES conjunto
  de instrucciones
SINO conjunto de instrucciones FINSI
```

— Las palabras SI, ENTONCES, SINO y FINSI son obligatorias en el texto

— La expresión lógica va a ser un conjunto de condiciones simples unidas por uno de los operadores O o Y, y precedidas o no cada una de ellas por un NO.

Las condiciones simples pueden ser de tres tipos:

a) Con características numéricas:

Están formadas por características numéricas, contadores o números afectados o no de los operadores +, —, * y /, y relacionados con los operadores >, <, =, ≠

Ejemplos:

```
PRECIO < 50
PRECIO > 50
CANT* PRECIO/1000 > 80
CANT > STOCK
```

b) Con características alfabéticas:

Están formadas por características alfabéticas y literales relacionadas entre sí por el operador =

Ejemplos:

```
NOMBRE = 'GONZALO'
APEL1 = 'LOZANO'
NO (APEL2 = 'MUÑOZ')
```

c) Con características de palabras clave:

Están formadas por palabras clave afectadas o no del operador ¬, y relacionadas entre sí o no por los operadores — o ,

Ejemplos:

```
ELECTRONICA
¬ ELECTRONICA
ELECTRONICA — ¬ DIGITAL
ELECTRONICA, SEMICONDUCTORES
```

Todos los ejemplos puestos de condiciones simples son válidos, y pueden ir cada una de las condiciones simples unidas a otras por los operadores Y, O.

Ejemplo:

```
AUTOR = 'GRIES' y AÑO > 1973 y (COMPILACION
        o SISTEMAS-OPERATIVOS)
```

Los operadores que se utilizan con características numéricas y alfabéticas son de sobra conocidos por todo el mundo, pero no tanto con los operadores entre palabras clave:

La aparición de una palabra clave solamente en una condición simple quiere indicar si se encuentra esa palabra clave entre las del abstracto; en el ejemplo anterior:

```
Si ELECTRONICA ENTONCES
```

Indica que si ELECTRONICA aparece en el abstracto entonces se hará tal cosa.

Si la palabra clave va afectada del operador ¬ (unario) indica la ausencia de la palabra clave del abstracto

El operador — entre dos palabras clave no afectados de ¬ indica la presencia de la concatenación de las dos palabras clave en el abstracto

Si va afectada de ¬ la segunda, entonces, indica la presencia de la concatenación de la primera palabra clave con otra que no sea la segunda palabra clave

El operador, entre dos palabras clave indica la presencia de las dos palabras clave relacionadas en ese mismo orden dentro del abstracto:

— El conjunto de instrucciones que vienen detrás del ENTONCES puede contener una o varias instrucciones condicionales El numero máximo de niveles de instrucciones condicionales está limitado a 30, para no crear una pila de etiquetas demasiado grande El conjunto de instrucciones no puede ser vacío, tiene que haber por lo menos una instrucción

— El conjunto de instrucciones que vienen detrás del SINO es análogo al asociado al ENTONCES, con una salvedad Este conjunto de instrucciones sí que puede ser vacío, pudiendo acabar una instrucción condicional en SINO FINSI

Ejemplo de instrucción condicional:

```
SI precio<50 y NO (precio<100) y AUTOR='PEPE' y MATARATOS o PASATIEMPOS
ENTONCES Imprimir (Autor); Imprimir ('Mataratos');
         Imprimir (Abstracto)
SINO SI CANTIDAD>100 ENTONCES MODIFICAR (CANTIDAD, 50)
                     SINO FINSI
FINSI
```

V.2.2. *Descripción formal*

Las instrucciones del lenguaje interrogación vienen descritas por la siguiente gramática:

<PRN> → <INSTN> <PC> <CONTPRN> <INTE>

<INSTN> → <ETIQ> : <INSTN1> | <INSTN1>

<INSTN1> → <SSPN> | <CONDN> | <INCONDN> | ∅

<PC> → ;

<CONTPRN> → <INSTN> <PC> <CONTPRN> | ∅

<INTE> → ?

<SSPN> → AGREGAR(<LP1>) | MODIFICAR (<LP11>) | ELIMINAR (<LP2>) |
IMPRIMIR (<LP3>) | CREAR (<LP4>)

<CONDN> → <SI> <LTCNDN> <SINON> <FSI>

<SI> → SI

<LTCNDN> → <CNDN> <ENTONCES> <INSTN> <CLCN>

<CNDN> → <CNDE> <RCNDN> | NO <CNDN> <RCNDN> | (<CNDN>) <RCNDN>

<RCNDN> → <OPL> <CNDN> | ∅

<CNDE> → <EXPRES> <OPR> <EXPRES> | <IDA> = <LITERAL> |
<OPC> <IDC> <OIDC>

<ENTONCES> → ENTONCES

<CLCN> → ; <INSTN> <CLCN> | ∅

<SINON> → <SN> <CTSN>

<SN> → SINO

<CTSN> → <INSTN> <CLCN> | ∅

<FSI> → FINSI

<OPL> → Y | O

<OPR> → < | = | ≠

<OPC> → ¬ | ∅

<OIDC> → <OPC1> <IDC> <OIDC> | ∅

<OPC1> → . <OPC> — <OPC>

<INCONDN> → BIFURCAR A <ETIQ>

<ETIQ> → <DIGITO> <ETIQ> | ∅

<LP1> → (<CLP>

<CLP1> → <ID1> <CM> <EXPRES>) <CNP1> | <IDA> <CM> ' <LITERAL> ') <CNP1> |
<IDC>) <CNP1>

<CNP1> → <LP1> | ∅

<LP11> → (<CLP11>

<CLP11> → <ID11> <CM> <EXPRES>) <CNP11> | <CONTADOR> <CM1> <EXPRES>) <CNP11> |
<IDA> <CM> ' <LITERAL> ') <CNP11>

<CNP11> → <LP11> | ∅

<CM1> → ,

<ID1> → <ID>

<ID11> → <ID>

<LP2> → <ID2> <CONTP2> | <IDA> <CONTP2> | <IDC> <CONTP2>

<CONTP2> → <CM1> <ID2> <CONTP2> | ∅

<LP3> → <EXPRES> ' <LITERAL> ' @ TOTREG ∅ ; $ ABSTRACTO

<LP4> → (<CLP4>

<CLP4> → <CONTADOR> <CM1> <EXPRES>) <CNP4>

<CNP4> → <LP4> | ∅

<CM> → '

<EXPRES> → <TERMINO> <CLX>

<CLX> → <OPAD> <EXPRES> | ∅

<OPAD> → + | —

<TERMINO> → <FACTOR> <CLY>

<CLY> → <OPMU> <TERMINOS> ∅

<OPMU> → * | /

<FACTOR> → <NUMERO> | <ID> | <CONTADOR> | (<EXPRES>)

<ID2> → <ID>

<IDA> → <ID>

<NUMERO> → <NUMER> <MENOS> <NUMER>

<NUMER> → <ENTERO> <CTNUME>

<ENTERO> → <DIGITO> <CTNUM>

<CTNUM> → <DIGITO> <CTNUM> | ∅

<CTNUME> → <ENTERO> | ∅

<MENOS> → —

<ID> → <LETRA> <CONTID>

<CONTID> → <LETRA> <CONTID> | <DIGITO> <CONTID> | ∅

<CONTADOR> → W <DIGITO> <DIGITO>

<LITERAL> → <LETRA> <LITERAL> | <DIGITO> <LITERAL> | <SE> <LITERAL> | ∅

<LETRA> → A | B | ... | Z

<DIGITO> → 0 | 1 | 2 | ... | 9

<SE> → @ | (|) | $ | . | ! | ...

<IDC> → <ID>

V 2 3. *Ejemplo de programa*

Supuesto creado el banco de datos de ordenadores electrónicos de todas las firmas constructoras de los ultimos cinco años

Una consulta a esta información podría ser, por ejemplo, sacar una relación de todos los computadores que cumplen las siguientes características:

— El año de salida de fábrica sea después de 1972.
— El precio total sea menor de 90 millones de pesetas
— El ciclo de memoria sea de 0,75 y el numero medio de instrucciones por segundo sea de 700 000.
— Si es de palabras tenga 750 K.
— Si es de octetos tenga 3 000 K.
— Si es de sextetos tenga 4 200 K, por lo menos
— Tenga más de 2 CPU's, 10 cintas y 5 discos
— Sea compatible con el IBM 1401 o IBM 360.
— Tenga compilador de ALGOL y LISP preferentemente

```
CREAR ((W01, 1972) (W02, 0));
SI AÑO < 1972 ENTONCES BIFURCAR A 1 SINO
SI precio soft + precio hard < 90 000 000 o precio soft + precio hard > 80 000 000
ENTONCES SI ciclo M < 0,75 y N INSTRUCT > 700 000 ENTONCES
        SI (ORGMEN = PALABRAS' y NBITSW > 32 y NBITSW < 36 y SIZEM > 750) 0
           ((ORGMEM = BYTES') y ((BYTES = OCTETOS' y SIZEM > 3000) 0
           (BYTES = 'SEXTETOS' y SIZEM > 4200)) ENTONCES
           SI NCPU > 2 y NTAPE > 10 y NDISC > 5 ENTONCES
           SI COMPATIBLE, IBM-1401 ENTONCES AÑADIR ((IBM-360, ABSTRACTO));
        BIFURCAR A 2 SINO SI COMPATIBLE, IBM- 360, ENTONCES BIFURCAR A 2 SINO FINSI FINSI
           SINO FINSI
        SINO FINSI
        SINO FINSI
SINO FINSI; BIFURCAR A 3 FINSI

2: IMPRIMIR ($ ABSTRACTO);
   IMPRIMIR (AÑO);
   IMPRIMIR (PRECIO SOFT + PRECIO HARD);
   IMPRIMIR (CICLOM);
   IMPRIMIR (NINSTRUC);
```

```
   IMPRIMIR (ORGMEM);
   IMPRIMIR (SIZEM);
   IMPRIMIR (NCPU);
   SI ALGOL Y LISP ENTONCES IMPRIMIR ('ALGOL Y LISP') SINO FINSI;
   :BUFURCAR A 3;
1: IMPRIMIR (AÑO);
   IMPRIMIR ('ANTIGUO');
   BIFURCAR A 3;
3: IMPRIMIR ('ORDENADOR');
   MODIFICAR (W02, W02+1);
   IMPRIMIR (W02); ?
```

Este programa daría como resultados:

El abstracto de los ordenadores que cumplen las características pedidas, el año de fabricación, precio, ciclo de memoria, número de instrucciones, organización de memoria, tamaño de la memoria y número de CPU's.

Si el año de fabricación es anterior a 1972 imprimirá al año y el literal ANTIGUO y ORDENADOR, número de ordenador.

Si el año de fabricación es correcto imprimirá ORDENADOR, número ordenador.

V.3. *Compilador del lenguaje interrogación.*

V.3.1. *Tipo de compilador.*

Dadas las características del lenguaje de interrogación que puede ser descrito mediante una gramática determinista, el compilador será del tipo más adecuado para este tipo de gramática, es decir, constará de un analizador TOP-DOWN, determinístico LL (K), lo cual implica que el análisis se hace de arriba abajo siguiendo la gramática del Lenguaje, cuya raíz es PRN siguiendo las ramas de un árbol hasta la sentencia fuente y un generador que va generando código objeto a medida que va analizando, realizando las acciones semánticas correspondientes.

La construcción del compilador en un lenguaje de alto nivel, tal como el ALGOL, permite una programación muy estructurada y modular.

Por cada regla de producción de la gramática se construye una procedure, cuyo nombre es el de la parte izquierda de la regla de producción.

El contenido de la procedure se forma con la parte derecha de la regla de producción, según las siguientes leyes de formación:

a) Si hay alternativas en la regla de producción:

*a*1) Decidir por qué alternativa hemos de continuar preguntando por los posibles terminales por los que puede empezar cada una de las alternativas.

b) No hay alternativas:

*b*1) Por cada no terminal generar una llamada a la procedure, cuyo nombre es el del no terminal.

*b*2) Por cada terminal generar una llamada a una procedure que testee el terminal.

*b*3) Generar una llamada a una procedure semántica, donde se generará lo que corresponda para el programa objeto.

Para decidir por qué alternativa de una regla de producción se hace por medio de un procedimiento ES (carácter), que es un procedimiento booleano que devuelve el valor de verdad si el carácter corriente a analizar es por el que se pregunta y falso en caso contrario.

Para testear por un terminal se hace por medio de una procedure TEST (carácter) que testea si el carácter corriente a analizar es el del parámetro de la procedure y avanza al carácter siguiente a analizar; si no, escribe un asterisco en la posición errónea y pone una variable de generación de programa objeto a false y avanza al siguiente carácter a analizar.

V.3.2. *Forma general del programa objeto.*

El programa objeto deberá ejecutarse para todos los registros del fichero, y procesando cada registro hasta el fin de fichero, y procesando cada registro de acuerdo con el proceso que indica el programa de interrogación.

Por tanto, la estructura del programa objeto será:

```
A2:
LEER REGISTRO DEL FICHERO (SI FIN DE FICHERO IR a A1)
PROCESO DEL REGISTRO
GO TO A2
A1 : FIN
```

Mas concretamente, el código FORTRAN sería:

```
INTERGER CLL
COMMON OBJET (191), CLL
INTERCER KEY, PCA  CLDC. W(10). DESCP(25), DC(228)
3 READ 7,10000) END=2) KEY, DESP (25). PCA, OBJET (191), CLL, CLDC. DC(228)
```

Procesó del registro segun el programa INTERROGACION:

```
GOTO 3
2 STOP
10000 FORMAT ( )
END
```

V 3 3 *Representación de los objetos del lenguaje de interrogación*

En nuestro lenguaje de interrogación aparecen los siguientes objetos externos:

- — Constantes.
- — Contadores
- — Características
- — Etiquetas

que son transformados, teniendo en cuenta que nuestro lenguaje objeto es FORTRAN en los siguientes objetos externos del lenguaje objeto:

LENGUAJE FUENTE		LENGUAJE OBJETO
CONSTANTE 35	TRANSLITERACION →	CONSTANTE 35
CONTADOR Wj	SUSTITUCION →	ELEMENTO DEL VECTOR DE CONTADORES W [j]
CARACTERISTICA PRECIO	SUSTITUCION →	ELEMENTO DEL VECTOR OBJETO [l] (l se define según la tabla de símbolos (a la compilación, en forma estática) y la estructura del objeto real (a la ejecución, en forma dinámica))
ETIQUETA 26	TRANSLITERACION →	ETIQUETA FORTRAN 26

Hemos añadido ejemplos triviales en cada caso.

Si el lenguaje objeto fuera distinto (por ejemplo, lenguaje ensamblador o máquina) deberíamos tener en cuenta los objetos proporcionados por el nuevo lenguaje para realizar esta transformación Esto añadiría quizá una representación más laboriosa, pero conceptualmente nada nuevo

V 3 4. *Algoritmos de manipulación de las tablas de símbolos*

Las tablas de símbolos (directa e inversa) del banco de datos se consultan desde el programa compilador para: 1 º Obtener el numero de código de una característica 2 º Para obtener el tipo de una característica:

1.º Se quiere obtener el número de código de una característica porque el programa objeto trabaja con números de código en vez de con nombres de característica.

El algoritmo de consulta del número de código es el siguiente:

A1. J = función hash (nombre de característica).
A2. Si TPS(J) igual 0, ir a A9.
A3. K = TPS(J).
A4. Si NDS(K) igual nombre de característica, entonces NCOD = NCS(K), ir a A9.
A5. K = PTR(K).
A6. Si K igual 0, entonces ir a A7; si no ir a A4.
A7. Escribir (el Identificador, nombre de característica, no existe.)
A8. NCOD = 0.
A9. Fin.

Mediante una función *hash* del nombre de característica obtenemos la entrada a una tabla *hash* (TPS), tabla primaria de símbolos cuyo contenido será un *pointer* a la entrada correspondiente al grupo de sinónimos del nombre de característica en la tabla de símbolos; por tanto, investigamos ese grupo de sinónimos siguiendo los enlaces hasta encontrar la entrada buscada.

Si el primer nombre del grupo de sinónimos NDS (K) es el buscado hemos terminado, si no vamos a la entrada que nos indica el *pointer* PTR (K) y volvemos a hacer la misma pregunta hasta terminar el grupo de sinónimos o hasta encontrarle.

2.º El tipo de una característica se requiere para saber el contexto en que está enclavada.

Por ejemplo, si una característica es de tipo numérico su contexto ha de ser nada o características de tipo numérico, contadores o números ligados entre sí por los operadores +, —, *, /.

Se utiliza varias veces para saber por qué alternativa de las de una regla de producción hemos de empezar el análisis.

El algoritmo es el siguiente:

A1. Hallar el NCOD de la característica mediante el algoritmo anterior.
A2. TIPO = TIPINV(NCOD).

TIPINV es la entrada al tipo en la tabla de símbolos inversa.

V.3.5. *Lenguaje objeto.*

Hemos escogido como lenguaje objeto el FORTRAN. Las razones han sido estrictamente de comodidad y fácil comprensión de los programas objeto generados. Es un lenguaje lo suficientemente conocido y utilizado como para que le dediquemos mayor atención.

Un punto a su favor (además de su enorme divulgación) es la presencia de compiladores muy eficientes y probados sobre la gran mayoría de los ordenadores actuales.

Esto hace que el sistema implementado sea fácilmente trasladable a otro ordenador sin realizar cambios importantes.

V.3.6. *Compilación de las distintas instrucciones.*

Toda instrucción fuente del lenguaje INTERROGACION genera una o varias en lenguaje objeto.

Vamos a ver la compilación de cada una por separado:

V.3.6.1. *Instrucciones de llamadas a subprogramas.*

Sentencia AGREGAR.

Su formato es:

AGREGAR ((IDENT$_1$, EXPR$_1$) [(IDENT$_n$, EXPR$_n$)])

según se ha visto en el punto V.2.1.

Por cada par (IDENT$_i$, EXPR$_i$) se genera:

a) Si IDENT$_i$ es una característica de tipo numérico, cosa que se puede saber mediante la procedure TIPO, que accede a la tabla de símbolos inversa, entonces habrá que comprobar que EXPR$_i$ es una expresión aritmética con características numéricas y generar una llamada a la subrutina INSTHS, que ejecutará lo necesario para agregar la característica al objeto:

```
CALL INSTHS (NCOD, EXPR)
```

donde NCOD es el número de código de la característica que se va a agregar.

b) Si IDENT$_i$ es una característica de tipo alfabético hay que comprobar que lo que viene a continuación es un literal, y generar una llamada a una subrutina INSTHA, que en tiempo de ejecución agregará la característica en cuestión con el valor del literal:

```
CALL INSTHA (NCOD, 'literal')
```

donde NCOD es el número de código de la característica que se va a agregar.

c) Si IDENT$_i$ es una característica de palabra clave, entonces EXPR$_i$ debe ser la palabra *abstracto,* y se genera la llamada a

una subrutina, que en tiempo de ejecución hallará el vector de posiciones de la palabra clave en el abstracto y agrega esa característica al objeto con el vector de posiciones La llamada será:

CALL INSTHP (NCOD)

Sentencia MODIFICAR

Su formato es:

MODIFICAR ((PARAM$_i$, EXPR$_i$) [( )])

segun se ha visto en el punto V 2.1

El formato es comprobado por el analizador y por cada par (PARAM$_i$, EXPR$_i$) se generará:

a) Si PARAM$_i$ es una característica de tipo numérico se hallará su numero de código (NCOD) y se generará una llamada a la subrutina MODIFY que en tiempo de ejecución cambiará el valor de la característica por el de la expresión:

CALL MODIFY (NCOD, EXPR)

EXPR debe ser una expresión aritmética que no contenga características alfabéticas ni de palabras clave.

b) Si PARAM$_i$ es una característica de tipo alfabético se hallará su numero de código (NCOD) y EXPR$_i$ deberá ser un literal y se generará una llamada a una subrutina MODIFA que, en tiempo de ejecución, cambiará el valor de la característica por el de literal:

CALL MODIFICA(NCOD, 'leteral')

c) Si PARAM$_i$ es un contador, entonces EXPR$_i$ es una expresión aritmética que no deberá contener ni características alfabéticas ni de palabras clave y se generará:

ω (número) = EXPR$_i$

Sentencia ELIMINAR

Segun se ha visto en el punto V 2 1 el formato es:

ELIMINAR (IDENT$_1$ [IDENT$_2$, , IDENT$_n$])

El formato es comprobado por el analizador y por cada IDENT$_i$ se generará:

a) Si IDENT$_i$ es una característica numérica se hallará su numero de código (NCOD) y se generará una llamada a la subrutina SUPTHS que, en tiempo de ejecución, quita dicha característica del objeto:

CALL SUPTHS (NCOD)

b) Si IDENT$_i$ es una característica alfabética o de palabra clave se generará una llamada de subrutina SUPTHA, que suprimirá la característica indicada del objeto:

CALL SUPTHA (NCOD)

Sentencia IMPRIMIR

Segun se ha visto en el punto V 2 1 el formato es:

IMPRIMIR ([expresión | literal | ∅ | @ TOTREG | $ ABSTRACTO])

Este formato será comprobado por el analizador y se generará:

a) Caso expresión:

*a*1) La característica primera es numérica, contador o número en cuyo caso se generará:

N1 = expresión

WRITE (6.10000) N1

*a*2) Si la primera característica es de tipo alfabético no debe estar relacionada con nada más y se generará una llamada a una subrutina que rellene un buffer CARACT con el valor de la característica alfabética que está en DC y una orden de impresión de CARACT:

CALL RELL (NCOD, CARACT)

WRITE (6,10000) CARACT

b) Caso literal

Se generará una orden de imprimir con un formato que contenga el literal:

WRITE (6, generar (INF))

generar INF Format (10x, 'literal')

siendo INF un contador de etiqueta que cada vez que se utiliza se incrementa en uno y en principio se inicializa con 1 001

c) Caso ∅

Se imprimirá la descripción del objeto y, por tanto, generaremos:

WRITE (6,10000) DESCP

d) Caso @ TOTREG

Se imprimirá la descripición del objeto y sus características numéricas y alfabéticas.

Se generará:

```
WRITE (6,10000) DESCP
TP=29          Ponter a la tabla primaria
D=0            Posición de una caracte-
                 ristic numérica dentro
                 del objeto
```

generar ET CALL BCTDS (TP, D, NCD, E, CR).

Esta subrutina deberá buscar en la dirección indicada en la entrada de la tabla primaria el elemento siguiente, hacer NCD igual a su número de código; con él acceder a la tabla de símbolos y hacer E igual 0, 1, 2 si es numérica, alfabética o palabra clave respectivamente y hacer CR igual al nombre de la característica.

Si la característica es de tipo numérico hacer D igual a la posición del valor de la característica dentro de OBJET.

Cuando TP devuelva cero no hay más características en el objeto.

```
        IF (TP.EQ.0) GOTO generar ET+1
        IF (E.EQ.1) GOTO generar ET+2
        IF (E.EQ.2) GOTO generar ET+3
        WRITE (6,10000) CR, ∅BJET(D)

generar ET+3 GOTO generar ET
generar ET+2 WRITE (6,10000) CR
             CALL RELL (NCD, CARACT)
             WRITE (6,10000) CARACT
             GOTO generar ET
generar ET+1
incrementar ET
```

Compilación de la sentencia CREAR.

Su formato es CREAR ((CONTADOR, EXPRES) [(CONTADOR, EXPRES) .)].

Su función es inicializar contadores

El formato de la instrucción es comprobado por el analizador y por cada par (contador, expres) se generará:

Contador = expres.

(W (número) = expres).

Compilación de la sentencia BIFURCAR A.

Su formato es BIFURCAR A número.

El formato es comprobado por el analizador y generará:

La aparición de BIFURCAR A hace generar un GOTO, y la aparición del número (etiqueta) desencadena las siguientes acciones:

— Meter la etiqueta en un campo NUM justificado a la derecha rellenando lo sobrante a la izquierda con ceros.

— Hacer una búsqueda secuencial en la tabla de etiqueta por si ha sido ya declarada o referenciada con otra instrucción *bifurcar a*. Si el indicador de etiquetas está a 2, es decir, que esa etiqueta ha sido ya declarada, como ahora se referencia ponemos el indicador de etiquetas a 3 (declarada y referenciada).

— Si el indicador de etiquetas está a 1 ha sido referenciada, como ahora se referencia otra vez sigue a 1.

— En ambos casos se suma 90.000 al número de etiqueta de usuario y se generará ese número calculado.

Al final del programa han de estar todos los indicadores de etiquetas a 3 (todas las etiquetas declaradas y referenciadas).

Sentencia SI.

Su formato es:

SI condición, ENTONCES instrucción [, instrucción, . . , instrucción], SINO instrucción [, instrucción, . . , instrucción], FINSI,
siendo instrucción cualquiera de las anteriores e incluso el propio SI.

Este formato es comprobado por el analizador y se generará cada vez que se encuentre SI condición:

```
IF ( NOT (condición))
```

Se suma uno a un contador de etiquetas C y se lleva a una pila TOP:

— Cuando encuentra el ENTONCES genera:

```
GOTO generar cabeza de TOP
```

Cada instrucción generará lo correspondiente según estamos viendo, dependiendo de la instrucción que sea.

— Cuando encuentra SINO, suma uno al contador de etiquetas C y se genera:

```
GOTO generar C
```

generar cabeza de TOP y eliminar la cabeza, llevar C a la pila TOP.

— Cuando encuentra el FINSI, generar cabeza de TOP y decapitar.

7)Storage of the intermediate code in disk.

8)Output the starting and ending addresse of the core segment taken.

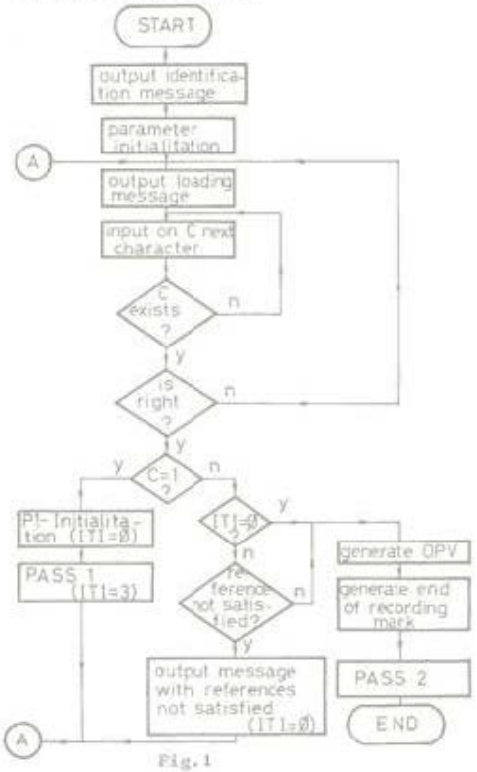


Fig. 1

Information blocks processing.

We have three differents kinds of information blocks and therefore we have three different processing segments.

-External information block processing.The

2)External symbol processing.

3)Building of an auxiliar vector MAX with each external identifier preceded by its length.

-Track information block processing.

1)Input information checksum(counting of the number of characters)

2)Hexadecimal digit pairs processing: this numbers are pointers to the auxiliar vector MAX, the external symbols are obtained and with these symbols as keys a table-look-up

-Entry points information block processing

1)Input information checksum

2)Processing of each pair:(identifier,offset) computing of the absolute address of the symbol(relative address plus offset).

3)Symbol processing (counting of the number of characters)

4)Symbol table look-up (the symbol is entered into the symbol table and an error message is producced if the symbol was already in the table.

5)Building of the memory map.

The first entry point of the first segment will be taken as the transfer address to begin the execution.

The loader starts at the address 1500 H and occupies about 4K of storage.

The address 10 H is the starting address, and following this address we have the vectors,tables... until address 1500 H.

Storage configuration in PASS-1 execution(fig2)

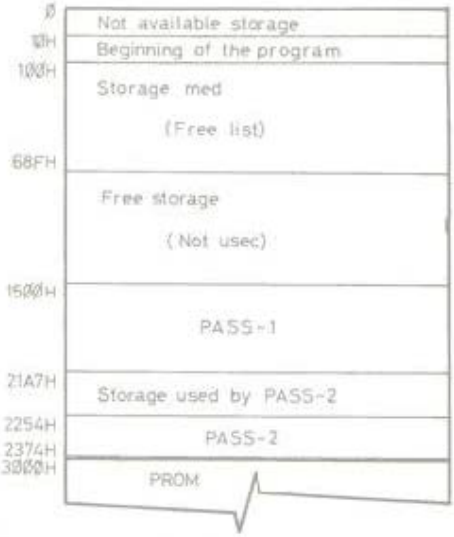


fig. 2

The low order addresses are used as variables by the monitor, FDOS. The first free location is at address 20 H.

The free list is used for variables represen_ tation,buffers and tables of the loader.

The principal objets in the free list are:
- output messages vector
- input buffer
- external auxiliar vector:MAX
- track information vector:OP
- vector with the second order relocatable addresses values (OPV)
- two tables with absolute blocks starting

PARTE II

BILBAO (1975-1983)

PRÓLOGO DE JOSÉ RAMÓN GARIAGOITIA

BILBAO (1975-1983)

Parte II

Conocí al profesor Isidro Ramos en el año 1974. Venía a ocupar una plaza de Profesor Agregado de Informática en la Universidad de Bilbao, en la actualidad Universidad del País Vasco. Entonces, la informática española era tan reciente que muchas cosas carecían de nombre y, a diferencia de lo sucedido en Macondo, ni siquiera teníamos el recurso de señalarlas con el dedo. No exagero: sólo dos años antes se habían aprobado las tres primeras facultades de informática. Isidro estaba prácticamente recién llegado de la Universidad de París VI y traía bajo el brazo los apuntes de un curso de estructuras de datos del profesor Pair, en el que se introducían una serie de operadores para la especificación abstracta de las estructuras; esto no era sino una primera aproximación a los tipos abstractos de datos, pero claro, como el nombre no existía, nadie lo llamó así.

Nuestra promoción fue la primera a la que impartió docencia; apenas habíamos hecho un curso de Fortran y habíamos programado algunos algoritmos relacionados con el cálculo numérico, así que Isidro nos introdujo en la informática «de verdad»: metodología de la programación y estructuras de datos, para continuar con la teoría de autómatas y lenguajes formales y acabar con un curso de compiladores, que culminó con la construcción de un macroensamblador cruzado para el microprocesador Intel 8080.

El Departamento de Automática e Informática había adquirido en 1974 un sistema de desarrollo de software para microprocesadores 8080 de Intel, el Intellec 8/80, pero venía en barco, debía pasar por la aduana y no llegaría hasta el verano de 1975. Para entonces, mi promoción había terminado sus estudios y Isidro nos invitó a formar parte de un grupo que, en principio, trabajaría alrededor de los microprocesadores y el nuevo sistema de desarrollo. El embrión del grupo se formó con dos personas, Josu y yo mismo (en clase sólo éramos 14).

En el curso 1995-96, Isidro mantuvo un seminario sobre programación concurrente que marcó nuestro trabajo para siempre. Hicimos un sistema multimicroprocesador con memoria compartida y un cargador reubicable para el Intellec 8/80, dando lugar a varias publicacio-

nes. Una de ellas, en la revista Euromicro Newsletter de abril del año 1977, puede ser uno de los primeros trabajos con microprocesadores españoles publicados en algún medio internacional. Se da la circunstancia de que en la patente 6219830 otorgada a Apple el 17/4/2001 aparece esta referencia, así que es posible que algún aparato de Apple incorpore alguna idea desarrollada por nuestro grupo.

El curso siguiente ya había media docena de personas más trabajando en el grupo. Fue un año muy intenso. Por un lado, Isidro consiguió un proyecto con una empresa vizcaína para la realización de un registrador cronológico en base a un sistema microprocesador. Este proyecto abrió la puerta a proyectos con esta y otras empresas que continuaron hasta la década de los años 90. Por otro lado, Isidro consiguió poner en marcha el Centro de Informática de la Universidad en Leioa del que fue su primer director. La infraestructura informática giraba en torno al terminal Univac DCT-2000 que se mantenía conectado al ordenador Univac 1108 del Centro de Cálculo y de Proceso de Datos del Ministerio de Educación. Todo esto funcionaba con tarjetas perforadas, así que las lectoras y perforadoras de tarjetas las consiguió en un «outlet» y los miembros del grupo aprendimos que la informática podía pesar mucho.

Esto son algunos de mis recuerdos de aquellos tiempos en los que comenzó mi relación con Isidro y, por lo tanto, son sólo una versión incompleta de estos, es más, dado el tiempo transcurrido, es posible que todo sea fruto de mi imaginación, por lo que nadie más será responsable de lo aquí escrito. En todo caso, es el comienzo de una gran historia sobre el paso de Isidro por Bilbao, una historia de trabajo, esfuerzo y dedicación.

La estancia de Isidro en Bilbao cambió la visión de la informática de varias generaciones y, directa o indirectamente a través de éstas, introdujo una nueva forma de hacer y sentir la informática en su entorno académico e industrial. Muchos miembros del grupo de trabajo han resuelto muy satisfactoriamente sus vidas profesionales gracias al impulso recibido desde aquel grupo. En mi caso, Isidro me atrajo a la informática y marcó mi trayectoria profesional con sus enseñanzas y su forma de hacer. Estoy convencido de que hay algo de él detrás de cada uno de mis logros, por lo que le estoy afectuosamente eternamente agradecido.

José Ramón Garitagoitia Padrones
25 de junio de 2014

"SYSTEME POUR LA CREATION ET L'EXPLOITATION DES BANQUES DE DONNEES"

I. RAMOS (Université de Bilbao (Espagne))

ABSTRACT: "On a développé un système pour la création et la gestion de banques de données qui peut être classé comme software de traitement de fichiers en batch processing orienté vers la gestion. On a definit un langage pour la création des banques (DDL) qui est interpreté et un langage pour l'interrogation et la modification des banques qui est précompilé. On pose le problème du temps de récupération des objets (entités) en introduisant à cet effet une phase de préadressage en même temps qu'on fait l'analyse lexicographique du texte et qui définit le sousensemble de la banque qui intéresse l'usager.

I INTRODUCTION

On considère dans ce travail qu'une banque de données est un ensemble d'objets dont la nature importe peu ici et telle que chaque objet est susceptible d'avoir un certain nombre de charactéristiques (à valeur: numérique, chaîne de characters , texte) designées par des identificateurs et en nombre fini à un instant donné, qui peuvent varier par la valeur designée ainsi que pár son nombre.

On a consideré aussi que le problème posé avec les banques de données est (aux instants de traduction et execution prés) le même que le problème d'implementation d'un langage de programmation où il faut:

- Représenter les objets du langage à compiler et qui sont declarés en utilissant des "instructions declaratives"(DDL)

- Représenter les traitements qu'on veut faire subir à ces objets et qui sont exprimés en utilisant des "instructions éxécutives" (Langage de commande et langage fournit au programmeur d'applications).

C'est-à-dire, d'une façon schématique il faudra fournir à l'usager:

1- Un langage pour la déclaration de la banque qui soit proche aux langages de programmation usuells et désigné de façon à permettre une déclaration "intuitive" de la banque. Comme pour nous une banque est un ensemble d'objets, une facon de définir un ensemble est l'ennumeration et c'est la solution choisie.

2- Un interpréte: les programmes écrits en utilisant le DDL doivent être interpretés et on obtient comme résultat le schèma associé à la banque de données ainsi que les dictionnaires (Tables de symbols) et un tableau booléen nécesaires pour les traitements suivants. De cette facon est réprésentée la banque de données sur un support secondaire (dans notre cas le tambour magnétique FASTRAND UNIVAC-1100)(nous avons le rôle de l'administrateur et le gestionnaire dans cette création).

Ramos, I. Système pour la création et gestion des banques de données. Seminaires Institut de Programmation (Université de Paris VI). 1974.

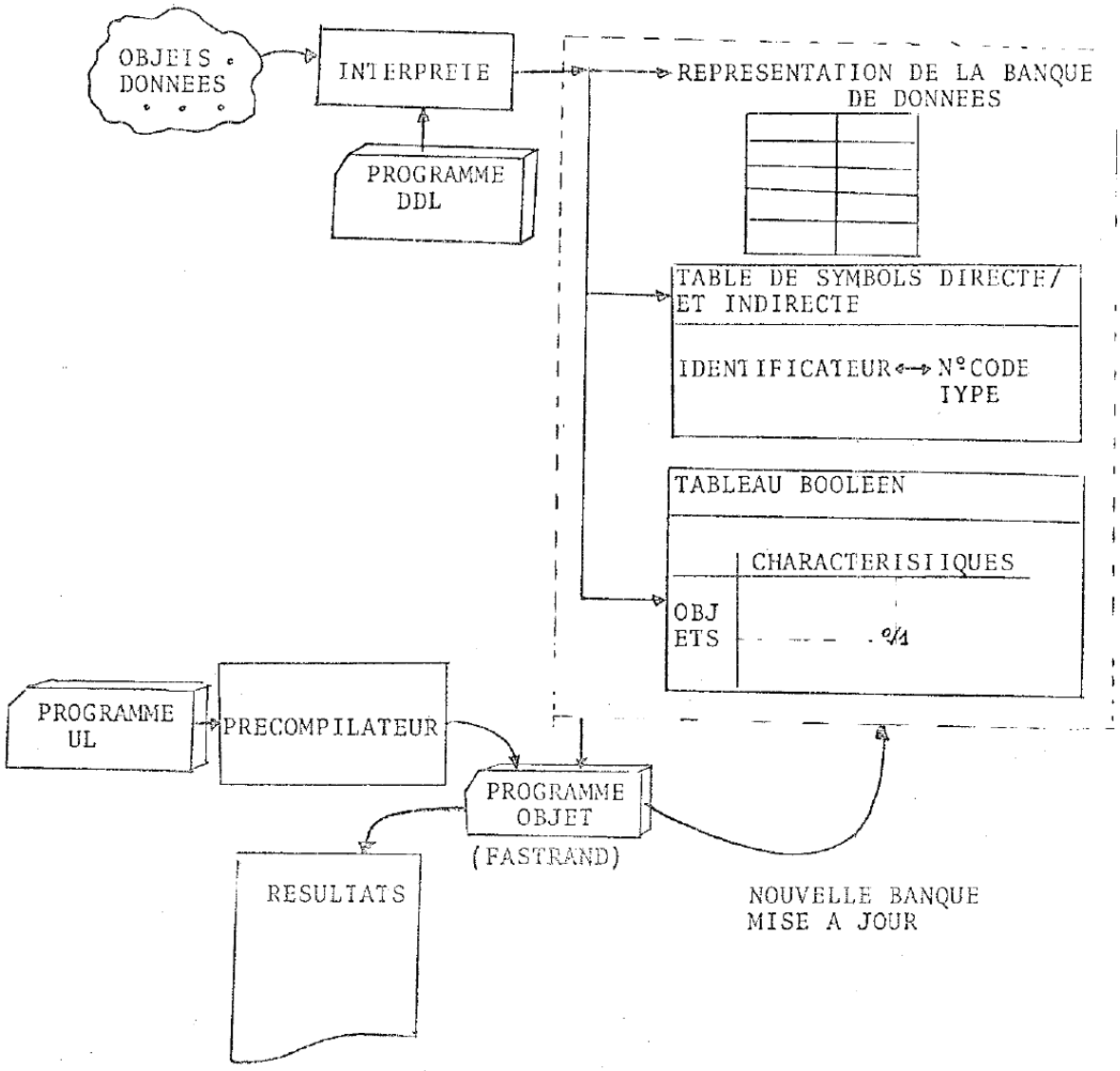
3-Langage de traitement (UL): pour l'utilisation de cette banque il faudra fournir au démandeur et l'utilisateur un langage qui exprime les traitements à effectuer, c'est-à-dire:

mise à jour: modifier pour certains objets les valeur de certaines charactéristiques, ajouter ou supprimer certaines charactéristiques....

interrogation: écrire, éditer... les valeurs de certaines charactéristiques de certains objets...

et tout cela en fonction des relations introduites sur les valeurs de cettes charactéristiques: on a besoin d'une instruction conditionelle: si... alors... sinon... fsi on introduit le aller à...

4-Traducteur des programmes écrits dans ce langage (langage hôte plus extensions) au langage objet. Nous avons choisi la solution précompilateur en utilisant comme langage hôte et objet le FORTRAN V (UNIVAC SERIE 1100). Nous avons alors le schéma suivant



II LANGAGE POUR LA CREATION DES BANQUES ET INTERPRETE ASSOCIE

Etant donné qu'on a choisi l'ennumération comme méthode pour décrire l'ensemble d'objets qui constitue la banque, un programme sera une description sequentielle de tous les objets. On pose un exemple:

FICHERO ' UNIVERSIDAD AUTONOMA DE BILBAO, BIBLIOTECA'

{ ACTUALIZACION
  CREACION TALLA OBJETOS n INFORMACION OBJETOS m }
[CARACTERISTICAS { ALFABETICAS | NUMERICAS | PALABRAS CLAVE } identificateur characteristique
  [, identificateur de characteristique [,...]]]
  [; { ALFABETICAS | NUMERICAS | PALABRAS CLAVE } identi...] ...

OBJETO P ' Informations sur l'objet ' [ABSIRACTO ' text '] .

{ identificateur charactéristique = [identificateur charac
teristique = [... =]] { valeur | "literal" } ; ...
mot clé ; [mot clé ;] ... }*

OBJETO Q ...

.
.
.

FIN

NOTE . Les metaparenthèses: [ ] indiquent que le contenu peut manquer
{ } " qu'il faut choisir une des alternatives

. Le metasymbol ... indique la repétition de ce qui precède un numèro indeterminé de fois et aussi etc.

. Les terminaux ont été soulignés

Aprés le terminal FICHERO il y a l'information sur la banque: nom... (on pourrait ajouter aussi une certaine protection: clé-verrou...), suivie de ACTUALIZACION (si on veut actualiser une banque déja existente) ou CREACION si on veut créer une nouvelle banque, dans ce dernier cas il faut especifier la taille maximalle prevue pour chaque objet (TALLA OBJETOS n) et pour l'information sur l'objet (INFORMACION OBJETOS m).

On déclare à continuation tous les identificateurs des characteristi

ques et son type (CARACTERISTICAS {ALFABETICAS ...} ...)

Aprés on décrit les objets: (OBJET P) ... et puis une certaine information sur chaque objet et un possible abstract (ABSTRACTO ...) et on définit les charactéristiques qui sont présentes et éventuellement sa valeur. Les mots clés qui peuvent être utilisés pour "récupérer" l'objet sont aussi définis. Le tout se termine par FIN, qui marque la fin du programme de création.

Ce langage à été defini à l'aide d'une gramaire regulière et les programmes qu'on écrit sont interprétés par l'automate d'états finis associé, qui crée la banque ainsi que les tables de symbols... (program conduit par les données) sur le tambour magnetique d'une façon sequentielle.

La table de symbols directe (identificateur de charactéristique → type, numéro de code) est representée par une table d'adressage dispersé avec chaînes de synonimes formant une liste chainnée, tandis que la table de symbols indirecte (Nº code → identificateur de charactéristique) permet une répresentation par une table d'aces direct.

Les objets à son tour (qui ont une taille variable) sont représentés sur les n mots par une autre table d'accés dispersé avec listes de synonimes chaînées. Chaque élément de la liste est un tripléte (Nº de code, "valeur", suivant) et la répresentation de la valeur dépend du type de la characteristique. Le rest des n mots est organisé en deux espaces libres chainés de tailles 3 et 4 respectivément. Il y a aussi une zône de débordement à la fin.

III LANGAGE POUR L'INTERROGATION ET LA MODIFICATION DES BANQUES ET COMPILATEUR ASSOCIE

Ce langage est à la fois le langage de commande de la banque et le langage fournit au programmeur d'applications. Il n'existent pas des declarations puisque tous les objets sont declarés avec le DDL.

- Les objets fournis par le langage d'interrogation sont:
  - Objets FORTRAN: constants, variables...: ils sont transliterés du langage source au langage objet
  - Compteurs: sont des variables fournies au programmeur pour être utilisées comme conpteurs ... ils sont substitués par un élément d'un tableau réel.
  - Charactéristiques: sont des identificateurs de charactéristiques (aspect syntaxique) qui sont substitués par un élément de tableau: OBJET (K) où K est un index calculé à l'execution et tel que sa valeur dépend de l'objet traité à chaque instant (par des subroutines ajoutées au programme objet).
- D'un autre coté les actions qu'on veut réalisser sur les objets qui constituent la banque sont:
  - Consulter
  - Modifier
  - Ajouter
  - Eliminer.....

Certaines characteristiques ou tout l'objet.

Cettes actions sont realisées par des sousprogrammes de biblio thèque dont on disposse:

AJOUTER: Une ou plusieures charactéristiques en définissant sa valeur.

MODIFIER: Une ou plusieures charactéristiques en définissant sa nouvelle valeur.

ELIMINER: Une ou plusieures charactéristiques.

CREER: Un ou plusieurs compteurs.

ECRIRE: Une ou plusieures charactéristiques ou tout l'objet..

Il éxiste aussi l'instruction conditionnelle: si ε alors I_1 sinon I_2 ; ou ε est une expression booléene et I_1, I_2 sont des series d'instruc tions (par exemple des autres si ... alors....) ainsi que le aller à.

L'expression booléene est définie par des opérateurs de rélation traditionnels (>,≥ , =, ...) et objets du langage source et les opérations logiques clasiques: ET, OU, NON et aussi pour la présen ce ou absence de certains mots clés dans un certain ordre dans l'objet.... (ABSTRACT).

Par example on peut exprimer:

SI PRECIO<50 y no (PRECIO<100) y AUTOR = "DAVID GRIES"
y COMPILER CONSTRUCTION ENTONCES...SINO... FSI

Le précompilateur pour ce langage (qu'on définit par une grammai re LL(1)) est basé sur une analyse syntaxique descendente déterministe et réalise un seul passage sur le texte source ou successivement (au-fur-et-à-mésure de sa lecture) sont realisées les fonc tions:

- lecture et édition du texte source
- analyse morphologique
- analyse léxicographique plus génération d'un vecteur entier qui caractérise le programme selon les charac téristiques présentes ou absentes.
- génération de texte objet.

Le programme objet est tel qu'il s'applique successivement à tous les objets qui peuvent intéréser l'usager de la banque.

Pour définir si un objet peut intéresser ou non au programme de l'usager ont réalise à l'éxécution du programme objet une fonction de préadressage basée sur le tableau booléene construit pendant la création de la banque (et modifié par les mis-à-jours) et le vecteur entier qui caractérise le programme (caractéristiques présentes ou absentes). L'objet K intéresse si la variable booléene B (a valeur initialle FAUX) est VRAI aprés le procés.

pour i = 1 pas 1 jusqua N CARACT faire B: = B +BV[i].MB[K, i];

c'est-à-dire- si dans l'objet il y a au moins une characteristique des charactéristiques présentes dans le programme de l'usager, ce qui est assez brutal (il y a très peu d'objets exclus). Nous pensons qu'on pourrait caractériser le programme par une expression booléene entre les charactéristiques Ci, par example;

$$C1.\overline{C2}.C5..... + C6.C7....$$

avec . = ET

\+ = OU...

et définir une fonction de préadressage plus performante, ce qui n'a pas été fait pour l'instant.

IV CONCLUSION

Le travail a été realisé par un equipe formé de deux élèves de L' INSTITUTO DE INFORMATICA DE MADRID à mi-temps et moi même un jour par semaine en 5 mois sur un UNIVAC 1108 et il a obtenu le PRIX NATIONAL D'INFORMATIQUE CCS 1974. On à realisé une programmation structuré de l'interprète et le compilateur, mais on à trouvé tres tôt les difficultés posées par la manque d'idoneité du langage NU-ALGOL (en particulier l'identification de procédures et modules disparait vite puisque le nombre maximall de procédures d'un programme est limité par l'implementation).

On à construit un systeme pour la gestion des banques de données où la proximité et adecuation des langages implementés aux problèmes traités sont , nous croyons, assez efficacesbien que l'absence d'informations partagées entre diférents objets et d'un mécanisme de protection qui serait trés facillement d'ailleurs comme on indique dans le travail), le fait qu'on ne peut pas définir un objet en fonction des autres ainsi que on ne dispose pas de messures de performance, conseillent une continuation du travail.

V BIBLIOGRAPHIE

1. PAIR, C., "Sujet d'examen"

2. PAIR, C., "Les structures de l'information" Ales 1971

3.º CONGRESO NACIONAL
INFORMATICA Y AUTOMATICA
MADRID - 29/31 OCTUBRE 1975

SESION
v. 31 - B1/08

"METAPROGRAMACION ESTRUCTURADA"

por

I. RAMOS, UNIVERSIDAD DE BILBAO

R. MOYA, UNIVAC

RESUMEN

"Se presenta un sistema de metaprogramación estructurada basado en la Teoría de las Ramificaciones y la Teoría Algebraica de lenguajes, obteniéndose un generador de programas general. Se hace un estudio de los problemas de determinismo y finitud de las ejecuciones (cálculos) de los programas estructurados, que se generan".

Moya R., Ramos I. Metaprogramación Estructurada. 3er Congreso Nacional de Informática y Automática. 1975.

1. INTRODUCCION

1.1 Introducción

La programación estructurada conoce en estos momentos un desarrollo espectacular tanto en los aspectos teóricos que la conciernen (1) como en las aplicaciones que la utilizan (2). No obstante, y como veremos en este trabajo, no aporta nada nuevo desde el punto de vista informático, siendo quizás su principal virtud la posibilidad de "industrializar" una labor tan creativa como es la programación (legibilidad y facil documentación de los programas, control del rendimiento del programador e independencia con relación a las peculiaridades ("trucos de programación") de éste).

Nuestro trabajo trata de metaprogramación estructurada, haciendo uso de la Teoría de las Ramificaciones (3) y la Teoría Algebraica de Lenguajes, nos replanteamos el concepto de programa estructurado y proponemos un sistema de programación en el que dada la descripción del programa (aspecto estático), mediante un metaprograma general somos capaces de generar todas sus ejecuciones posibles (cálculos, aspecto dinámico del programa). Nos planteamos además los problemas de determinismo y finitud de los programas que son ejecutados y damos ejemplos de uso del sistema.

1.2 Ramificaciones y gramáticas

Ramificaciones: El "árbol programático" es la representación gráfica normalmente utilizada para expresar un programa estructurado (de forma análoga a como lo es el diagrama de flujo para un programa no estructurado). Pese a que nuestro concepto de árbol programático difiere del normalmente utilizado en la bibliografía, nos vamos a ver obligados también a tratar arborescencias (en el sentido de la teoría de grafos) con una o varias raices, con un cierto orden de izquierda a derecha tales que cada nodo está etiquetado por un símbolo de un cierto alfabeto V: es lo que vamos a llamar una ramificación sobre V, notaremos su conjunto como $\hat{V}$.

Para poder algebrizar el estudio de las ramificaciones definamos la estructura binoide libre como el cuadruplete

$$(\hat{V}, \oplus, x, \Lambda)$$

- donde $\hat{V}$ es el conjunto de ramificaciones sobre el alfabeto V.

- $\oplus$ es una ley de composición interna sobre el conjunto $\hat{V}$ (de las ramificaciones) tal que:

$$\forall\, r_1, r_2 \in \hat{V} \quad r_1 \oplus r_2 \in \hat{V}$$

la llamaremos concatenación de ramificaciones y formaliza el concepto intuitivo de yuxtaposición de ramificaciones: si r_1 es una ramificación de m raices y r_2 una de n raices, $r_1 \oplus r_2$ es la ramificación de m+n raices resultantes de concatenar a la derecha de r_1, r_2.

Esta L.C.I. tiene la propiedad asociativa y además un elemento neutro que notaremos Λ (ramificación vacía), tal que:

$\forall r \in \hat{V}$ $r \oplus \Lambda = \Lambda \oplus r = r$

- x es una ley de composición externa tal que:

$\forall a \in V, \forall r \in \hat{V}$ asocia: $a \times r \in \hat{V}$

la llamaremos enraizamiento puesto que a un elemento $a \in V$ y una ramificación $r \in \hat{V}$ asocia la ramificación notada $a \times r$ cuya raíz es a.

Identificaremos los elementos de V con las ramificaciones formadas por un solo elemento:

$\forall a \in V$, a se identifica con $a \times \Lambda$, a efectos de notación, y es en este sentido que calificamos de libre al binoide definido.

- Λ como hemos dicho es la ramificación vacía.

Visto el concepto de ramificación pasemos a recordar los conceptos necesarios sobre las C-gramáticas (o Tipo 2 en la jerarquía de Chomsky).

C-Gramática: Una C-Gramática es el cuadruplete:

$$(N, T, ::=, X)$$

donde:

- T y N son dos conjuntos finitos llamados respectivamente alfabeto terminal y alfabeto no terminal; los elementos de T los llamaremos terminales y los de N no terminales.

Llamamos $V = T \cup N$ alfabeto de la gramática.

- X, es un elemento de N que llamamos axioma.

- ::= es una relación binaria llamada relación de producción entre N y V^* (conjunto de palabras sobre V).

Un par $A \in N$ y $\alpha \in V^*$ tal que: $A ::= \alpha$ forma una regla(cuyo número suponemos finito).

Ramificaciones engendradas por una C-Gramática: A toda regla de una C-gramática asociamos una ramificación en la forma siguiente:

$A ::= \alpha$ tiene como ramificación asociada r, que es aquella en la que A es la raiz de r y α es la palabra de las hojas de r.

Para definir en forma precisa los conceptos de raiz y palabra de las hojas recurrimos al siguiente teorema:

Teorema: Toda ramificación no vacía, sobre V se escribe de forma única en la forma:

$$r = A \times s * t; \quad A \in V$$

$$s, t \in \hat{V}$$

lo que nos permite definir:

-raiz (c palabra de raices de r) la notaremos $\rho(r)$ y la describiremos recursivamente como:

- $\rho(A) = \varepsilon$ (ε palabra vacía sobre V)

- $\rho(A \times s*t) = A.\ \rho(t)$ (".":L.C.I. sobre V^*)

- palabra de las hojas la notaremos $\phi(r)$ y la definiremos recursivamente como:

- $\phi(A) = \varepsilon$

- $\phi(A \times s*t) =$ si $s = A$ entonces $A.\phi(t)$ si no $\phi(s).\ \phi(t)$;

Con estas definiciones, es facil especificar el lenguaje engendrado por una C-gramática: será el conjunto de palabras $\in T^*$ tales que forman el conjunto de palabras de las hojas de las ramificaciones r cuya $\rho(r) = X$;

o séa, definiendo la función $\phi_T(r)$:

- $\phi_T(A) = \varepsilon$

- $\phi_T(A \times s*t) =$ si $A \in N$ entonces $\phi_T(s).\phi_T(t)$ si no si $A \in T$ entonces $A\ \phi_T(t)$;

con ello

$$L_G = \{\alpha \in T^* \mid \alpha = \phi_T(r),\ y\ \rho(r) = X\}$$

Por ejemplo: dada la G:

T = {a,b}

N = {X}

::= X ::= a X b | a b

X = axioma

Las ramificaciones engendradas por G son:



o también X x (a ⊛ X ⊛ b), X x (a ⊛ b)

y $L_G = \{a^n\ b^n \mid n \geq 1\}$

1.3 Representación de ramificaciones en una memoria direccionable

Las ramificaciones vistas en el punto anterior, necesitaremos representarlas en memoria a efectos del sistema de programación que proponemos en el punto (3) de este trabajo.

Vamos a cambiar nuestro punto de vista algebraico y situarnos en el de Estructuras de la Información.

Una ramificación es una estructura de la información y por lo tanto vendrá definida por un triplete (4):

$$EI = (E, L, M)$$

- siendo E un conjunto de objetos

- L un conjunto de funciones de acceso

- M un conjunto de modificaciones

En el caso de las ramificaciones, el conjunto E estará formado por: V U P siendo:

- V conjunto de etiquetas de los nodos de la ramificación.

- P conjunto de nodos.

(no debemos confundir el conjunto de etiquetas V con el conjunto de nodos P, puesto que una etiqueta puede, por ejemplo, etiquetar distintos nodos).

El conjunto L (funciones de acceso estará formado por las siguientes funciones:

- $r_1, r_2, \ldots, r_p$ conjunto de raices que nos dan acceso a la ramificación (son p funciones de cero argumentos).

- $s_1, s_2, \ldots, s_q$ conjunto de funciones de sucesión tales que: $s_i : P \to P$, $i = 1,q$

- v: función de etiquetaje:

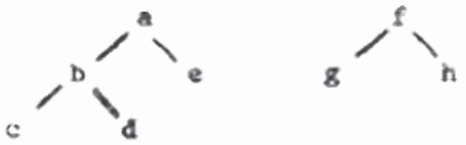
$$v: P \to V$$

Así pues: $L = (r_1, r_2, \ldots r_p, s_1, s_2, \ldots s_q, v)$

obviamente estas funciones de acceso pueden componerse dando lugar a elementos de un conjunto que llamaremos $\bar{L}$. Un elemento de este conjunto sería por ejemplo

$$v\ s_1^2\ r_1$$

y que en el caso de la ramificación que damos como ejemplo valdría c.



El conjunto de modificaciones M sería:

m_1 : cambio de valor de v en un nodo

m_2 : eliminación de un nodo

m_3 : adición de un nodo

m_4 : modificación de una función s_i en un nodo

Este conjunto de modificaciones se puede componer con el de las funciones de acceso dando lugar a todos los posibles accesos, modificaciones sobre la estructura de información que llamamos ramificación (pero teniendo en cuenta que cualquier función compuesta nos pasa siempre de una ramificación a otra ramificación).

El problema que nos planteamos en este punto es la representación en memoria de estas ramificaciones de forma que:

- ocupemos el mínimo de memoria

- las funciones de acceso y modificación no se vean severamente afectadas.

Una forma usual en Informática de realizar esta representación es pasar toda ramificación a una ramificación equivalente binaria (el semigrado exterior de todo nodo del grafo soporte es como máximo 2), definiendo dos funciones de sucesión que llamaremos enlace vertical (EV) y enlace horizontal (EH), tales que:

EV: P → P y EH: P → P definidas de la forma siguiente: La función EV nos dá para cada nodo su primogénito caso de que exista y nil caso de que no y la función EH nos dá para cada nodo su hermano siguiente y nil si éste no existe.

Estos nos permite reducir el número de funciones de acceso en la forma siguiente:

$$r_i = EH^{i-1} \cdot r_1 \quad i = 2,p$$

$$s_i = EH^{i-1} \cdot EV \quad i = 2,q$$

con lo que el conjunto L queda reducido a: $L = (r_1, EV, EH, v)$

Podemos además representar EV ó EH por contiguidad sobre un vector que llamamos NOM (nombres) y EH, en este caso la ramificación del ejemplo vendría dada por:

INDICE	NOM	LH
1	a	5
2	b	8
3	c	10
4	-1	0
5	f	0
6	g	12
7	-1	0
8	e	0
9	-1	0
10	d	0
11	-1	0
12	h	0
13	-1	0

En donde hemos representado la no definición de EV por un -1 añadido en la posición siguiente. EH viene dado explícitamente y v no es más que el contenido del vector NOM para cada índice. Nos queda por representar r_1, esto lo haremos sobre un vector que llamaremos COM (comienzo) que para cada ramificación

nos da el índice donde comienza. En nuestro ejemplo el vector COM estaría reducido a un solo elemento

COM (1) = 1

ya que representamos una sola ramificación y esta empieza en el índice 1.

Como las ramificación que vamos a tratar son las engendradas por una gramática, debemos pues codificar esta. Lo haremos en la forma siguiente:

T (alfabeto terminal) por los números naturales: 1,2,..., t = cardinal (T).

N (alfabeto no terminal) por los números naturales: t+1, t+2,...,n =cardinal (V).

::= (relación de producción) por las ramificaciones engendradas y estas en la forma expuesta anteriormente.

X: por t+1

Visto esto podemos ya abordar el punto 2.

2. CONCEPTO DE PROGRAMA ESTRUCTURADO

2.1 Aspecto dinámico y estático de un programa

Un programa presenta dos aspectos:

- Dinámico: Es lo que llamamos ejecuciones del programa (cálculos) que consiste en la ejecución secuencial (no consideramos en este caso el cálculo paralelo) de un conjunto de acciones (noción primera) especificadas mediante instrucciones (que nosotros llamaremos tratamientos terminales y notaremos T_i) Así pues una ejecución de un programa será una cadena de elementos T_i de un cierto alfabeto T (alfabeto de tratamientos terminales! pueden ser desde instrucciones máquina hasta programas completos) $T = (T_1, T_2,..,T_n)$ una ejecución será pues: $T_iT_j...T_k \in T^*$

- Estático: El aspecto estático de un programa (que es lo que normalmente se entiende como programa en Informática) especifica el conjunto de todas sus ejecuciones posibles y especificará por lo tanto un subconjunto de T^*.

Siguiendo las indicaciones de Dijkstra (5), en programación, es conveniente que los dos aspectos de un programa sean lo más próximos posibles (el ideal sería programas estrictamente lineales donde obviamente, no existen el go to, if...then...else...; ó while... do...;). Esto nos lleva directamente al punto siguiente.

2.2 Programa como gramática que especifica todas sus ejecuciones

Una forma natural de especificar subconjuntos de cadenas sobre un alfabeto (en este caso T) es mediante una gramática. Así pues llegamos a la definición siguiente: "Un programa (aspecto estático) es pues la gramática que especifica todas sus ejecuciones posibles (ya que estas son cadenas sobre el alfabeto T)".

En programación estructurada tenemos solamente las siguientes estructuras:

- Concatenación de tratamientos (lo que implica una secuencialidad en su ejecución). Según nuestro punto de vista se corresponde con la concatenación de elementos de T.

- Estructura: si... entonces ... si no...;en nuestro caso la estructura es más potente ya que se corresponde con un número arbitrario de partes derechas de una "regla de producción". En concreto el tratamiento

T: si B entonces I_1 si no I_2; se especificaría como:

$T::= I_1|1|\ I_2|2|$

Que es la notación usual para las gramáticas, donde además hemos numerado sus reglas mediante una función que a toda regla asocia un número natural. Mediante el número de regla especificamos el predicado que debe cumplirse para que se tome esa regla.

Así pues tendríamos:

Predicado 1: = B

Predicado 2: = $\overline{B}$

- Estructura: mientras B hacer I;

Esta estructura se especifica de forma recursiva:

T:: = I.T |1|

Predicado 1:= B;

Lo que se corresponde perfectamente con la estructura indicada ya que el test sobre B se hace antes de ejecutar I.

Un programa (aspecto estático) será pues una gramática.

Por ejemplo: Supongamos que queremos leer una serie de números enteros (positivos y negativos), sumar los positivos y almacenar la suma en la variable NP, sumar los negativos y almacenar la suma en la variable NN. El final de la serie se detecta porque el número leido es cero.

El programa (gramática) sería:

Programa::= Leer. Tratamiento |1|

Tratamiento::= Sumar-positivos. Programa |2|

Sumar-negativos. Programa |3|

Escribir |4|

Donde hemos subrayado los tratamientos terminales, su equivalencia en un lenguaje de programación sería:

Leer: READ (N);

Sumar-positivos: NP ← NP + N;

Sumar-negativos: NN ← NN + N;

Escribir: WRITE (NP,NN);

y los correspondientes predicados serían:

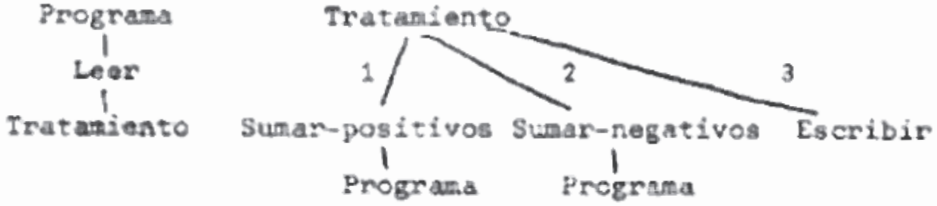
P_1: TRUE

P_2: N > 0

P_3: N < 0

P_4: N = 0

Con el programa tenemos asociado el árbol programático que en este caso sería la ramificación siguiente:



Toda ejecución del programa viene representada por, un camino que va desde la raiz (axioma) a una hoja. El problema es: ante una serie de alternativas ¿Qué subramificación elegir?. Esta pregunta solo puede resolverse en general, de forma dinámica (en la ejecución) y es el resultado de la ejecución de tratamientos anteriores. Para resolverla introducimos con cada predicado que define una alternativa, un tratamiento de valor booleano que nos resuelve el problema.

Tenemos pues los siguientes niveles de programación:

Nivel 0: METAPROGRAMACION: (Metaprograma general capaz de generar todas las frases (ejecuciones) de un programa (gramatica)) — METAPROGRAMA GENERAL

↑ — ↑

Nivel 1: PROGRAMACION (gramática que especifica todas las ejecuciones) — G = (T,N,::=X)

Descrito mediante una gramática ↑ — ↑

Nivel 2: EJECUCION DEL PROGRAMA: $T_1\ T_2\ \ldots\ T_n \in T^*$

El metaprograma general, dada una descripción del programa mediante una gramática, puede generar todas sus ejecuciones posibles, y es en este sentido que hablamos de metaprogramación estructurada.

2.3 La programación estructurada desde un punto de vista algebraico

Para nosotros pues, un programa estructurado es una gramática. Dispondremos de un alfabeto de tratamientos terminales $T = (T_1, T_2, \ldots T_T)$, un alfabeto de tratamientos no terminales $N = (N_1, N_2, \ldots N_N)$, cada tratamiento no terminal sirve para etiquetar (nombrar subcadenas de tratamientos que necesariamente se realizan en secuencia (aunque pueden nombrar distintas subcadenas según ciertos predicados), un conjunto finito de raglas de producción (::=) que da cuenta de la estructura del programa y un axioma que nos sirve para nombrar todas las cadenas (ejecuciones) del programa.

Especifiquemos que entendemos por tratamiento:

1. Todo $T_i \in T$ es un tratamiento

2. ϵ (tratamiento vacío) es un tratamiento

3. Si T_1 y T_2 son dos tratamientos, su concatenación también lo es

4. Si $T_1, T_2, \ldots T_p$ son p tratamientos

   $T_1 | T_2 | T_3 \ldots | T_p$ es un tratamiento

5. Si T_1 es un tratamiento, T_2 especificado como

   $T_2 ::= T_1\ T_2\ |1|$ también lo es

6. Todo $N_1 \in N$ (alfabeto no terminal) es un tratamiento. A efectos de representación:

- $\forall\ T_i \in T$: vendrá representado: por un segmento de instrucciones operativas (no de control) de un lenguaje de programación (que podrán ser macrosustituidas en el programa objeto)

- por un subprograma cerrado.

- $\forall\ N_i \in N$: Vendrá representado necesariamente por un subprograma cerrado.

3. METAPROGRAMACION ESTRUCTURADA

3.1 Concepto de Metaprograma

Para nosotros un metaprograma será un programa que acepta como dato la descripción de otro programa (mediante una gramática en nuestro caso) y es susceptible de generar todas las ejecuciones posibles del programa dato.

3.2 Metaprograma General

El Metaprograma general que hemos realizado, se encarga de recorrer la gramática que especifica el programa a ejecutar, partiendo del axioma y recorriendo la ramificación en forma descendente (escogiendo las alternativas según el valor de los predicados) hasta llegar a una hoja: es decir genera una ejecución del programa dato.

En este recorrido se encuentra con tres tipos de nodos (cuyo valor supondremos almacenado en la variable B).

Recordemos la codificación hecha de la gramática en 1.3.

-T ≥ B> 0 el nodo B representa un tratamiento terminal, lo ejecuta (EJEC(B)) y pasa al tratamiento siguiente.

-T < B > 0 el nodo B representa un tratamiento no terminal y por lo tanto va a recorrer su ramificación asociada (J:=COM(B)) pero guardando en una pila el índice J del tratamiento no terminal (lo que permitirá después de la ejecución de B, reanudar el recorrido en el nodo siguiente)

-B < 0 marca el fin de una regla, se da control al tratamiento que está en la cumbre de la pila +1 y se decapita ésta o se termina caso de que esté vacía.

NOTA: Al iniciar la ejecución de cualquier tratamiento no terminal se escoge la parte derecha cuyo predicado es verdad, dando un mensaje de error y terminando la ejecución si ninguno de ellos es verdad.

El metaprograma lo damos en ALGOL-60 (en este caso el del miniordenador HP-2116B) y las variables que se usan tienen el significado siguiente:

T = cardinal del conjunto de tratamientos terminales.

N = cardinal del conjunto de tratamientos no terminales.

G = talla de los vectores necesarios para representar la gramática.

NOM = vector para la representación de la ramificación (ref.1.3).

LH = vector para la representación de la ramificación (ref.1.3).

COM = vector para la representación de la ramificación (ref. 1.3)

PILA: pila necesaria para el tratamiento.

H: variable apuntador de gestión de la pila

J: " " para recorrer la ramificación

B: " cuyo contenido es el valor de los nodos.

R: " " " " el número de regla.

Usamos además dos prodedimientos:

- CB(R) (condición booleana) que toma el valor del predicado asociado con la regla R.

- EJEC(B) que ejecuta el texto del programa asociado con el tratamiento terminal B.

El Programa es el siguiente:

```
begin
comment Declaración de los objetos (variables,...) usados por el programa dato;
boolean procedure CB(R); value R; integer R;
begin
case R
begin
comment Asignación a CB de los distintos (R) predicados;
end;
end;
procedure EJEC(B); value B; integer B;
begin
case B
begin
comment textos de los distintos (B) tratamientos terminales;
end;
end;
integer T, N, G;
label RE, L, FIN;
read  (T, N, G);
begin integer array NOM, LH (1:G), COM (T+1:N),PILA (1:MAX);
      integer H, J, B, R;
      H:= 0;
      read (NOM, LH, COM);
  J:= COM (T+1);
L:R:=NOM (J);
  J:= J+1
  if not CB(R) then if LH(J) ≠ 0 then begin J:=LH(J);
                                            go to L;
                                       end
```

```
                                else begin write ("error"),
                                           go to FIN;
                                     end;

RE: B: = NOM (J);
    if: B>0 then begin if B>T then begin H:= H+1;
                                        PILA(H):= J;
                                        J:=COM(B);
                                        go to L
                                   end;
                               else begin EJEC(B);
                                        J:=J+1;
                                        go to RE
                                   end;
                end;
    if H≠0 then begin J:=PILA(H)+1;H:=H-1;go to RE end;
PIN:;
end
```

Tal y como ha sido concebido este metaprograma el número de regla aparece como primer elemento de cada parte izquierda.

En el metaprograma, los comentarios que aparecen:

NOTA: Los corchetes han sido sustituidos por paréntesis.

- Declaración de los objetos del programa dato.

- Cuerpo del procedimiento CB

- Cuerpo del procedimiento EJEC

Dependen de cada programa dato, así como la lectura de: T,N,G,NOM,LH,COM. En el vector PILA se prevee una talla máxima que nosotros hemos llamado MAX.

3.3 Sistema de Programación

3.3.1 Lenguaje de Programación

Tal y como hemos pensado la programación estructurada, podríamos dar un lenguaje de programación basado en estas ideas y en el que todo programa tendría la estructura siguiente (se da el programa del ejemplo de 2.2)

```
*LENGUAJE OBJETO ALGOL-60;

*TRATAMIENTOS TERMINALES LEER (ABIERTO), SUMAR-NEGATIVOS (ABIERTO), SUMAR-POSITIVOS (ABIERTO), ESCRIBIR (ABIERTO);

*TRATAMIENTOS NO TERMINALES (PROGRAMA, CONT);

*DECLARACION DE OBJETOS INTEGER NN:=0,NP:=0, NO;

*ESTRUCTURA DEL PROGRAMA

    PROGRAMA: LEER CONT |1|

        CONT: SUMAR-NEGATIVOS PROGRAMA |2|

              SUMAR-POSITIVOS PROGRAMA |3|

              ESCRIBIR |4|

*CONDICIONES

        1: TRUE;

        2: NO < 0;

        3: NO > 0;

        4: NO = 0;

*TEXTO TRATAMIENTOS TERMINALES

        LEER: READ (1,#(I3),NO);

 SUMAR-NEGATIVOS: NN:=NN+NO;

 SUMAR-POSITIVOS: NP:=NP+NO

 ESCRIBIR: WRITE(2,#("NN",I5,3x,"NP,I5),NN,NP);

*END
```

3.3.2 Traductor

Damos solo el esquema de las acciones del traductor al encontrar cada una de las frases del programa:

*LENGUAJE OBJETO ALGOL-60;

Podemos tener otros lenguajes objeto: FORTRAN, COBOL.....

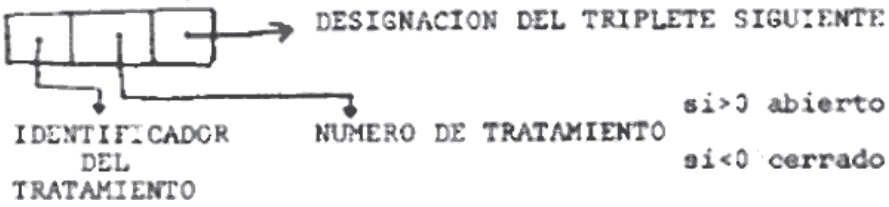
- Análisis de la corrección sintáctica de la frase.

- Generación del metaprograma general en ese lenguaje (lo que no presenta mayor dificultad ya que éste es un programa iterativo (gracias al uso de la PILA)).

*TRATAMIENTOS TERMINALES....;

- Análisis de la corrección sintáctica de la frase.

- Análisis lexicográfico de la lista de tratamientos y numeración secuencial de estos (lo que nos proporciona T), al mismo tiempo los almacenamos en una tabla de símbolos representada mediante un direccionamiento disperso ligado tal que cada nodo es un triplete:



*TRATAMIENTOS NO TERMINALES...;

id. al punto anterior. Esto nos proporciona N y además conocemos la talla del vector COM que es N-T ya que el rango de variación del subíndice es (T+1:N).

*OBJETOS

Las declaraciones de los objetos se copian literalmente en el programa objeto (ya que se especifican directamente en el lenguaje objeto) que es el metaprograma y en el lugar indicado por el primer comentario (ref. 3.2)

*ESTRUCTURA DEL PROGRAMA

- Análisis sintáctico de las frases

- Construcción de los vectores: NOM, LH y COM (que en el punto 3.2 necesitan ser leidos)

*CONDICIONES

-Análisis sintáctico de las frases

-Copiar literalmente cada una de las condiciones booleanas por orden y en la forma:

CB:= condición booleana;

forman el cuerpo del procedimiento CB(R) (segundo comentario del metaprograma).

*TEXTO TRATAMIENTOS TERMINALES

Id. al punto anterior. Forman el cuerpo del procedimiento EJEC(B)(tercer comentario del metaprograma)

El traductor es pues simple y puede actuar en una sola pasada.

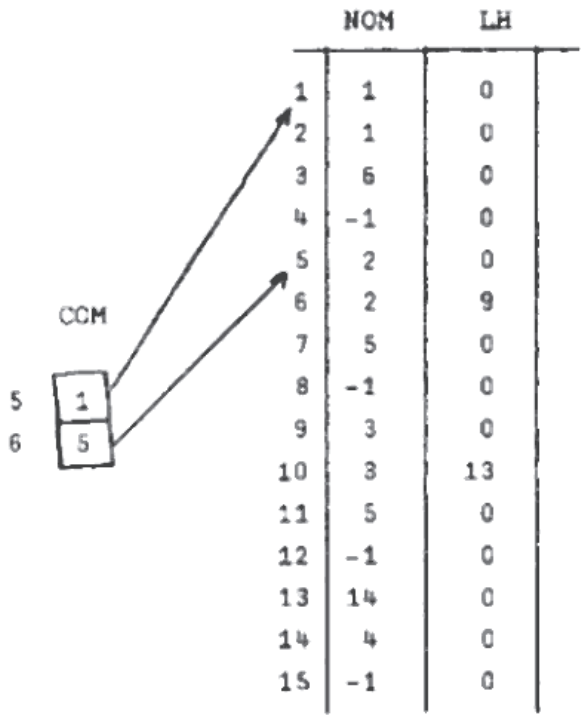
3.3.3 Estructura del programa objeto

Según lo que hemos dicho la estructura del programa objeto (en nuestro caso en ALGOL-60 de la HP 2116B) sería:

```
HPAL,L,B
BEGIN
_____... Declaración de los objetos del programa
BOOLEAN PROCEDURE CB(R); VALUE R; INTEGER R;
   BEGIN
   CASE R
   BEGIN
        CB:=...;...Condiciones booleanas ordenadas
   END;
   END;
PROCEDURE EJEC(B); VALUE B; INTEGER B;
   BEGIN
   CASE B
   BEGIN
           ....Tratamientos terminales ordenadas
   END;
   END;
INTEGER T,N,G;
    .
    .       .... Metaprograma general
   ..
END;
END $
```

Damos a continuación el programa objeto generado así como su ejecución para el ejemplo propuesto en 2.2 (EJEMPLO 1).(Anexo 1).

ya que en este caso: T = 4, N = 6, G= 15 y los vectores NOM, LH y COM vienen dados por



NOTA: Recordemos que el número de regla precede a cada parte de recha en la representación de la ramificación.

EJEMPLO 2

Damos como 2º ejemplo el programa de búsqueda de un elemento de valor dado (VAL) en una matriz A (I,J);

Programa::= Inicialización Lectura Tratamiento |1|

Tratamiento::= Linea Tratamiento |2| Nada |3|

Linea::= Elemento Linea|4| Nada |5|

Elemento::= igual |6| distinto |7|

Con las condiciones Booleanas:

CB(1):= TRUE;
CB(2):= I ≤ NLIN;
CB(3):= I > NLIN;
CB(4):= J ≤ NCOL;
CB(5):= J > NCOL;
CB(6):= VAL = A(I,J)
CB(7):= VAL ≠ A(I,J);

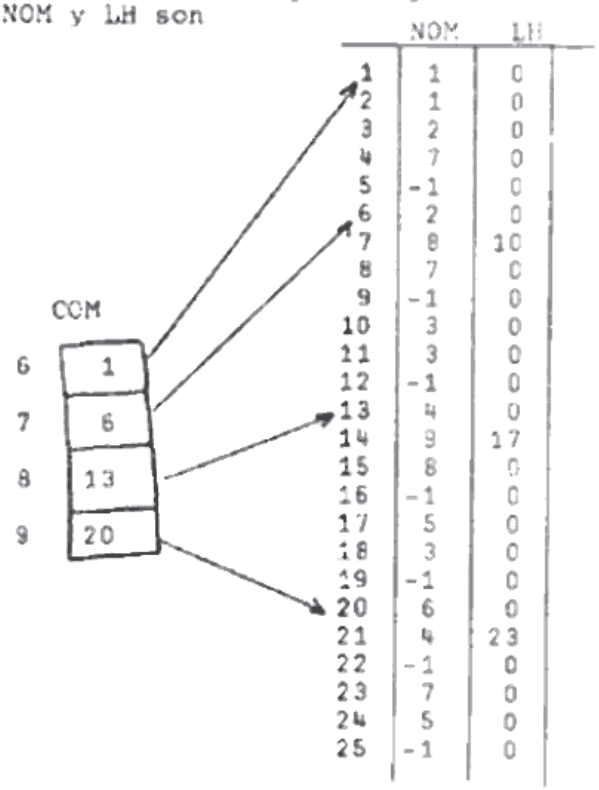
Los tratamientos terminales son:

T = (Inicialización (1), Lectura (2), Nada (3), Igual(4), Distinto(5)) con lo que T = 5.

Los tratamientos no terminales son:

N = (Programa (nº6), Tratamiento (7), Linea (8), Elemento (9)) con lo que N = 9.

Los textos de los tratamientos terminales aparecen ordenados formando el cuerpo del procedimiento EJEC y los vectores COM, NOM y LH son



	NOM	LH
1	1	0
2	1	0
3	2	0
4	7	0
5	-1	0
6	2	0
7	8	10
8	7	0
9	-1	0
10	3	0
11	3	0
12	-1	0
13	4	0
14	9	17
15	8	0
16	-1	0
17	5	0
18	3	0
19	-1	0
20	6	0
21	4	23
22	-1	0
23	7	0
24	5	0
25	-1	0

Damos en las páginas siguientes el correspondiente programa objeto y una ejecución del mismo.(Anexo 2).

3.4 Problemas de determinismo y finitud

- Determinismo: Tal y como hemos concebido la programación estructurada podemos realizar tests en la ejecución para ver si un programa es determinista, es decir, si ante varias alternativas una sola es verdad. Esto necesitamos realizarlo a la ejecución incluyendo una llamada a un procedimiento booleano antes de la ejecución de todo tratamiento no terminal, Todo tratamiento no terminal tiene la forma:

X: ——— |1| ———|2|.....|p|

Obviamente, la condición de determinismo es que ∀ X ∈ N se cumpla:

$$(\exists CB(i) = TRUE \text{ y } \forall\, CB(j),\ j \neq i,\ CB(j) = FALSE)$$

Es decir que entre todos los predicados al menos uno sea verdad (primera condición) pero que no existan más de uno que sea verdad (segunda condición). Lo que conduciría al indeterminismo, puesto que tendríamos más de una alternativa posible.

- Finitud Para que el programa sea finito, debido a la ausencia de bucles (no hay go to...) se debe de cumplir que:

$$(\forall N_i \in N | N_i \overset{+}{\succ} \phi N_i \phi' \text{ y } \phi \overset{*}{\succ} \varepsilon)$$

con ϕ, $\phi' \in V^*$ y ε = cadena vacía.

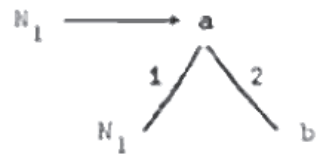
La relación $\overset{+}{\succ}$ es el cierre transitivo de la relación ::= (siendo $\overset{*}{\succ}$ el cierre transitivo y reflexivo de ::=).

Con el significado siguiente:

$\phi \overset{*}{\succ} \varepsilon$?
- $\phi = \varepsilon$: lo que provoca una serie infinita de llamadas recursivas sin efectuar ningún tratamiento.
- $\phi \overset{+}{\succ} \varepsilon$

La cadena vacía la interpretamos como una cadena de longitud arbitraria de tratamientos terminales: $T_1 T_2 \ldots T_n$ tales que ($\nexists T_i$, i=1,n)| que modifique el predicado formado por la palabra CB(p)...CB(q) obtenida en la derivación $N_i \overset{+}{\succ} \phi N_i \phi'$ en un tiempo finito).

Demos un ejemplo: supongamos el programa (dado por su arbol programático)



o séa: N_1 ::= a N_1 |1| b |2|

con: $N_1 \in N$ y a,b $\in T$

Si el tratamiento a no modifica CB(1) y este es inicialmente verdad, el programa será infinito.

No obstante solo podemos detectar algunos casos de infinitud. Veamoslo mediante dos ejemplos:

PROGRAMA INFINITO NO DETECTABLE

```
Programa ::= a Programa |1| b |2|
Objeto del programa N:= 4;
a: N: = N-1;
CB(1):= N < 5;
CB(2):= N = 5;
```

PROGRAMA INFINITO DETECTABLE

```
Programa ::= a Programa|1|  b  |2|
Objetos del programa N:= 4, M = 0;
```

```
a: M:= M+1
b: arbitrario;
CB(1):= N < 5;
CB(2):= N = 5;
```

ya en la traducción del programa fuente al programa objeto podemos detectar que el programa es infinito, puesto que el tratamiento a no afecta a CB(1).

En definitiva pues, podemos resolver los problemas de determinismo siempre y los de finitud solo en algunos casos.

4. CONCLUSION

Damos una aproximación a la programación estructurada que creemos natural, en el sentido de que especificamos secuencialmente los tratamientos a realizar.

Por otra parte el hecho de nombrar subcadenas de tratamientos (caso de los tratamientos no terminales) nos parece también una forma usual del pensamiento humano (es el propósito de la terminología). Construimos un metaprograma general (que implementamos en ALGOL 60 HP 2116B) y lo aplicamos a dos ejemplos. Al mismo tiempo sugerimos un sistema de programación en el que las dificultades de confección de un programa dependen solo de la estructura de este y no del lenguaje de programación escogido, que en nuestro sistema aparece como parámetro.

Deslindamos claramente lo que es estructura del programa de los objetos y en particular, el vernos obligados a enumerar claramente los predicados que se deben cumplir, para escoger cada una de las opciones, creemos; que ayuda a no cometer errores de programación.

La especificación de los tratamientos terminales en el lenguaje objeto escogido (ALGOL 60, FORTRAN...) facilita también la programación puesto que en general estos (los tratamientos terminales) se reducen a una sola instrucción o a un pequeño número de ellas en las que no aparece ningún aspecto de control (estructura) del programa (suelen ser programas lineales).

5. BIBLIOGRAFIA

(1) Böhm, C y Jacopini, G., "Flow Diagrams, Türing Machines and languages with only two formation Rules". Communications of the ACM. Vol, 9, Nº 5, May 1966.

(2) Sigplan Notices, 1974,75.....

(3) Pair, C., "Sur des notions algebriques liées a l'Analyse syntaxique". R.I.R.O. (4e année, R-3, 1970, pp 3-29).

(4) Pair, C.; "Les structures de l'information" Ecole AFCET Informatique Ales 1971.

(5) Dijkstra, E., "Notes on structured Programming". Report EWD, Technische Hogeschool. Eindhoven, 1969.

```
!PROG,ALGOL
PAGE  001


001 02400 HPAL,L,B
002 02000 BEGIN
003 02003 INTEGER NN←0,NP←0,NO;
004 02003       BOOLEAN PROCEDURE CB(R);VALUE R;INTEGER R;
005 02006            BEGIN
006 02010            CASE R
007 02011              BEGIN
008 02014              CB←TRUE;
009 02017              CB←NO<0;
010 02022              CB←NO>0;
011 02027              CB←NO=0;
012 02034              END;
013 02045         END;
014 02047        PROCEDURE EJEC(B);VALUE B;INTEGER B;
015 02053                BEGIN
016 02053                CASE B
017 02053                BEGIN
018 02056                READ(1,#(13),NO);
019 02071                NN←NN+NO;
020 02075                NP←NP+NO;
021 02101                WRITE(2,#("NN",I5,3X,"NP",I5),NN,NP);
022 02127                END;
023 02140            END;
024 02141        INTEGER T,N,G;
025 02141        LABEL RE,L,FIN;
026 02144          READ(1,#(3I2),T,N,G);
027 02163        BEGIN INTEGER ARRAY NOM,LH(1:15),COM(5:6),PILA(1:20);
028 02250        INTEGER H,J,B,R;
029 02250        H←0;
030 02252        READ(1,#(2(15I2/),2I2),FOR NO←1 TO 15 DO
031 02320              (NOM[NO]),FOR NO←1 TO 15 DO (LH[NO]),
032 02337         FOR NO←5 TO 6 DO (COM[NO]));
033 02363        J←COM[T+1]
034 02366        ;
035 02373     L: R←NOM[J];
036 02400        J←J+1;
037 02403        IF NOT CB(R) THEN IF LH[J]#0 THEN BEGIN J←LH[J];
038 02424                                            GO TO L;
039 02425                                          END
040 02425                                    ELSE BEGIN WRITE(2,#("ER"));
041 02440                                            GO TO FIN
042 02441                                          END;
043 02441    RE: R←NOM[J];
044 02446        IF B>0 THEN BEGIN IF B>T THEN BEGIN H←H+1;
045 02462                                          PILA[H]←J;
046 02471                                          J←COM[B];
047 02476                                          GO TO L
048 02477                                        END
049 02477                                    ELSE BEGIN EJEC(B);
050 02503                                          J←J+1;
051 02515                                          GO TO RE
052 02527                                        END;
053 02527                 END;
054 02527        IF H#0 THEN BEGIN J←PILA[H]+1;H←H-1;GO TO RE END
055 02524            FIN;
056 02524 END
057 02524 END;

PAGE  002

ACCIONES= 000525 BASE PAGE= 013765  ERRORES=000
```

EJEMPLO 1 (EJECUCION DEL PROGRAMA)

```
 4 515
 1 1 6-1 8 9 5-1 3 3 5-1 4 4-1
 8 [illegible] 813 [illegible]
 1 5
1R0
-19
1R1
-15
[illegible]
-15
[illegible]
[illegible] -4     [illegible] 364
[illegible]
[illegible]
```

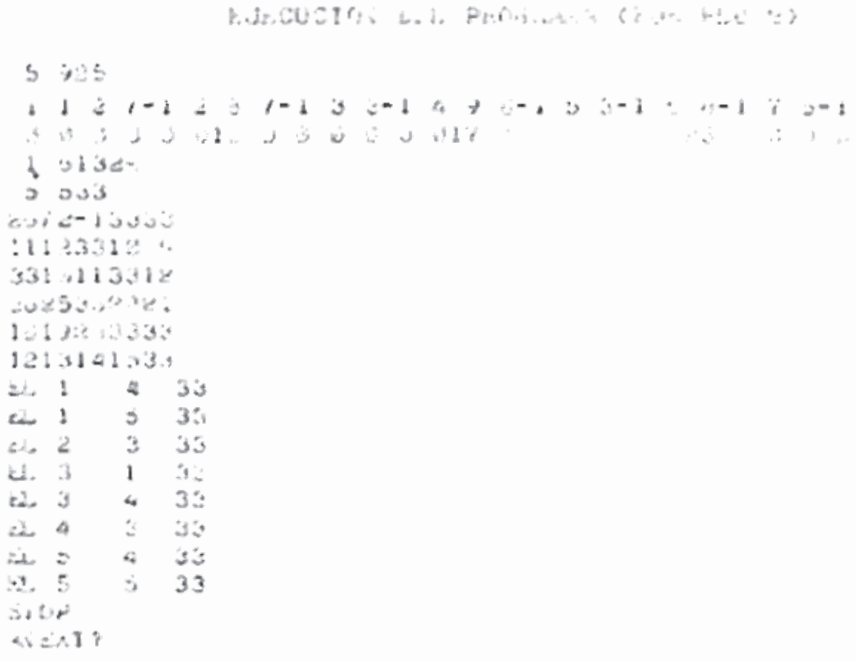
ANEXO 2 (EJEMPLO 2)

```
...ALGOL
PAGE  001


001 02800 ...VAL,L,B
002 02800 BEGIN
003 02933 INTEGER K,L,NLIN,NCOL,VAL;INTEGER ARRAY A[1:5,1:5];
004 02935   INTEGER I,J;
005 02935         BOOLEAN PROCEDURE CB(R);VALUE R; INTEGER R;
006 02937           BEGIN
007 02942           CASE R
008 02942           BEGIN
009 02945           CB:=TRUE;
010 02953           CB:=I<=NLIN;
011 02957           CB:=I>NLIN;
012 02966           CB:=J<=NCOL;
013 02975           CB:=J>NCOL;
014 02134           CB:=VAL=A[I,J];
015 02128           CB:=VAL#A[I,J];
016 02134           END;
017 02150           END;
018 02152         PROCEDURE EJEC(B);VALUE B;INTEGER B;
019 02153           BEGIN
022 02156           CASE B
041 02156           BEGIN
022 02161           I:=J:=1;
023 02165           READ(1,#(3I2/5(5I2/)),NLIN,NCOL,VAL,
024 02210           FOR K:=1 TO 5 DO(FOR L:=1 TO 5 DO A[K,L]) );
025 02253           ;
026 02254           BEGIN WRITE(2,#("EL",3(I2,2X)),I,J,A[I,J]);
027 02305           IF J<NCOL THEN J:=J+1 ELSE IF I<NLIN THEN
028 02323           BEGIN J:=1;I:=I+1 END ELSE BEGIN I:=I+1;J:=J+1 END;
029 02337           END;
030 02340           IF J<NCOL THEN J:=J+1 ELSE IF I<NLIN THEN
031 02356           BEGIN J:=1;I:=I+1 END ELSE BEGIN I:=I+1;J:=J+1 END;
032 02373           END;
033 02405           END;
034 02406         INTEGER T,N,G;
035 02406         READ(1,#(3I2),T,N,G);
036 02425         BEGIN INTEGER ARRAY NOM,LH[1:25],COM[6:9],PILA[1:48];
037 02564       LABEL RE,L,FIN;
038 02567             INTEGER K,J,R,H;
039 02567             H:=0;
040 02571             READ(1,#(2(25I2/),4I2),FOR K:=1 TO 25 DO[NOM[K]],
041 02632             FOR K:=1 TO 25 DO[LH[K]],FOR K:=6 TO 9 DO[COM[K]]);
042 02702             J:=CO:[T+1];
043 02712           L: R:=NOM[J];
044 02717             J:=J+1;
045 02722             IF NOT CB(R) THEN IF LH[J]#0 THEN BEGIN
046 02736                                 J:=LH[J];GO TO L END
047 02744                                       ELSE BEGIN
048 02745                                 WRITE(2,#("EN"));
049 02757                                 GO TO FIN END ;
050 02760          RE: R:=NOM[J];
051 02765             IF R>0 THEN BEGIN IF R>T THEN BEGIN
052 02776                               H:=H+1;PILA[H]:=J;J:=COM[R];
053 03015                               GO TO L   END
054 03016                                  ELSE BEGIN
055 03017                               EJEC(R);J:=J+1;GO TO RE END;
056 03026                         END;
057 03026             IF H#0 THEN BEGIN J:=PILA[H]+1;H:=H-1;GO TO RE END;
PAGE  002


059 03043           FIN:;
059 03063 END;
060 03063 END.

PROGRAM 001344  USE PRES= 000063 , ERRORS=000
```



3.º CONGRESO NACIONAL
INFORMATICA Y AUTOMATICA
MADRID - 29/31 OCTUBRE 1975

SESION
V. 31 - B1/01

EVALUADOR SINTACTICO DEL LENGUAJE DE CONTROL DE TRABAJOS DEL SISTEMA OPERATIVO EXEC-8.

por

Autores: .ISIDRO RAMOS SALAVERT (Universidad Autónoma Bilbao)
.ROBERTO MOYA QUILES (UNIVAC
.PEDRO MARTIN YEBRA (UNIVAC)

RESUMEN

El trabajo consiste en la construcción de un procesador de Lenguaje para la prueba de sintaxis del Lenguaje de Control EXEC-8 de las series UNIVAC 11ØØ.

Las características más resaltables son: ser reentrante, estar capacitado para operar de forma standard dentro del Sistema Operativo, comprobar las peticiones de facilidades, catalogar opcionalmente el RUN que está siendo sometido a prueba de sintaxis, o probar la de RUNs ya establecidos en ficheros permanentes o temporales.

Abstract: Due to run streams working in Batch mode are processed by a Interpret, difficulties arise from debugging sets of run control sentences.

SYNTAX works as a compiler does, detecting errors before the control sentences are submitted into the operating system.

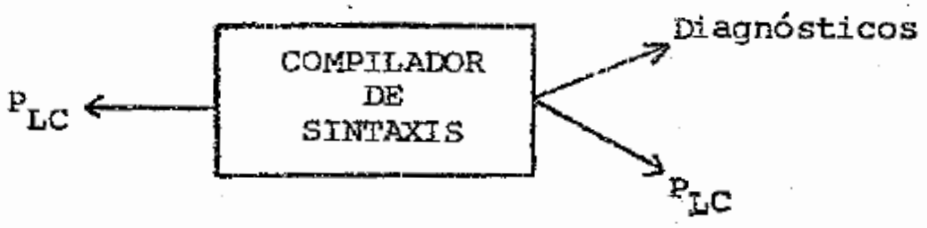
Reentrancy, a standard form operation capability in the operating system, facilities requests verification, testing catalogued RUN's from temporary or permanent files and cataloguing RUN being processed,

Ramos, I., Moya, R., Martín, P. Evaluador Sintáctico del Lenguaje de Control de Comandos del Sistema Operativo EXEC-8. 3er Congreso Nacional de Informática y Automática. 1975.

1. INTRODUCCION

Usualmente el Lenguaje de Control de Trabajos de los Sistemas Operativos es procesado por un Intérprete; esto implica que la evaluación sintáctica de la n-esima Sentencia de Control se pospone a la ejecución de sus precedentes, lo que ocasiona en caso de síntaxis incorrecta, la pérdida de las tareas realizadas hasta ese punto.

En nuestro trabajo construímos un evaluador sintáctico que opera en modo Compilador, permitiendo así la prueba de sintaxis de un conjunto P_{LC} de sentencias previamente a su ejecución, esquemáticamente:



La implementación fué realizada en el procesador UNIVAC 1110 para su Sistema Operativo EXEC-8.

2. DESCRIPCION DEL LENGUAJE DE CONTROL

Se denomina lenguaje de control del sistema operativo EXEC-8 al conjunto de frases reconocidas por su elemento CSI (Coarse Scheduler Interpreter).

La estructura de estas frases es:

@ etiqueta: comando,opciones campos de especificaciones

donde el símbolo @ las caracteriza.

La aparición en un frase de la secuencia /*/ se interpreta como comienzo de comentarios; en tanto que el símbolo ; indica que la frase continúa en el registro siguiente.

Desde un punto de vista semántico, las frases son evaluadas por elementos del sistema operativo o por procesadores software standard o de usuario, dependiendo de los campos de especificaciones y comando.

Una descripción exhaustiva en forma de manual de uso se halla en (1).

sentencia-::=@ [etiqueta -:] [body-]

etiqueta-::= letra-[alfanumérico-...]

body-::= comando-[,opciones-]%... [{,.campo-...}]
| identificador-[,opciones]%... [{,.identificador-

opciones-::= letra-...

campo-::={/.subcampo-...}

subcampo-::=[identificador-|constante-]

comando-::= ASG|BRKPT|CAT|COL|CULL|END|ENDCL|ENDF|ENDX|EOF|
FIN|HDG|JUMP|LOG|MODE|MSG|PMD|RSTRT|RUN|SETC|
START|SYM|TEST

identificador-::=[[[cadena-]%]cadena-[(ciclo-)][{/.llave-...}
:]cadena-[/cadena-][(ciclo-)]

cadena-::=secuencia-...

secuencia-::= alfanumérico-|=|$

alfanumérico-::= letra-|dígito-

constante-::= ASCASC|ASCFD|ASCII|BCD|C|CB|C9|D|DC|DCS|D14|I
D24|D4|D4Ø|D8|EBCDIC|F|FCS|F14|F17|F2|F24|F4|
F6Ø|F8|OFF|POS|PRINT$|PUNCH$|Q|T|TRK|U|UB|U9|
U9V|XSASCI|XSEBCD|11ØØAS|11ØØFD|12|12D|12N|16
16N|2A|2ØN|3A|4C|6|6C|8|8C| 9ØØØAS|9ØØØFD

llave-::símbolo-...

ciclo-::=[+|-] dígito-...

letra-::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y

dígito-::=Ø|1|2|3|4|5|6|7|8|9

símbolo-::= cualquier símbolo del computador.

Las variables metalinguísticas se han representado en m
culas, caracterizándose por llevar un guión. La elipsis (..
indica la repetición. Las llaves indican elección obligator
Paréntesis cuadrados indican opcionalidad.

3. EL COMPILADOR DE SINTAXIS

3.1. Características Generales

Usualmente la compilación de las sentencias de salto de un programa fuente, escrito en un lenguaje que permite referencias hacia adelante, requiere la inicialización previa de la tabla de símbolos (TDS) mediante una primera lectura del programa.

El Lenguaje de Control de EXEC-8, permite evitar la doble lectura del conjunto de sentencias fuente, al asumir que la definición de la "etiqueta" objeto de una sentencia de salto (que siempre es adelante) tiene lugar en dicha sentencia.

La estrategia de análisis elegida es la "top-down" partiendo del axioma de la gramática hasta las unidades sintácticas elementales, habiendo desglosado este camino en los tradicionales pasos de análisis sintáctico y morfológico.

3.2. Analizador Morfológico

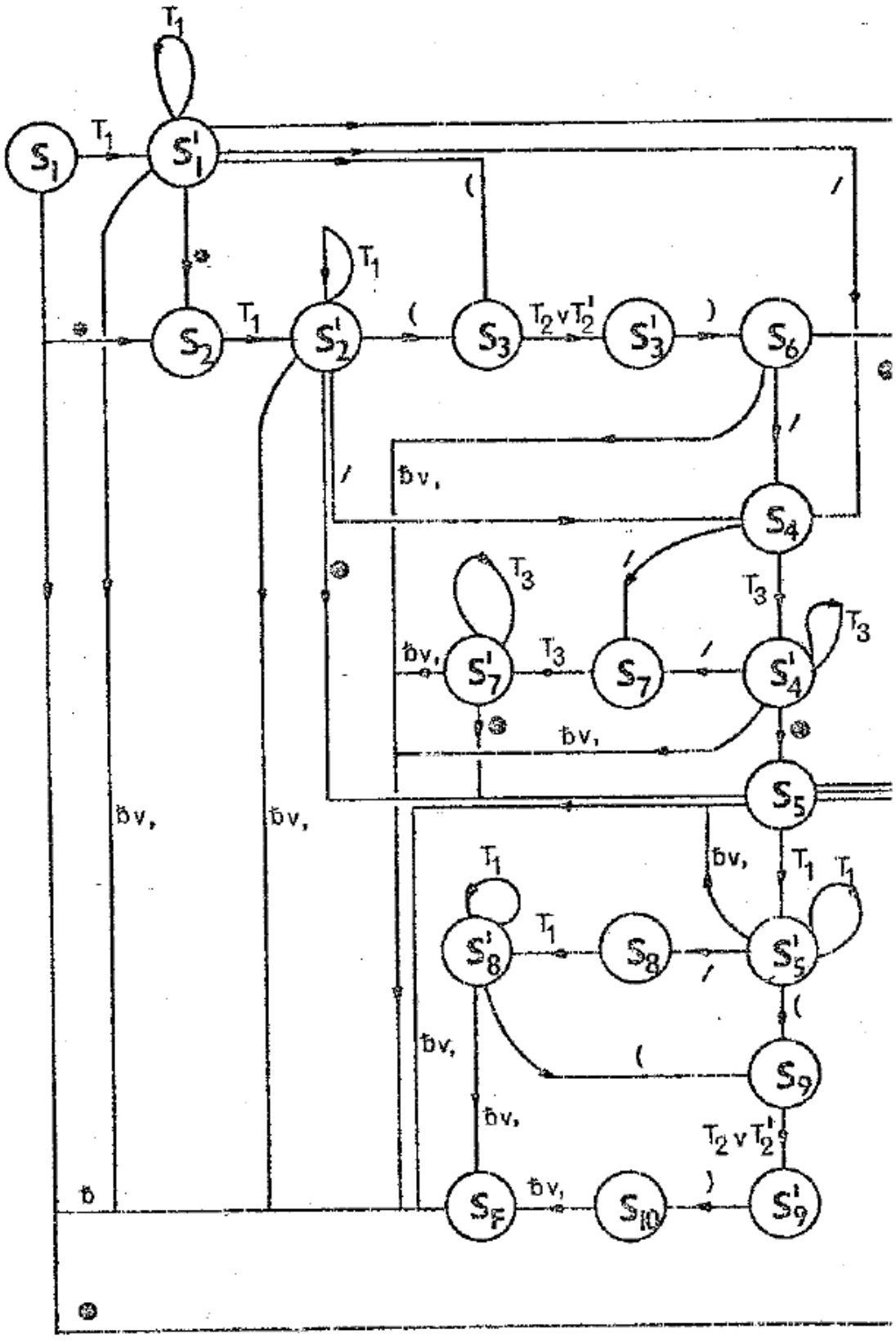
Está encargado de leer e imprimir las sentencias fuente junto con los mensajes de error si los hubiere, gestionar la tabla de símbolos y reconocer las unidades sintácticas :identificador- y etiqueta-.

El proceso seguido para reconocer estas unidades sintácticas se describe mediante un autómata regular cuyo diagrama de estados aparece en figura 1, donde T_1, T_2 y T'_2, T_3 corresponden respectivamente a las metavariables: secuencia-, ciclo-, símbolo- de nuestra gramática.

3.3. Analizador Sintáctico

Encargado de reconocer la clase sintáctica sentencia-, en la clase body- decide la alternativa a tomar y conoce los constituyentes para cada sentencia particular mediante el método "table driven", materializado en la tabla de comandos standard del sistema operativo.

Recibe información del analizador morfológico a través de una estructura de lista.


T1
S1
S'1
T1
T1
S2
T1
S'2
S3
T2vT'2
S'3
S6
bv,
S4
T3
T3
T3
bv,
S'7
T3
S7
S'4
bv,
S5
bv,
bv,
T1
T1
T1
bv,
S'8
T1
S8
S'5
bv,
S9
bv,
T2vT'2
b
SF
bv,
S10
S'9

Considerando como terminales las clases sintácticas: etiqueta-,comando-,opciones-,identificador- y constante- de nuestra gramática especificada en el apartado 2, este analizador sigue la gramática LL(1) siguiente:

sentencia-::= @ contsent-

contsent-::= body-|labeled-sent

labeled-sent ::=etiqueta- : body-

body-::= comando-,opciones- Þ cont1body-|

identificador-,opciones- Þ cont2body-

cont1body-::=campo- cont11body-

cont11body-::=Ø|,campo- cont11body-

cont2body-::=Ø|identificador- cont21body-

cont21body-::=Ø|, identificador- cont21body-

campo-::=Ø|subcampo-/contcampo-

contcampo-::=Ø|subcampo-/

subcampo-::=Ø|identificador-|constante-

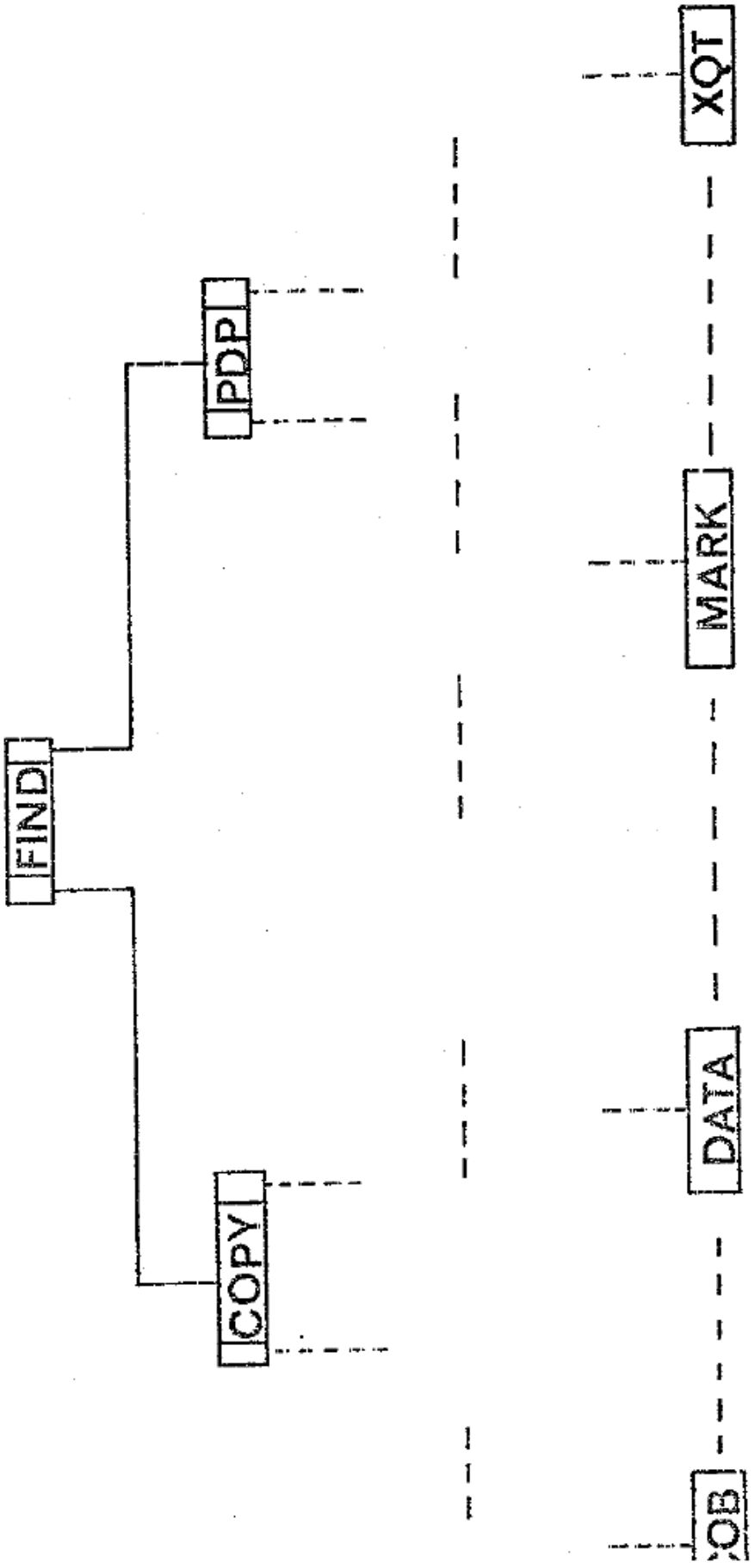


FIGURA 2

3.4. Tablas

3.4.1. Tabla de símbolos

Está estructurada con organización Hash mediante la función de direccionamiento:

dirección vector Hash = f(inicial),

donde inicial es el primer símbolo de la etiqueta.

3.4.2. Tabla de comandos

Está organizada en forma de árbol binario (figura 2), dada la flexibilidad de esta estructura para ulteriores inserciones y el bajo número de accesos requerido en las operaciones de búsqueda.

3.4.3. Tabla de constantes

Al igual que la anterior está organizada en forma de árbol binario, respondiendo a la metavariable 'constante-' de nuestra gramática.

4. IMPLEMENTACION

4.1. Generalidades

El compilador se ha implementado utilizando la tecnología de construcción de traductores TWS (Translator Writing Systems) (3) (4), si bien siguiendo un lenguaje de escritura, que parte de una gramática LL(1), probado anteriormente por nosotros (5), y que ha sido implementado aquí utilizando la facilidad de macro-instrucciones del Ensamblador UNIVAC 1100.

Con el objeto de que el compilador opere de forma standard, se utilizó el software de "interface" con el Sistema Operativo (1).

La estructuración del compilador sobre la memoria se diseñó para que fuese reentrante en un "environment" de multiproceso (utilización simultánea en el tiempo por varios usuarios), constando de dos partes denominadas Banco I y Banco D.

En el Banco D residen exclusivamente las informaciones particulares de cada usuario, como son: tabla de símbolos (TDS); archivo asignado por el EXEC-8; "buffers" de entrada y salida, entre otras. En cada momento habrá tantas copias de este Banco como usuarios estén utilizando el Compilador. El mecanismo de multiplicidad de Banco D está a cargo del Sistema Ope-

6. BIBLIOGRAFIA

(1) UNIVAC 1100 series Operating System Programmer Referen UP-4144.

(2) LUCAS,P., LAUER,P., STIGLEITNER,H.: Method & Notation f the Journal Definition of Programming Languages. IBM TR25,Ø87. IBM Laboratory: Vienna, Austria, 1968.

(3) Knuth: Top-Down Syntax Analysis, International Summer School on Comp.Prog. Copenhagen 1967.

(4) FELDMAN, J. & GRIES: Translator Writing Systems. CACM, brero 1968.

(5) MOYA, R. y RAMOS, I.: Construcción de un traductor. MI Madrid, 1974.

(6) Joachim von Peschke: Syntactic Rules Embedded in PL/1. SIGPLAN Notices,July 1974.

7. APENDICE

```
@.SYNTAX,L                    .  PRUEBA DE SINTAX|S

@RUN,A/RC    SYNTAX,UN4240DDC,EDUCACION

@ASG,A    QSQ3G$*EDUCAC|CN.

@COPY,A    Q$Q3Q$*EDUCAC|ON.SYNTAX,TPFS.SYNTAX

@.SYNTAX,L                    .  PRUEBA DE S|NTAX|S

@ASG,T  PRUEBA-P.,F

@DATA,|D    PRUEBA-P.

@A: . SEGU|DAMENTE PONEMOS EJEMPLOS DE SENTENC|AS CON ERR
ETIQUETA  **A      **  DUPL|CADA O NO REFERENC|ADA

@CUALIFICADOR*FICHERO3$-12(32?).ELEMENTO3$$$/-$$--$

   COMANDO  ELEMENTO3$$$  NO STANDARD

@Q:PEPE.A,ALKJGHL
ET|QUETA  **Q      **  DUPL|CADA O NO REFERENC|ADA

   COMANDO  A              NO STANDARD

@ASG,A     QUALI*FICHER(3)/LLAV/LLVE.ELEM/QU|NCE(45),T

@JUMP   3

@Q:F|CHERO.
ET|QUETA  **Q      **  DUPL|CADA O NO REFERENCIADA

   COMANDO                NO STANDARD

@Q:
ETIQUETA  **Q      **  DUPLICADA O NO REFERENCIADA

@USE       |NTERNO.,A*EXTERNO(1).,F
*CARACTER  F  |LEGAL EN LA POS|C|ON    033

@JUMP      ASGMNH

@JUMP      1ASKLMNH
*CARACTER  A  |LEGAL EN LA POS|C|ON    011
*CARACTER  S  |LEGAL EN LA POS|C|ON    012
*CARACTER  K  |LEGAL EN LA POS|C|ON    013
*CARACTER  L  |LEGAL EN LA POS|C|ON    014
*CARACTER  M  |LEGAL EN LA POS|C|ON    015
*CARACTER  N  |LEGAL EN LA POS|C|ON    016
```

```
*CARACTER  "  ILEGAL EN LA POSICION     017
*CARACTER     ILEGAL EN LA POSICION     018
*CARACTER     ILEGAL EN LA POSICION     019

$BAS,DE:
*CARACTER  3  ILEGAL EN LA POSICION     001
*CARACTER  ,  ILEGAL EN LA POSICION     004

$AEBX:
ETIQUETA  **AEBX  **  DUPLICADA O NO REFERENCIADA

$ASG,T    ABC*NOMBRE/E/F.,F

$ET1:
ETIQUETA  **ET1   **  DUPLICADA O NO REFERENCIADA

$ABCDE:
ETIQUETA  **ABCDE **  DUPLICADA O NO REFERENCIADA

$JUMP,N       0

$QUAL*PEPE(+1).437SP/3P5

   COMANDO  437SP          NO STANDARD

$G:

$JUMP         AEBX

$ASG,T     DEMO(3)/LEER/ESCR.,F4///500

$USE     A.,DEMO(3)/LEER/ESCR.

$1235:
*CARACTER  1  ILEGAL EN LA POSICION     001

$ESLARGA:
*CARACTER  A  ILEGAL EN LA POSICION     007

$ASY,UL   EDUCACION.SYNTAX,EDUCACION.SYNTAXA,EDUCACION.SYNTAX

$MAP    EDUCACION.SYNTAXABSA,EDUCACION.SYNTAX

$END

$.SYNTAX,LP   PRUEBA-P.

$.SYNTAX,A

$.SYNTAX,LPA     PRUEBA-P.
```

© Euromicro, 1976
North-Holland Publishing Co., Amsterdam

CROSS-MACROASSEMBLER FOR MICRO MCS-80 IN A HP-2116-B MINICOMPUTER

I. Ramos J. Aramberri

Departamento de Automática e Informática

Universidad de Bilbao

Bilbao (Spain)

Abstract

A cross-macroassembler has been implemented in a HP-2116-B minicomputer, oriented to the INTEL'S MCS-80 microcomputer. The macro-assembly language allows nested macro-definitions and uses. The "local" macros remain in the macrotable only during the time of using the outer macro(s) and are erased therafter. In a similar way as the variables . . . of a block structured language as ALGOL-60 are managed. In this way the use of the minicomputer storage is optimized in the macro-assembly process.

FORTRAN has been used as the implementation language allowing an easier transportability to other minicomputers. The word length requiriment is 16-bits (normal word length in actual minicomputers).

Following the macroassembly process a two-pass assembler translates the symbolic code to the machine code of the MCS-80. Besides the macrofacilities above mentioned, the whole language has been implemented, and some small improvements have been added.

The use of a minicomputer as a resident computer for this cross-macroassembler allows a more flexible use of the micro-computer, avoiding also the need of a costly Time Sharing Service or a big computer. Our hardware configuration (mini-micro), we think, is a much more normal situation in the Automatic Control Laboratories an Industrial Laboratories than big machine configuration, rending our cross-macroassembler a useful software tool.

1. INTRODUCTION

The use of microprocessors as normal components in more complex designs (control circuits sets ..) grows every day because of the drop of the costs and the easier use of the programming tools. Indeed, the software tools furnished by some constructors are either very poor (assembly language level) or cross-software implemented in costly big computers (or also costly Time-Sharing Services).

In our laboratory (and we think in many other cases) we have a minicomputer (HP-2116-B with a 16-bits word length) and a Microsystem (INTEL'S MCS-80).

The absence of cross-software on minicomputers (in spite of the fact they are the normal computing systems in the Automatic Control laboratories and Industrial Laboratories) has been the motivation of our work: design and implementation of a cross-macroassembler is a minicomputer. We have added also the capability of defining and using macros within macros (the need of a macroprocessor working recursively arises). The storage limitations of normal minicomputers implies then a judicious management of the storage.

The implementation language has been FORTRAN and we have explicitely avoided the particular capabilities of our computer, with the idea of obtaining a macro-processor which is easily transportable.

The macro-assembler runs as a subsystem of MTS (Magnetic Tape Operating System of the HP-2116-B) and accepts as input a program in assembly language (with macros) punched on paper tape, and outputs and absolute objet program in the format accepted by the MCS-80.

The process in done in three steps: the first one is the macro-process (processing of the macro-declarations and macrocalls) and the second and third ones are the two passes of a clasical assembler (symbol table construction and objet code generation). Although the first two steps could have possibly been made into a single one, limitation in the available storage have forced on us the present split-up version.

The system is operating in our laboratory since december 1975.

2. THE LANGUAGE

The implemented language is the Intel 8080 Assembly Language with some improvements:

- A more powerful macroprocessor: we have the possibility of defining and using "local" macros within outer macros. This implies a recursive working of the macroprocessor and a storage management of the macrotable similar to the block structured programming language storage management (a LIFO free list).

- The operand field of the pseudoinstructions SET and EQU may be an arbitrary expression (obviously with an absolute value).

- The labels of the instructions or pseudoinstructions are separated by a semicolon (":") of the operation code field. Normalizing the label format.

As the macroprocessor works in one step the macrodefinitions must appear before they are used. We think that this is not an important restriction.

To avoid ambiguities we give a Context-Free grammar which (the empty string) describing the implemented language.

The blanks may appear freely between the instruction fields

Ramos, I. Aramberri, J. Cross-Macroassembler for Micro MCS-80 in a HP 2116-B Minicomputer. Euromicro. 1976.

and also within the operands and operators of the operand field. This feature is not considered in our grammar for the sake of clarity.

The grammar is:

```
<programa>::=<operación> CR-LF <cont -
             prog> END CR-LF

<cont-prog>::=<operación> CR-LF <cont-
              prog>|ε

<operación>::=<instrucción>|<pseudo-ins-
              trucción>|<def.macro>|<uso
              macro>

<instrucción>::=<etiqueta>:<instrucción>
               |<inst. no etiq.>

<inst. no etiq>::=<ins 1>|<inst 2>|
                  <inst 3>

<ins 1>::= CMC|STC|CMA|DAA|NOP|RLC|RRC|
           RAL|RAR|SCHG|XTHL|PCHL|REJ|
           RZ|RNC|RC|RPO|RPE|RP|RM|EI|
           DI|HLT

<inst 2>::=<CO 2><reg>|<CO 3><preg>|
           <CO 4><dato inmediato>|<CO 5>
           <dirección>|<CO 7><expresión>

<inst 3>::=<CO 6><expresión>,<expre-
           sión >

<CO 2>::= INR|DCR|ADD|ADC|SUB|SBB|ANA|
          XRA|ORA|CMP|

<reg>::=  B|C|D|E|H|L|M|A|<expresión>

<CO 3 ::= STAX|LDAX|PUSH|POP|DAD|INX|
          DCX |

<preg>::= B|D|H|SP|PSW|<expresión>

<CO 4>::= ADI|ACI|SUI|SBI|ANI|XRI|ORI|
          CPI

<dato inmediato>::=<expresión>

<CO 5>::= SHLD|LHLD|STA|LDA|JMP|JNZ|JZ|
          JNC|JC|JPO|JPE|JP|JM|CALL|CZ|
          CNC|CC|CPO|CPE|CP|CM

<dirección>::=<expresión>

<CO 6>::= MOV|LXI|MVI

<CO 7>::= RST|IN|OUT

<expresión>::=<expresión>OR<term>|<expre
              sión>XOR<term>|<term>

<term>::=<term>AND<factor>|<factor>

<factor>::= NOT<fact 1>|<fact 1>

<fact 1>::=<fact 1><opadit><term 1>|
           <term 1>

<op.adit>::= + |-

<term 1>::=<term 1><op multip><fact 2>|
           <fact 2>

<op multip>::= *|/|MOD|SHL|SHR

<fact 2>::= -<fact 3>|<fact 3>

<fact 3>::=<cte>|<identif>|(<expresión>)

<cte>::=<B >|<0>|<D>|<H>|<A>|$

<B>::=<db><rdb>B

<rdb>::=<db><rdb>|ε

<db>::= 0|1

<0>::=<do><do><do><do><do><OQ>

<do>::=<db>|2|3|4|5|6|7|ε

<OQ>::= O|Q

<D>::=<dd><dd><dd><dd><dd><PD>

<PD>::= D|ε

<dd>::=<do>|8|9|ε

<H>::=<dd><dH><dH><dH><dH>H

<dH>::=<dd>|A|B|C|D|E|F|ε

<A>::= '<ca><ca'>'

<ca'>::=<ca>|ε

Fca ::= "caracteres ASCII"

<identificador>::=<letra'><resto de iden-
                  tificador>

<resto de identificador>::=<letra><rest.
                           ident.><dd>
                           <rest.ident.>
                            ε

<letra'>::= @ |?|<letra>

<letra>::= A|B|C|.....|Z

<etiqueta>::=<identificador>
```

```
<pseudoinstrucción>::= DB<lista>|DW<lis-
                       ta>|DW<expresión>
                       |ORG<expresión>|
                       EQU<expresión>|
                       SET<expresión>|IF
                       <expresión>|ENDIF
                       |<etiqueta>:<pseu
                       doinstrucción>

<lista>::=<expresión><cont. lista>

<cont.lista>::= ,<expresión><cont.lista>
                |ε

<def.macro>::= MACRO CR-LF<int.macro>
               CR-LF MEND CR-LF

<int.macro>::=<linea nombre>CR-LF<cuerpo
              macro>

<linea nombre>::=<nombre><lista 1>

<nombre>::=<letra><resto nombre>

<resto nombre>::=<letra><resto nombre>|
                 <d.d.><resto nombre>|ε

<lista 1>::=<par 1><rest.par>|ε

<rest.par>::= ,<par 1><rest.par. |ε

<par.1>::= &<letra><rest.par 1>|&<d.d.>
           <rest.par.1>

<rest.par.1>::=<letra><rest.par.1>|<d.
              d.><rest.par.1>|ε

<cuerpo macro>::=<linea&>CR-LF<cont.cuer
                 po >

<cont.cuerpo>::=<linea&>CR-LF<cont.cuer-
                po>|ε

<c.op.&>::=<c.op.>|<par>

<cte.&>::=<cte.>|<par>

<identificador&>::=<identificador>|<par>

<uso macro>::=<eti>:<nombre macro><lista
              actual>|<nombre macro><lis
              ta actual>

<lista actual>::=<cte.><rest.lista>|
                 <eti><rest.lista>|ε

<rest.lista>::= ,<cte.><rest.lista>| ,
                <eti><rest.lista>|ε
```

The semantics of the language is given in the INTEL 8080 Assembly Language Programmers Manual (2).

We have not considered in our grammar some "semantic" rules (maximum value of the differents constants...), this is doing by the evaluation routines.

By "Caracteres ASCII" we understand the set of "ASCII" characters (1-Appendix C).

3. THE MACROPROCESSOR

Following are the basic characteristiques of the macroprocessor:

- As we have already said the macroprocessing is done in one step.

- A macrocall may have from zero to ten parameters.

- A complete field or part of it may be substituted by an actual parameter of a macrocall.

- The macrodefinitions may be nested until a maximum depth of 5 (arbitrarily set and easily changeable). A macrodefinition is a valid one (is present in the macrotable) if the macro within wich is defined is in use. The macros of level "O" are always valid and are called "absolute" macros, we call all the others "local" macros. This implies a LIFO storage management.

A set of errors are detected in this step.

A flowchart of the macroprocessor action appears in Figure 1.

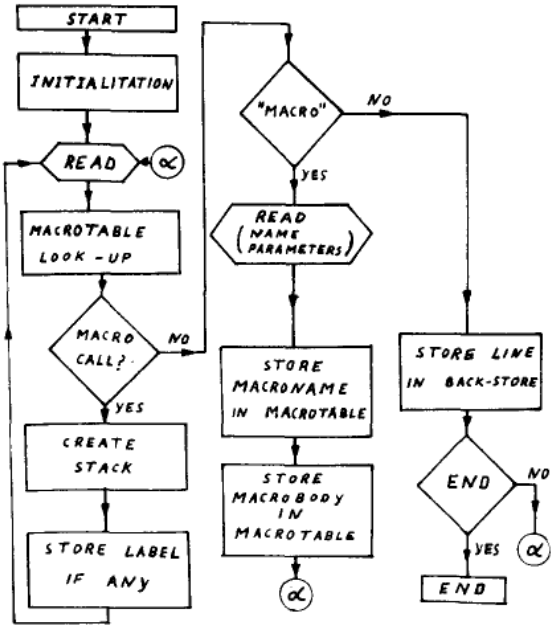


Figure 1.

Flowchart of the macroprocessor.

The macroprocessor implementation follows the ideas of Donovan (1).

The Free list (integer array IEL) is managed as a contiguos segment using two pointers (IEL (1) and IEL (2)). See Figure 2.

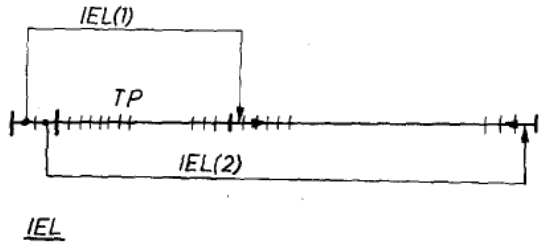


Figure 2.

Macrotable.

As macrotable we use a hash table with linked lists for the owerflows. In figure 2, TP is the Primary Table.

We allocate in the sense of the decreassing adresses the "triplets" forming the collision lists, for each entry in the Primary Table with the following structure (see Figure 3).

pointer to the external macro-name	pointer to the macrobody	List Link

Figure 3.

Triplets of macrotable.

The external name and the macrobody are stored contiguosly following the increasing sense of the adresses (the first word conteins the length of the field in words).

Let us give an example:

Macrodefinition

```
MACRO
SUMAR   &   DAT1,&DAT2,&DAT3
LDA     &   DAT1
ADA     &   DAT2
STA     &   DAT3
MEND
```

The macrotable status after the macrodefinition is shown in Figure 4.

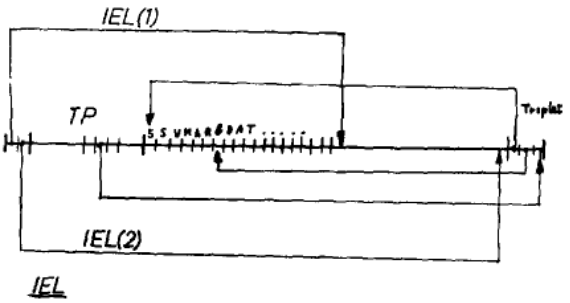


Figure 4.

The macroname and the virtual parameters appear at the beginning of the macrobody.

Because the feature of using "local" macros the Macroprocessor works recursivily. It is a well known theoretical result that every recursive process has an equivalent iterative process working on a stack. In our case the actual parameters are pushed in the stack at every call in the way shown in Figure 5.

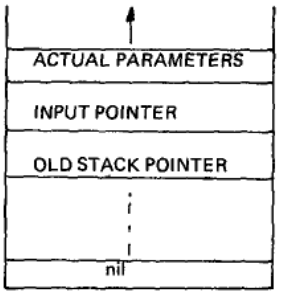


Figure 5.

The macrobodies of the absolute macros remain in the macrotable all the time, but the bodies of the local macros which are no longer necessary at the end of the call of the outer embedding macro are then suppresed. To achieve this we store in the variables IEL 1 and IEL2 the actual values of the pointers used in the management of the Freee List (IEL(1) and IEL(2)) and in IBOR the adresses where we have linked the names of the suppresed macros. In this way we can set nil (marking the end of the lists).

The whole process is sumarized in Figure 6.

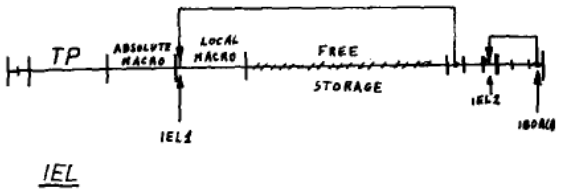


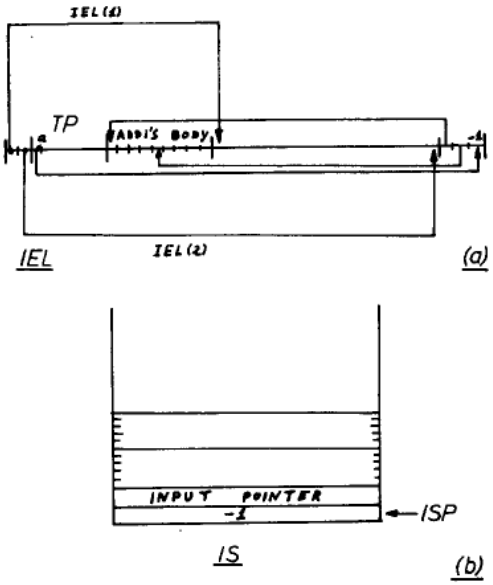
Figure 6.

In the stack ISP we store the context of the macro during its macroexpansion, bacause we can have several others macro-calls within the macrobody. This allows us to restore the "old" context at the end of the expansion of the inner macro. This is a usual process in the storage management of well parenthized or block structured Programming languages.

We can see the whole process in the following example:

```
SOURCE PROGRAMM
ORG     1000
MACRO
ADDI    &TER1;&TER2
MACRO
ADD     &TER
CLA
LDA     4000
ADA     &TER
STA     2000
MEND
ADD     &TER1
CPA
OTA     11
ADD     &TER2
OTA     11
MEND

ADDI    200,300
```



In the first step we include the absolute macro ADDI in the macro table from the beginning until the ending MEND, including the internal macro ADD (see Figure 7-a).

At the call ("ADDI 200, 300") we store in the stack the actual parameters in the form exhibited in Figure 7b.

We start the macro ADDI. We find a macro definition (ADD) and then we store it in the macrotable (Figure 7c).

We continue expanding the macro ADDI. We find a macro-call for ADD and the stack changes to the configuration shown in Figure 7d.

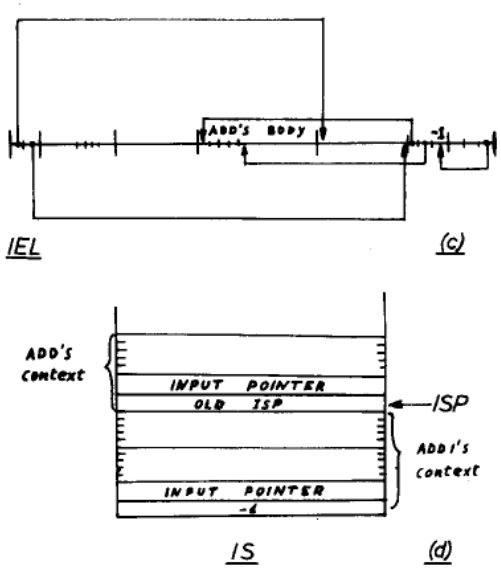
At the end of the macro-call, the stack is restored to the previous configuration. With the new macro call "ADD 300" we have the same process. At the end of the absolute macro "ADDI" the stack is empty (the control pointer is set to nil which is represented as -1).

4. ASSEMBLER

The output of the macroprocessor (assembly language) is input to a classical assembler working in two steps.

Step 1 performs a syntactic analysis on each instruction and pseudo-instruction checking its correctnees and typing the possible errors. It also builds the symbol table asigning the corresponding value to each label. It changes the symbolic operation codes to the binary equivalents (this binary code may be updated following the evaluation of the operand field performed in step 2, this depends of the operation type). An intermediate code is generated on magnetic tape with the following structure.

$ INSTRUCTION TYPE$ OPERATION CODE$ OPERAND FIELD ELABORATED$ SOURCE INSTRUCTION.



Some pseudoinstructions are completely evaluated in Step 1: ORG, SET, EQU... The operand field is partly elaborated.

The symbol table is hash table with the same structure than the macrotable used in the macroprocess (overflow with chaining).

In fact we could have done the macroprocessing at the same time as the Assemblers Step 1 is done, but storage limitations were an important reason to avoid this solution.

Step 2. Inputs the intermediate code and symbol table which constitutes the out-put of the step 1, and generates objet code acording with the following record format:

: N SA HC CB

N: number of bytes (expressed as two hexadecimal characters) in the record.

SA: starting address

HC: hexadecimal code

CB: control byte (longitudinal parity control byte)

That is the objet code accepted by the MCS-80 loader.

5. CONCLUSION

1st We have implemented a powerful cross macroassembler allowing "local" macros and a classical two step assembler.

2nd FORTRAN has been used as implementation language on a minicomputer (HP-2116-B), allowing an easy transportability.

3rd We have done a modular programming allowing a more flexible implementation and a easier understanding of the process.

4th The macroprocessor accepts as input an arbitrary input programm writing in assembly language.

6. REFERENCES

1. DONOVAN "Systems Programming" Mc.Graw Hill, 1972.

2. INTEL 8080 "Assembly language Programming "Manual"

7. APPENDIX

Macroprocessor

Input:

```
        ORG     100H
DIV:    MVI     E,9
        MACRO
        REP &A1,&A2,&OPE
        MOV &A1,&A2
        &OPE        :SERA RAL O RAR
        MEND
DI1:    MOV A,B
        MOV B,A
        REP A,C,RAL
        MOV C,A
        DCR E
        JZ  DI2
        REP A,B,RAL
        SUB D
        JNC DI1
        ADD D
        JMP DI1
DI2:    RAL
        MOV E,A
        MVI A,OFFH;COMPLEMENTAR
        XRA C
        MOV C,A
        REP A,E,RAR
        HLT
        END
```

Output:

```
        ORG 100H
DIV:    MVI , E,9
        MOV A,B
DI1:    MOV B,A
        MOV A,C
        RAL     ;SERA RAL O RAR
        MOV C,A
        DCR E
        JZ  DI2
        MOV A,B
        RAL     ;SERA RAL O RAR
        SUB D
        JNC DI1
        ADD D
        JMP DI1
DI2:    RAL
        MOV E,A
        MVI A,0FFH;COMPLEMENTAR
        XRA C
        MOV C,A
        MOV A,E
        RAR     ;SERA RAL O RAR
        HLT
        END
```

Assembler STEP 1

Input: The output of the macroprocessor

Output:

```
        ORG     100H
     80      15     256   16    0 ...

DIV:    MVI     E,9
     73      6      14    3    18  14  9 16  0...
        MOV     A,B
     71      64     14    7    18  14  0 16  0...
DI1:    MOV     B,A
     71      64     14    0    18  14  7 16  0...
        MOV     A,C
     71      64     14    7    18  14  1 16  0...
        RAL             ;SERA RAL O RAR
      8      23     16    0...
```

Assembler STEP 2

Input: The output of the Assembler STEP 1

Output: Listing

```
1  1  0
 1              0100            DIV
 1              0103            DI1
 1              0115            DI2
                       ORG   100H
0100  1E09     DIV:    MVI    E,9
0102  78               MOV   A,B
0103  47       DI1:    MOV   B,A
0104  79               MOV   A,C
0105  17               RAL       ;SERA RAL O RAR
0106  4F               MOV   C,A
0107  1D               DCR   E
0108  CA1501           JZ     DI2
010B  78               MOV   A,B
010C  17               RAL       ;SERA RAL O RAR
010D  92               SUB    D
010E  D20301           JNC   CDI1
0111  82               ADD    D
0112  C30301           JMP    DI1
0115  17       DI2:    RAL
0116  5F               MOV   E,A
0117  3EFF             MVI    A,0FFH;COMPLEMENTAR
0119  A9               XRA   C
011A  4F               MOV    C,A
011B  7B               MOV   A,E
011C  1F               RAR       ;SERA RAL O RAR
011D  76               HLT
```

END

PROCESAMIENTO DEL PASO 2 FINALIZADO

Objet Programm

```
:100100001E09784779174F1DCA1501781792D20337
:0E0110000182C30301175F3EFFA94F7B1F76DC
:00010000FF
```

INFORMATIQUE DE BASE: THESE, ANTITHESE, SYNTHESE

I. RAMOS
Prof Informatique
Université de Bilbao (Espagne)

En comparant l'Informatique à des disciplines qui lui sont affines on peut dégager des idées sur son futur, en particulier la Linguistique et les Mathématiques ont fourni des situations homomorphes au Logiciel de base. Les solutions qui ont été prises ainsi que les tendances dans le Matériel et le Logiciel montrent que l'évolution du Logiciel de base de demain se fera dans une approche globale du système homme-machine et non plus dans une interdépendance chaque fois plus poussée entre les deux. Nous avons appelé cela PROGRAMMATION DIALECTIQUE ou FLOUE dans laquelle la conversationnalité réelle et l'Heuristique ainsi que l'Intelligence Artificielle seront les disciplines plus concernées.

It is making analogies between Computer Science and others similar Sciences that we have obtained some ideas about its future: Classical Linguistics and Mathematics have homomorphic situations with programming and software. The solutions that have been adopted together with the trends toward the future in hardware and software show that tomorrow's software will be done with an unified vision of the system man-machine and not as a result of an even more deep inter-relation between both.

We have named this approach Dialectic PROGRAMMING OR FUZZY PROGRAMMING where real converstionality and Heuristics, together with Artifical Intelligence will be the disciplines more concerned.

1.- THESE

1.1.- Considérations historiques et analogiques.

La Méthodologie scientifique trouve dans l'intuition et l'analogie deux outils puissants de création, et nous allons nous en servir dans cette approche prospective au Logiciel de base: intuition fondée sur notre expérience des années de travail en Logiciel de base et analogie historique entre l'Informatique et les Sciences qui lui sont plus proches: Les Mathématiques et la Linguistique Classique (d'après les idées de P. Naur, "Programming languages Natural Languages and Mathematics" CACM Dec. 1.975)

Dès ce point de vue et en considérant des situations homomorphes nous aboutirons à des conclusions en Informatique en partant de faits qui se sont produits avant en Mathématiques et Linguistique, puisque ces deux sciences la précédent historiquement.

Au dedans de l'Informatique nous allons nous constraindre au Logiciel de base bien que, nous croyons, nos conclusions peuvent s'appliquer à bien d'autres aspects

En Informatique il existe un "Grand Trou" entre Théorie et Pratique entre l'Informatique de nos universités et l'Informatique appliquée de nos entreprises, bien que nous-tous nous considérons informaticiens et croyons que la "vrai" Informatique se trouve de notre côté. Il est intéressant de lire dans cet esprit le travail de S. Gorn "Status of Computer Methodology : when the chips are down" CACM Nov. 75. La même situation existe entre les Mathématiques dites Théoriques et Appliquées et il paraît que la synthèse est lointaine: bien au contraire le Trou est chaque fois plus grand.

Les mathématiques théoriques trouvent dans la beauté de leurs théorèmes la justifications de sa Science, tandis que les "appliqués" seulement trouvent leur sens dans l'efficacité de leurs applications et encore il-y en a d'autres qui disent que plus on est théorique plus on est practique ("Il n'existe meilleure Pratique qu'une bonne théorie") et etc

Je crois qu'il s'agit d'une polémique assez proche et connue pour nous les informaticiens pour en parler davantage.

En tout cas, le fait de la division des Sciences en théoriques et practique est un résultat de la spécialisation et de l'impossibilité d'être globalement compris par l'homme après un certain temps de développement.

- Les préciosismes sans fin dans le développement des langages de programmation (cfr. ALGOL 68, pour ne pas parler de la Théorie des Langages...) contrastent très fortment avec
- La Programmation massive en COBOL réalisée dans nos entreprises.

Si nous acceptons l'analogie Informatique-Mathématique nous pouvons obtenir le résultat suivant:

- Cette duplicité existe et existera chaque fois d'une façon plus claire et explicite.

Cette situation dialectique conduira dans un futur non pas à l'union des deux tendances mais à une certaine façon de concevoir le Logiciel de base où l'Intelligence Artificielle, la Théorie des Langages... avec les besoins chaque fois plus durs en programmation changeront et révolteront le concept de Logiciel de Base tel que nous le connaissons aujourd'hui.

La deuxième discipline que nous considérons proche à l'Informatique est la Linguistique Classique comme tout homme travaillant dans le Logiciel de Base sait (Il suffit de considérer les travaux de Chomsky).

Dans cette discipline, l'étude des différences qui la séparent de l'Informatique et en particulier les deux suivantes sont très intéressantes à considérer:

- Les langues naturelles sont utilisées surtout oralement, la capacité d'expression orale étant très developpée dans l'homme. La mise au point interactive de ses idées en conversant avec d'autres interlocuteurs est aussi un procès normal.

Cependant la programmation est faite fondamentalement de façon écrite et avec un degré d'interaction très pauvre et inefficace (il suffit de réflechir sur les langages conversationals et interpretatifs);

- Sauf dans certains cas la langue parlée est vague et imprécise, tantôt du côté de celui qui la parle, tantôt du côté de celui qui l'écoute.

Cependant en Informatique les exigences de précision sont considérables. Cette différence n'est pas en aval des langages de programmation puisque tout ce qui peut être exprimé à l'aide d'un langage de programmation peut l'être aussi avec une langue parlée. Tandis que l'inverse n'est pas vrai.

Nous croyons que cette manque de précision est la cause qui rend possible le fait de parler d'une façon interminable sur de nouvelles idées, c'est-à-dire: une langue a un univers de discours extensible qui permet d'ajouter d'autres notations (par exemple: les langages de programmation)

C'est important de fixer ces deux idées puisque le proche avenir du Logiciel aura son fondament sur elles. D'un autre côté l'analyse de ce qui s'est passé avec les langues qui ont survécu à leurs histoires en évolutionnant et en s'adaptant, et de l'autre côté l'échec des langues artificielles telles que l'ESPERANTO, nous fournissent deux nouvelles idées:

- La simplicité (puisque l'évolution dûne langue se produit toujours en amont de sa complexité) et
- Le fait d'avoir survécu (nous assure le consensus sinon la correction de millions d'usagers)

Si nous trasladons ces idées à l'Informatique nous aurons du FORTRAN et du COBOL pour des années, étant donné que c'est eux les langues qui ont resté, tandis que la complexité d'ALGOL-68 (pour parfait qu'il soit) empêche sa diffusion

Une autre différence qu'il faut souligner c'est l'éxistance d'un style littéraire (ou technique littéraire) en Linguistique ce qui a possibilité la création de chef-d'oeuvres en littérature tandis qu'en programmation nous n'avons pas la discipline Homomorphe. Il a été dernièrement et surtout avec la Programmation Structurée que nous nous sommes occupés d'un certain style de programmation qui rend plus legibles, communicables les programmes.

Les conclusions que nous pouvons en tirer sont les suivantes:

- La séparations entre Informatique Théorique et Informatique appliquée continuera et se fera plus claire
- Il est un fait que nous programmons en FORTRAN-COBOL et que nous continuerons à le faire. Il sera très difficile d'introduire de nouveaux langages même s'ils sont très puissants. On voit que la solution ne se trouve pas de ce côté
- Le futur est dans les "Arts" qui cultivent le style de programmation, ce qui peut aboutir à des chefs-d'oeuvres (grands systèmes): Programmation structurée, conversationelle,

Ramos, I. Informática de Base: These, Antithèse, Synthèse. Convention Informatique. 1976.

1.2.- Le Matériel: les polysystèmes et les réseaux d'ordinateurs.

L'évolution du Matériel se fait dans le sens de possibiliter:

- Systèmes sur mesure
- Systèmes orientés
- Systèmes dispersés

en étant en plus des systèmes polyvalents capables de réaliser "macrofonctions" tant par Matériel que par Logiciel et susceptibles d'être combinés entre eux pour fournir une grande variété de configurations.

Aussi on est en train d'incorporer dans le Matériel le Logiciel plus coûteux: Logiciel de base et applications standards.

Les polysystèmes pourront être composés de façon dynamique et adopter ainsi une architecture adaptée aux besoins fonctionnels de l'utilisateur ce qui conduira à des réseaux hétérogènes et changeants d'ordinateurs.

Nous pouvons voir par des tables l'évolution du Matériel en distinguant cinq générations (E.C. Joseph, "Future 4th generation computer architecture-Polysystems" UNIVAC):

1ere Génération: Automatisation des procédés scientifiques et commerciaux qui avant étaient faites à la main.

2eme. Génération: Manipulation de grands volumes de données (data processing era)

3eme. Génération: Elaboration de très grands volumes de données par ordinateur (information processing era)

4eme. Génération: Terminaux éloignés, Réseaux (information on-line era)

5eme. Génération: "Adjoints". Assistence en temps réel et "in situ" de l'homme dans beaucoup de ses fonctions (en particulier la Programmation).

Génération: 1ère.- Date: 1951-52 - Calcul: Scientifiques et Commerciaux.- Matériel: Tubes. Composants Electriques (10^4 int./sec.).- Logiciel: Assembleurs. Symboliques. Bibliothèques.- Architecture: Spécialisé.

Génération: 2ème.- Date: 1958-60.- Calcul: Traitement de données.- Matériel: Transisteurs. Portes Logiques. (10^5 int./sec.).- Logiciel: FORTRAN-COBOL. Moniteurs. Macros.- Architecture: Générale.

Génération: 3ème.- Date: 1963-65.- Calcul: Traitement de l'Information.- Matériel: Circuits integrés. Circuits Multiples (10^6 int./sec.).- Logiciel: Système d'exploitation. Multiprog. Langages.- Architecture: Générale.-

Génération: 4ème.- Date: 1970-72.- Calcul: Traitement on-line de l'information.- Matériel: MSI/LSI (chips: registers. adders) (10^7 int./sec.).- Logiciel: Langages extensibles. PROMS. S. Conversationels.- Architecture: Réseaux.

Génération: 5ème.- Date: 1976-78 (?).- Calcul: Adjoints.- Matériel: LSI/GSI (chips: macrofonctions: CPU...).- (10^7 inst./sec.).- Logiciel: Programmation descriptive. Langage Naturel.- Architecture: Agrégés.-

Dans un futur prochain nous aurons aussi une augmentation de la portabilité et des accès éloignés ainsi que des E/S "orales" ce qui possibilitera une "vraie" conversationnalité.

Mais il y a des problèmes dans le Logiciel de Base que les 10^7 inst/sec. ne pourraient résoudre d'eux mêmes.

2.- ANTITHESE

2.1.- Le Logiciel de base: Asymptote ALGOL-68

D'après C. Pair (C. Pair, "Les structures de l'Information, Alés 1971) l'informaticien a à représenter dans la mémoire d'un ordinateur des objets sur lesquels il veut effectuer des traitements et il propose le schéma suivant pour la Programmation (Sens large: Analyse + Programmation):

Objets Externes	⟶	Objets Mathematiques	⟶	Objets d'un Langage de Programmation ⟶ Objets en Memoire
Traitements	⟶	Algorithmes	⟶	Programmes du Langage ⟶ Programmes Machine

La programmation devient alors une succession de représentations d'objets et traitements sur des successives "machines virtuelles". Dans chaque niveau d'abstraction ou Machine virtuelle, il existe des objets et des traitements fournis par le "langage machine" de ce niveau.

La Programmation finit quand on dispose d'une machine réelle (par Matériel ou Logiciel) qui accepte comme langage machine celui du dernier niveau d'abstraction.

L'évolution des langages de programmation a été faite et continue à être faite dans l'esprit de fournir au programmeur, des langages chaque fois plus puissants; pensons à l'evolution:

FORTRAN ⟶ PL/I ⟶ ALGOL-68
COBOL

Mais pour "le programmeur" cette évolution n'a presque pas existé l'ALGOL-68 étant une asymptote à son activité, imposée par la complexité du même langage.

La plus grande innovation dans le Logiciel sera peut-être la diminution de son coût comme conséquence de la programmation interactive en utilisant les facilités du Matériel et du Logiciel.

2.2.- Le "Programmeur virtuel": Asymptote complexité.

L'Art de la programmation (dans la terminologie de D. Knuth) ne peut pas être encore défini comme une discipline scientifique puisque on ne dispose pas d'une mesure de son progrès, il reste ainsi un "Art" (J.T. Schwartz, "what constitutes progress in programming" ACM Forum CACM Nov. 75), bien qu'il est en train de démarrer à niveau académique des méthodes de mesure de la "structuralité" d'un programme (R. Noyes "On a mesure of program structure" Thèse pHD, Carnegie Mellon University, Nov. 73).

Autant que la complexité du problème à résoudre ne dépasse pas la capacité d'un programmeur individuel, celui-ci apparaît historiquement doué d'une série d'outils de programmation qui ont changé son premier approche naïf. Du programmeur des machines de deuxième génération au programmeur d'aujourd hui il y a eu une évolution analogue et miméthique à celle qui a expérimenté le Logiciel de base. Nous avons passé par de succesifs "programmeurs virtuels" qui disposent chaque fois de langages (organigrammes, langages d'analyse...) plus puissants pour résoudre son problème. Nous voyons ici une interdépendance très poussée (miméthique) entre l'Homme et la machine.

Mais quand la complexité du problème empêche qu'un seul programmeur soit capable de le résoudre et il faut une équipe nombreuse de programmeurs, nous nous retrouvons presque dans la situation naïve du programmeur nu. Ce qui a conduit, dans quelques cas, à l'abandon des grands systèmes.

La programmation comme activité communitaire se trouve dans sa deuxième génération grâce à la Programmation Structurée, le "programmer's team"... qui configurent des "equipes virtuelles" de programmation.

Pour l'instant la complexité des grands systèmes reste une asymptote pour les equipes (réelles ou virtuelles) de programmation.

3.- SYNTHESE

Nous avons vu que l'évolution du Logiciel de base et du rôle du programmeur (individuel ou collectif) se réalise dans le sens d'une plus grande complexité (objets et traitements plus puissants) des langages (en général non conversationnels) qu'ils usent, ce qui diffère totalement de la situation historique et des solutions apportées par des disciplines très prôches comme les Mathématiques et la Linguistique:

- Simplification et efficacité (plutôt qu'élégance) des langues (seulement quelques constructions abstraites)
- Souligner le Style de Programmation (comme une technologie) et non les langages (Littérature informatique)
- Conversationnalité réelle
- Evolution des langages actuels plutôt qu'introduction de nouveaux
- Programmation floue (comme flou est le discours)

Ce qui a conduit aux échecs des grands systèmes du Logiciel de base et de la programmation qu'ils supportaient.

Mais il parait que le nombre des échecs n'a pas été encore suffisant ou que les investissements ne se sont pas encore assez amortis, pour changer la tendance de ces deux disciplines, bien qu'une possible solution se voit dans certains travaux. Déjà en 1967 R.W. Floyd ("Assigning Meaning to programs" Proc. Symposia in Applied Mathematics, vol. 19. American mathematical Society) mais surtout en 1971 ("Toward interactive design

of correct programs" IFIP Congress 1971) a posé la base (coïncidant avec les réflexions historiques analogyques de Peter Naur CACM Nov 1975) de la programmation comme résultat d une conversation réelle entre la machine et le programmeur en utilisant un langage non nécessairement borné dont la machine est capable même (comme il le démontre) de poser des contrexemples

Ces travaux de Floyd qui ont été utilisés pour la preuve de programmes (méthode des assertions) n'ont pas pourtant modifié les deux tendances anti-historiques-analogiques du Logiciel de base en les portant vers ce que nous avons nommé Programmation Floue, c'est à dire vers une programmation heuristique et interactive avec des langages simples

Nous croyons que ces nouvelles tendances dans le Logiciel de base et dans l'Art de la programmation qui se dégagent de la vision globale du système homme-machine et non plus de l interdépendance chaque fois plus poussée entre les deux, avec la tendance du Matériel (Polysystèmes et/Réseaux d'Ordinateurs) qui approchera encore plus l'ordinateur à l'homme, est la solution au Logiciel de base dans un futur pas très lointain

RELOCATING DIRECT LINKING LOADER FOR MCS-80 MICROCOMPUTER SERIES

J.R. Garitagoitia I. Ramos

Departamento de Automática e informatica
Facultad de Ciencias - Universidad de Bilbao
Bilbao (SPain)

Abstract

A powerful relocating direct linking loader has been implemented for the INTEL'S MCS-80 series microcomputers.

Key words: Loader , relocatable, MCS-80

Introduction

As a first project in the developement of a programing system oriented to INTEL'S MCS-80 microcomputer, a relocating direct linking loader has been designed and implemented. This Kind of loader has convenient features with not too severe storage requirements.

The actions performed by the loader are:

- storage allocation
- linking
- computing of first and second order relocatable addresses (external symbols)
- loading

The loader accepts as input, relocatable and absolute segments in any order and number and loads them into core.

The loader is operating in our laboratory on a MCS-80 16 k memory microcomputer since June 1976.

Loader Type

Among the different kinds of loaders [1] we have chosen the relocating direct linking approach because the storage requirements are not too restrictive and the performances offered are satisfactory enough for our purpose. A dinamic loader will take too much storage in our microprocessor context.

The loader accepts as input relocatable and absolute segments.

The possible overlapping of segments is detected by a message of error.

Two types of relocatable addresses are considered:

- addresses that reference words in the current segment (first order relocatable addresses)
- addresses that reference words in other segments (second order relocatable addresses or external symbols)

The second kind of relocatable addresses imposes a splitting of the loader into two steps: the first one to form a dictionary (table) of the external symbols and the second one to use it.

The loader outputs a warning message if some external references are not satisfied, and the user can load the missing segments or end the loading process and begin the execution if desired (This feature is useful in testing incomplete programs).

The loader optimizes the storage taken by the object program if the segments are of relocatable type, but if the segments are absolute ones this is not its concern.

It has been our purpose to minimize the amount of storage taken by STEP-2 of the loader because it is in core at the same time than the program loaded. In this aim it performs strictly the actions that Step 1 of the loader cannot perform (basically corputing of second order relocatable addesses).

The implementation has been done in the INTEL'S assembly language [2], [3] in a modular way, but we have not used the closed subprogram structure to avoid the long pameter lists and the time consuming prologues. All the objects (variables, vectors...) of the loader program are global to every subroutine Only the 1/0 subroutines and also a hexadecimal to binary conversion routine have been used, but in both cases the parameter list is minimal.

In the present implementation, the loader accepts as input punched paper tape (other inputs: disks files... wil be soon disponible and produces an intermediate code on disk after step-1 is performed, this intermediate code is input (with other information, vectors, tables ...) to step-2.
The overall process is controlled from a teletype. Wenced then this minimal peripheral configuration.

The loader reeds a certain information to perform its work and this information is given by the trnslators (assemblers...). This information is:

- segment length
- entry points list and the order in which they appear in the segment
- external symbols list (second order relocatable addresses).
- The object code with the relative addre

Garitagoitia, J.R., Ramos, I. Relocating Direct Linking Loader for MCS-8 Microcomputer Series. Proc. of COMPCON 77. 1977.

-sses assigned by the translator and the first and second order relocatable addresses flagged.

The loader actions are acccmplished in two steps in the following way:

Step-1 Performs the following actions

-Memcry allocation for the segments. In this way the starting loading addresses for each one is known.

-first order addresses relocation

Step-2 Performs the following actions:

- Linking between segments (second order relocatable addresses definition)
- Physical loading of the segments into core and exscution initialization.

To do these actions, step-1 needs the following data:

a) - Segment length or its equivalent (starting and ending addresses)

b) - Starting address for the loading of the first segment (In this case it is fixed to 20H but is easily modifisiable by the user).

c) - Track the relative location of each memory allocation instruction.

d) - Object text.

e) - Information abaut the first order relocatable addresses (it is necessary to mark them).

f) - Table with entry points external representations (identifiers) with their relative addresses in the segment.

g) - Table with the external points external representations with their relative addresses in the segment.

The data a, b, c, d, are necessary to perform the core allocation.

To perform the relocation of the first order relocatable addresses the starting address and data e) are necessary.

The data f) and g) are necessary for step-2 and are elaborated in step-1

The necessary data for step-2 are:

h) - Begin of execution addresse. This address is generated by step-1 using the data f) of the first segment loaded.

i) - Information about the second relocatable addresses (absolute value) and the places wherw they appear in the segment.

j) - Segment starting address.

k) - Object text of the segment to load.

The data i) are used to perform the chaining between segments and j), k) for the physical loading of the segment into core.

Datum h) allows the control transfer to the object program.

This is the minimal information needed by the loader to perform its task and is contained in the input segments.

The input segments will be the result of previous translation or library outputs. Every segment is built up of two parts:

- object text part.
- information text part.

All the text is formated in blocks long of 72 ASC II characters (of length) Each block begins by a ":" if it is a block containing relocatable text or by " " if it is a block containing absolute text.

Let us see the different kinds of blocks:

Machine instructions blocks (object text)

The machine instructions blocks may appear in arbitrary number.

They have the following structure:

- First character ":" or " "
- Two hexadecimal digits (block length in bytes).
- Block first byte address (four hexadecimal characters).

One byte not used by the loader (two hexadecimal digits): we keep it for future uses

- Object text bytes: is the information to be loaded after updating of the relocatable addresses.

  The first order relocatable addresses are proceded by "R"

  The second order relocatable addresses are preceded by "X"

  Each byte is represented by two hexadecimal characters.

- Longitudinal checksum control byte.

The ending block has not the object text field and the offset is the same as the first block one.

Information blocks (inforation text)

We have three different information blocks for each segment. Each block length fields: the first one is the block length in characteres and the second one depends on the kind on the block.

The blocks are:

- External information block: we have in this block the identifiers (max. 5 alphanumerical characters) of the segment external references separated by commas.

- track information block: formed by pairs of hexadecimal digits separated by commas. Each pair is a pointer to the corresponding identifier of the external information block and shows the order in which these symbols appear in the program.
- entry points information block: formed by pairs:(identifier,offset) separated by commas and where identifier is the entry point external representation and offset is the entry point offset(four hexadecimal digits) in the segment.

The end of each block is marked by a ";"

We can afford in a precise way the structure of this segments by giving a type-2 grammar in BNF (we give only the rules; the terminal symbols are underlined and the non-terminal ones are written between parentheses. The axiom is the left part of the first rule)

We have:

(segmento)::=(cadenas maquina) (cadenas de informacion)

(cadenas maquina::=(cadena maquina)CR/LF (resto cadenas maquina)

(resto cadenas maquina)::=(cadena maquina) CR/LF(resto cadenas maquina)/(cadena maquina ultima)CR/LF

(cadena maquina)::=(start)(numero de bytes) (direccion)(bytes de control)(bytes maquina)(control de cadena)

(cadena maquina ultima)::=(start)(numero de bytes)(direccion)(bytes de control)(control de cadena)

(start)::=:/_

(numero de bytes)::=(par de d.h.)

(direccion)::=(par de d.h.)(par de d.h.)

(bytes de control)::=par de d.h.)

(control de cadena)::=(par de d.h.)

(par de d.h.)::=(d.h.)(d.h.)

(d.h.)::= 0/1/2/3/4/5/6/7/8/9/A/B/C/D/E/F

(bytes maquina)::=(byte)(bytes maquina)/(s. reubicable)(bytes maquina)/_

(byte)::=(par de d.h.)

(s. rebicable)::=R/X

(cadenas de informacion)::=(cadena ext)CR/LF (cadena o.p.)CR/LF (cadena int)CR/LF

(cadena ext)::=(start)(par de d.h.):(cuerpo cadena ext);

(cadena int.)::=(start)(par de d.h.):(cuerpo cadena int);

(cadena o.p.)::=(start)(par de d.h.):(cuerpo c. o.p.);

(cuerpo c. ext)::=(etiqueta),(cuerpo c. ext)/(etiqueta)/_

(cuerpo c. o.p.)::=(par de d.h.),(cuerpo c. o. p.)/(par de d.h.)/_

(cuerpo c. int)::=(etiqueta),(direccion),(cuerpo c.int)/(etiqueta),(direccion)/_

(etiqueta)::=(letra')(resto etiqueta)

(resto etiqueta)::=(letra)(resto etiqueta)/(d.d.)(resto etiqueta)/_

(letra')::=a/?/(letra)

(letra)::=A/B/C/... ./Z

(d.d.)::=0/1/2/3/4/5/6/7/8/9

Implementation

The loader performs two passes because of the second order relocatable addresses, and we call PASS1 and PASS" these two program segments. We have another program segment that comunicates with the operator and controls the operation, it includes as a subsegment the PASS1 and we call it MP(main program). This program reduces also the amount of work to be done by STEP-2. A flowchart of MP is shown in fig 1, in the next page.
The variable ITI indicates the kind of block we have just elaborated according to the code besides:

ITI=0 No block has been elaborated

=1 The external information block has been elaborated

=2 The external track information block have been elaborated

=3 We have elaborated all the blocks.

The PASS-1 program is formed by two parts:

Object text blocks processing

The actions performed are:

1)Input information checksum

2)Storage allocation to the segment(it depends on the segment type)

3)First order relocatable addresses compilation.

4)Counting of the number of second order relocatable addresses. (This allows a matching between the track information block and the object text block).

5)Fields processing: number of bytes in the block, offset and control byte.

6)Building of a table with the absolute segments starting address(to detect overlaping)

7)Storage of the intermediate code in disk.

8)Output the starting and ending addresse of the core segment taken.

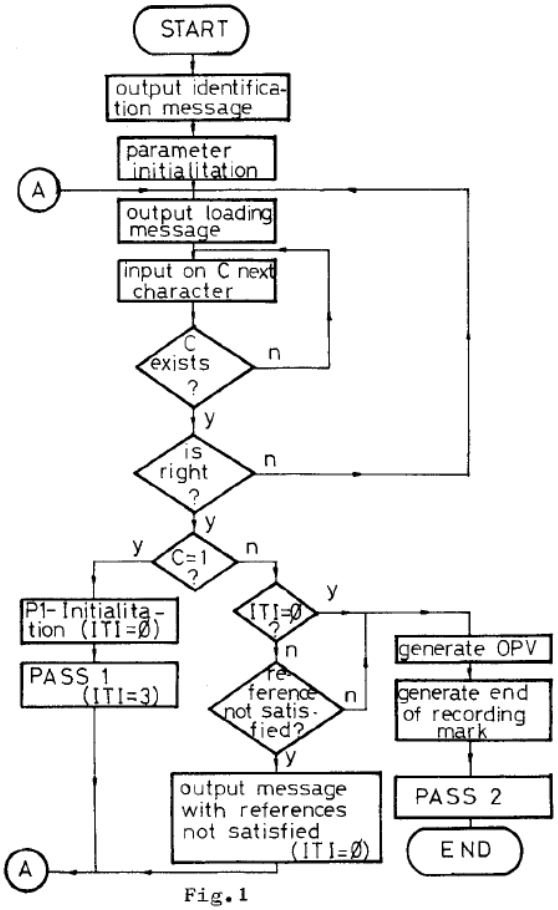


Fig.1

Information blocks processing.

We have three differents kinds of information blocks and therefore we have three different processing segments.

-External information block processing.The actions performed are:

1)Input information checksum(counting of the number of characters)

2)External symbol processing.

3)Building of an auxiliar vector MAX with each external identifier preceded by its length.

-Track information block processing.

1)Input information checksum(counting of the number of characters)

2)Hexadecimal digit pairs processing: this numbers are pointers to the auxiliar vector MAX, the external symbols are obtained and with these symbols as keys a table-look-up is made.

-Entry points information block processing

1)Input information checksum

2)Processing of each pair:(identifier,offset) computing of the **absolute** address of the symbol(relative address plus offset).

3)Symbol processing (counting of the number of characters)

4)**Symbol table look-up** (the symbol is entered into the symbol table and an error message is producced if the symbol was already in the table.

5)Building of the memory map.

The first entry point of the first segment will be taken as the transfer address to begin the execution.

The loader starts at the address 1500 H and occupies about 4K of storage.

The address 10 H is the starting address, and following this address we have the vectors,tables... until address 1500 H.

Storage configuration in PASS-1 execution(fg2)

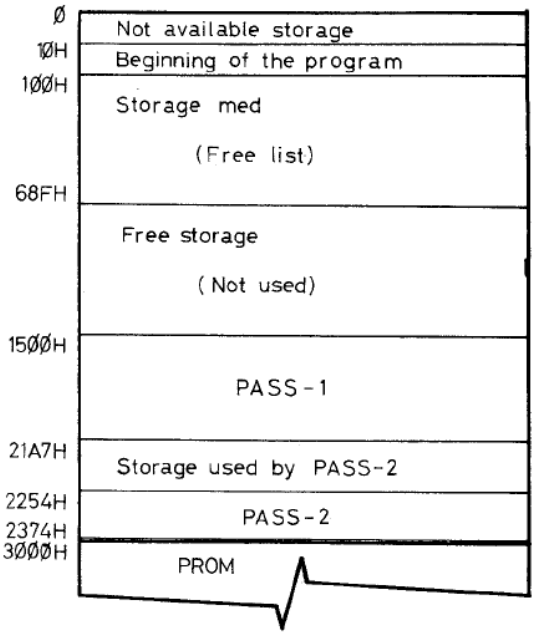


fig. 2

The low order addresses are used as variables by the monitor, FDOS. The first free location is at address 20 H.

The free list is used for variables representation,buffers and tables of the loader.

The principal objets in the free list are:
- output messages vector
- input buffer
- external auxiliar vector:MAX
- track information vector:OP
- vector with the second order relocatable addresses values (OPV)
- two tables with absolute blocks starting

and ending addresses(to check overlapping)
- disk I/O buffer.

All this vectors have a fixed lenght and are managed by pointres in the usual way.
- symbol table: used to store, in loader's PASS-1 the external representation(identifiers) of the second order relocatable addresses. It is a Hash table with chained overflow lists.

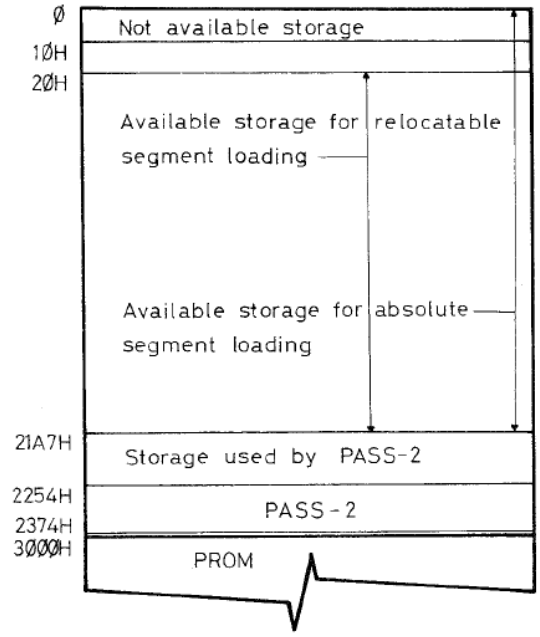


fig. 3

Storage configuration in PASS-2 execution(fg3)

Example: Supose the information text is

External information block :0D:ETI,COMA,DF;
Track information block :0D:01,CA,01,05;
Entry point information block
:12:ENT,0000,DF,010A;
The loader performs the following steps:

1.External information block processing

The loader reads in the first block on the input buffer (vector BUF) and generates the auxiliar vector MAX in the following way:

MAX

3 E T I 4 C O M A 2 D F

2.Track information block processing

The loader inputs the second information block into BUF and stores the symbols in the symbol table. After the processing we will have the configuration shown in the figure 4

3.Entry points information block processing

The "values" of the symbols in the symbol table are defined and we have the configuration shown in figure 5

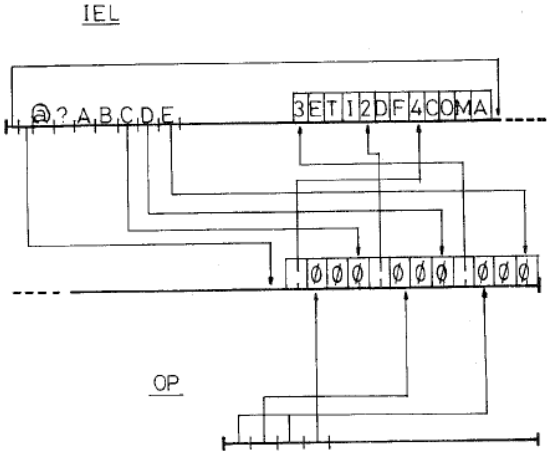


Figure 4

4.Generation of the vector OPV

At the end of the loading PASS-1, the vector OPV is generated in the following way

OPV

00 00 0A 02 00 00 00 C0

5.Object code elaboration

The object code text is elaborated and stored in disk as an intermediate code.

the intermediate code format is the following: we have fixed length blocks(50 characters), the first character of a block is:

- A colon ":" indicates that the second and third bytes are the starting loading address of the following segment of bytes.
- An hexadecimal character: is the beginning of an object text byte to be loaded in the corresponding sequential addresses(after the preceding one) and the same for the following bytes.
- A period marks the end of the blocks to load(end of program).

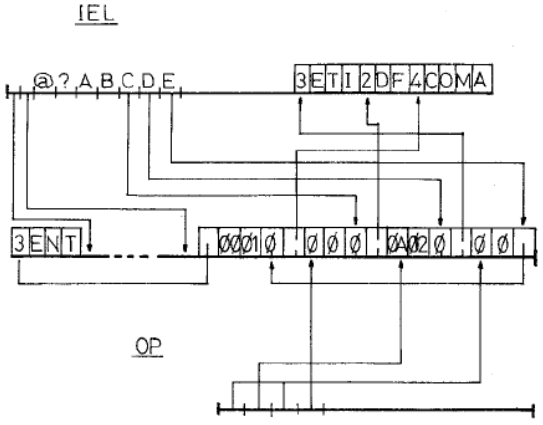


Figure 5

Every blok ends by a semicolon ";"

At the end of PASS-1 the first order reloca tableaddresses have been computed and we ha ve only to compute the second order relocata ble addresses (preceded by a X in the object code text).

PASS-2

The loader PASS-2 is in the high order addre sses portion oh the memory. It accepts as in put.

- The intermediate ccde in disk
- Vectors and tables in storage (OPV...)

The actions perfromed are:

- Input of an intermediate text block
- Conversion from hexadecimal ASCII charac ters to binary.
- Reading in the load starting address (after the ";").
- Computing the seccnd order reubicable ad dresses (The two irrelevant bytes after each X are changed by the corresponding value in OPV).
- Physical loading
- Control transfer to the loaded program (a transfer instruction is generated and executed in the first words of storage)
- Output of an "end of load" message

A flow-chart os PASS-2 is (shownin figure 7)

A complete set of error messages is furnished by the loader.

PASS-1 detects

- Errors in the input segments (mispun- ching...)
- Programing errors (multiple defined sym bols...)
- Loading errors (overlapping, memory over flow...)

Overflows in the different vectors, tables.

MP detects:

- A warning error is written if there are external symblos used but not defined.

An example is given in the Apendix.

4.- Conclusion

1st A direct linking relocatable loader for MCS-80 microcomputer series has been im plemented that allows independent transla tion of the input segments, liberating the programmer of address adjustement.

2nd The loader implemented accepts relocata ble and absolute text indifferently.

3rd We may have and arbitrary number of seg ments of data/ instructions with all po ssible cross-referencing.

4th The storage used has been minimized: all

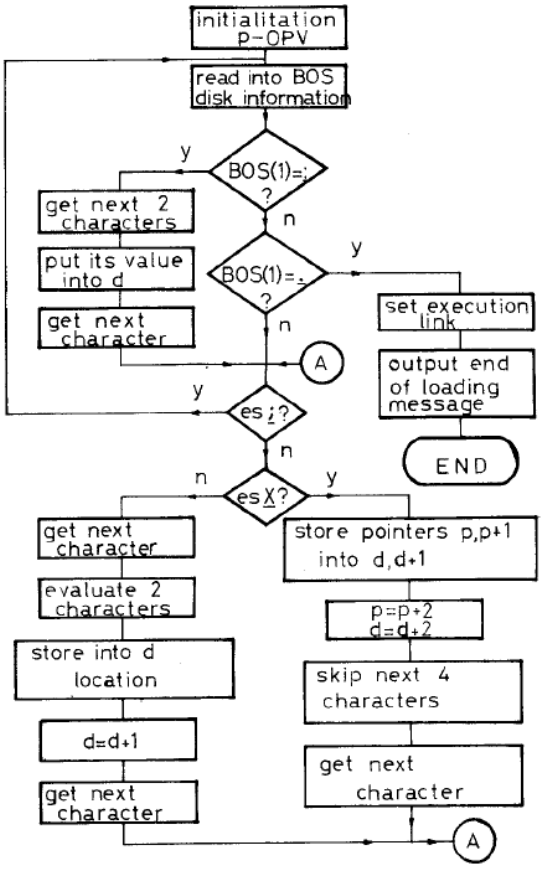


Fig.6

the possible processing is performed in PASS-1, liberating PASS-2 of innecessary ac tions (because PASS-2 is into ccre during the physical loading process).

5th This loader is a powerful tool allowing the implementation of high level progra mming languages, relocatable libraries...

5.- References

1.- DONAVAN J.J., "Systems Programming" Mc Graw-Hill 1972

2.- INTEL 8080 Manual: "Assembly language Programmirg manual"

3.- INTEL 8080 Manual: "8/MOD 80 Operators manual".

"SISTEMA EDITOR/PERFORADOR DE TEXTOS"

Por:
I. Ramos

1. INTRODUCCION

Existen varios tipos de sistemas editores de textos. En particular señalaremos:

- los editores orientados a líneas
- los editores orientados a cadenas de caracteres

El editor que presentamos en nuestro trabajo es del primer tipo: orientado a línea, pero lo hemos dotado de unas características poco usuales, que son, entre los editores orientados a líneas:

- El orden (con relación al número de línea) de introducción de las correcciones (PROGRAMA EDITOR) es totalmente arbitrario; lo que permite equivocarse en la introducción de éstas. Esta es una facilidad no usual en los editores corrientes, lo que los hace poco flexibles.

- El sistema Editor de textos puede ser usado para perforar programas bajo su control, es decir, permite introducir información por el dispositivo de entrada (en nuestro caso un teletipo) que se almacena en un soporte intermedio (cinta magnética) para ser volcada al final sobre el soporte definitivo (cinta de papel perforada), siendo actualizada la información de acuerdo con las correcciones introducidas (al mismo tiempo que el texto y en un orden cualquiera). Esta facilidad es particularmente necesaria cuando el dispositivo de entrada de información al ordenador es secuencial (cinta de papel perforada) ya que hace innecesario el repetir todo el texto por un solo error, problema que en los soportes de información discontinuos (tarjeta perforada) es trivial de resolver (cambiar una o varias tarjetas), no siéndolo así en los continuos.

El motivo de introducir estas facilidades ha sido la necesidad de perforar y corregir grandes volúmenes de información disponiendo de un editor "clásico". Obviamente, este Editor tiene sentido en instalaciones que no dispongan de dispositivos periféricos de acceso directo (discos, tambores).

Se han realizado varias implementaciones de este editor como "proyectos de curso" de la asignatura Informática Aplicada sobre el miniordenador HP-2116B del Departamento de Automática e Informática de la Facultad de Ciencias de la Universidad de Bilbao.

2. LENGUAJE DE CONTROL DEL EDITOR PERFORADOR

El Editor/Perforador acepta un TEXTO A EDITAR sobre un soporte físico (cinta de papel: /P, o cinta magnética: /M) y produce un TEXTO EDITADO sobre un soporte (Teletipo: /T, o cinta de papel) de acuerdo con un PROGRAMA DE EDICION (introducido por Teletipo o cinta de papel).

La elección de los soportes viene dada por nuestra configuración particular.

Observamos que puede ser usado como programa de utilidad: listado, perforación ... de cintas magnéticas, por ejemplo.

Las instrucciones que se usan son las siguientes:

1) /I,1

Insertar una o más líneas después de la línea 1. (Las líneas a insertar se dan a continuación)

2) /D, 1 6

3) /D, l_1, l_2 ... Borrar las líneas

Ramos, I. Sistema Editor / Perforador de textos. Revista de Automática e Informática. Volumen: nº 31. 1977.

comprendidas entre las líneas l_1 y l_2, ambas inclusive.

4) /E, l δ

5) /E, l_1, l_2

Reemplazar la línea l ó las líneas l_1 a l_2 por las líneas que aparecen a continuación.

6) / C l, l, C

Insertar en la línea l a partir del "$C^{ésimo}$ carácter los caracteres que se dan en la línea siguiente:

7) / C D, l, C u

8) / C D, l, C_1, C_2

Suprimir de la línea l el carácter "$C^{ésimo}$ o los caracteres comprendidos entre el "$C_1^{ésimo}$ y el "$C_2^{ésimo}$

9) / C E, l, C δ

10) / C E, l, C_1, C_2

Reemplazar en la línea el carácter "$C^{ésimo}$ por los caracteres que se dan a continuación o reemplazar entre el carácter "$C_1^{ésimo}$ y el "$C_2^{ésimo}$, ambos inclusive, por los caracteres que se dan a continuación:

11) / I

Listar señalizando.

12) / ↑

Suprimir la línea anteriormente escrita en el dispositivo de entrada.

13) / C

Continuar el proceso después de finalizada una edición.

14) / E

Marca el fin del PROGRAMA EDITOR.

15) / X

Indica que las líneas que vienen a continuación (hasta la próxima instrucción de EDITOR) son líneas de "TEXTO A EDITAR" que estamos escribiendo bajo control del Editor/Perforador y que se almacenan en cinta magnética.

NOTA: El número de orden de la instrucción es su código numérico de operación. Una gramática Tipo 3 del lenguaje es la siguiente: (damos las reglas)

2) ⟨instrucción⟩ ::= /⟨resto⟩

3) ⟨resto⟩ ::= I⟨n1⟩/1/D⟨n0⟩/2/E⟨n5⟩/3/C⟨n6⟩/4/L/5/↑/6/F/41/X/42/

4) ⟨n1⟩ ::= $_{\perp}$⟨n2I⟩/7

5) ⟨n2I⟩ ::= *int*/8

6) ⟨n0⟩ ::= $_{\perp}$⟨n2D⟩/9

7) ⟨n2D⟩ ::= *int*⟨n3D⟩/10

8) ⟨n3D⟩ ::= ϕ/12/$_{\perp}$⟨n4D⟩/11

9) ⟨n4D⟩ ::= *int*/13

10) ⟨n5⟩ ::= $_{\perp}$⟨n2R⟩/14

11) ⟨n2R⟩ ::= *int*⟨n3R⟩/15

12) ⟨n3R⟩ ::= ϕ/17/$_{\perp}$⟨n4R⟩/16

13) ⟨n4R⟩ ::= *int*/18

14) ⟨n6⟩ ::= I⟨n6I⟩/19/D⟨n6D⟩/20/R⟨n6R⟩/21/ϕ/22

15) ⟨n6I⟩ ::= $_{\perp}$⟨n2C1⟩/23

16) ⟨n2C1⟩ ::= *int*⟨n3C1⟩/24

17) ⟨n3C1⟩ ::= $_{\perp}$⟨n4C1⟩/25

18) ⟨n4C1⟩ ::= *int*/26

19) ⟨n6D⟩ ::= $_{\perp}$⟨n2CD⟩/27

20) ⟨n2CD⟩ ::= *int*⟨n3CD⟩/28

21) ⟨n3CD⟩ ::= $_{\perp}$⟨n4CD⟩/29

22) ⟨n4CD⟩ ::= *int*⟨n5CD⟩/30

23) ⟨n5CD⟩ ::= ϕ/32/$_{\perp}$⟨n6CD⟩/31

24) ⟨n6CD⟩ ::= *int*/33

25) ⟨n6R⟩ ::= $_{\perp}$⟨n2CR⟩/34

26) ⟨n2CR⟩ ::= *int*⟨n3CR⟩/35

27) ⟨n3CR⟩ ::= $_{\perp}$⟨n4CR⟩/36

28) ⟨n4CR⟩ ::= *int*⟨n5CR⟩/37

29) ⟨n5CR⟩ ::= ϕ/39/$_{\perp}$⟨n6CR⟩/38

30) ⟨n6CR⟩ ::= *int*/40

NOTAS: 1. Cada regla aparece numerada con el número del estado del Autómata Asociado (punto 3.1).

2. Por ϕ representamos la cadena vacía y por *int* un número entero sin signo.

3. Hemos numerado las reglas de la gramática de acuerdo con las acciones semánticas que realiza el autómata reconocedor asociado que veremos en el punto 3.

3. EDITOR PERFORADOR

El proceso de edición/perforación se ha dividido secuencialmente en las siguientes etapas:

- Reconocimiento del texto fuente y generación del texto intermedio

(PROGRAMA-1).

- Clasificación del Programa editor intermedio.
- Perforación y edición del texto (PROGRAMA-2).

Veamos estos pasos con más detalle:

3.1.- RECONOCIMIENTO DEL TEXTO FUENTE Y GENERACION DEL TEXTO INTERMEDIO

Damos el esquema de este programa en forma del diagrama de transición entre estados del autómata reconocedor y constructor (de acuerdo con la Gramática del lenguaje dado en el punto 2) del texto intermedio. Las acciones asociadas con las transiciones del autómata (Programación conducida por los datos) son triviales, teniendo en cuenta el texto intermedio generado (punto 3.2).

Este Programa-1 lee todo el texto fuente -instrucciones y texto a editar- grabando las primeras en memoria en forma de una lista doblemente ligada y el texto a editar secuencialmente en cinta magnética.

NOTAS: Hemos añadido el diálogo inicial: Transición 1→2 y final 31→32 y 32→33 de definición de dispositivos de E/S.

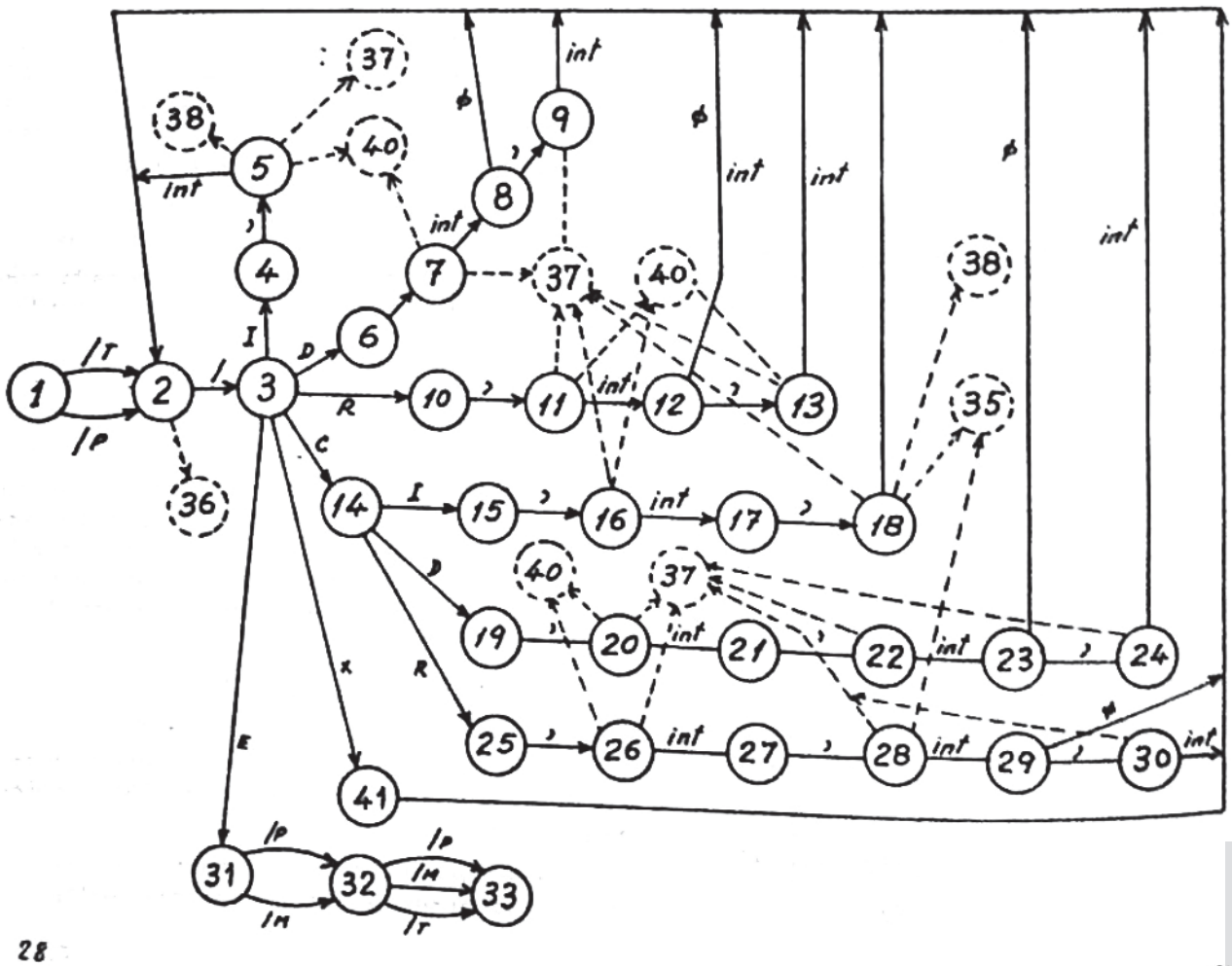
- Se han señalado algunos estados de error: 35, 36, --- 40. El estado de error 34 da cuenta de todos los otros errores.
- Las acciones asociadas son obvias dado el código intermedio a generar (3.2).
- El estado 33 es estado inicial del diagrama de transición del PROGRAMA-2.

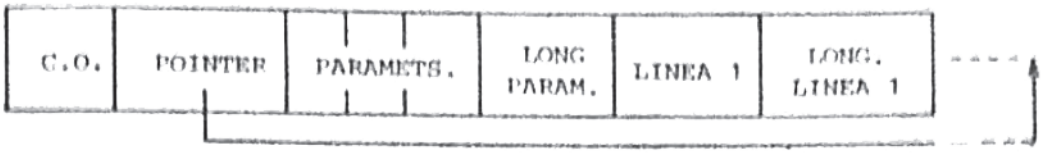
ESTADOS Y MENSAJES DE ERROR POSIBLES

NUMERO ESTADO	MENSAJE ERROR	ESTADO SIGUIENTE
34	"**CS ERR**"	2
35	"**CHAR OVERFLOW**"	2
36	"**EDIT OVERFLOW**"	1
37	"**ILLEGAL VALUE**"	2
38	"**INSERT ERROR**"	2
39	"**REPLACE ERROR**"	2
40	"**SEQUENCE ERROR**"	2 (EDITOR CLASICO)

3.2.- FORMA INTERMEDIA DEL PROGRAMA FUENTE

El PROGRAMA-I crea en memoria una versión del programa fuente, de acuerdo con el esquema siguiente:





siendo:

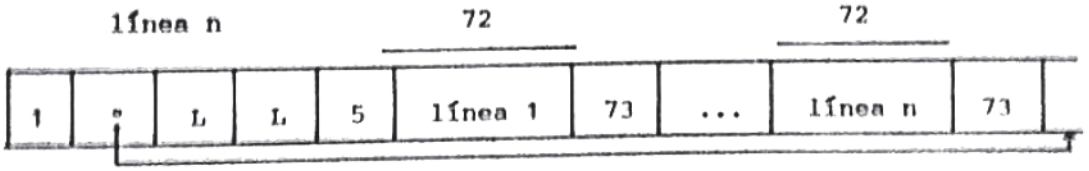
- C.O.: Código numérico de operación
- POINTER: Apuntador a la instrucción de editor siguiente, ya que son de talla variable.
- PARAMTS.: Parámetros de líneas o caracteres de las instrucciones.
- LONG.PARAM.: Número de palabras tomadas por el C.O., POINTER.
- PARAM.: (es variable según la instrucción).
  Este campo, así como LONG.LINEA i, permite recorrer de izquierda a derecha y de derecha a izquierda la instrucción de editor línea a línea, (lista doblemente ligada) lo que es necesario debido a la instrucción de borrado / t. En algunas implementaciones LONG.LINEA i se ha tomado constante e igual a 72 caracteres (máximo permitido por línea).
- LONG.LINEA i: Longitud en palabras tomada por la línea i (en algunas implementaciones suprimida si se toma longitud de línea fija).
- LINEA i: Texto de la línea i.

Las líneas del texto a editar (Modo / X) pasan directamente a cinta magnética: /M.

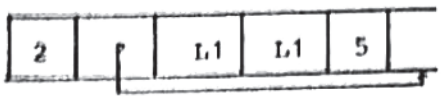
Veamos, para cada instrucción, un ejemplo de texto intermedio.

CODIGO INTERMEDIO GENERADO

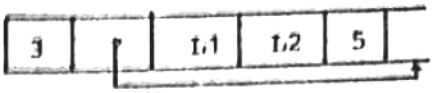
/I,L
línea 1
.....
línea n



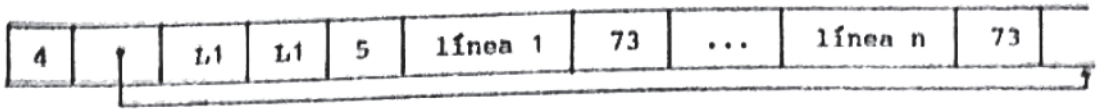
/D,L1



/D,L1,L2



/R,L1
Línea 1
....
Línea n

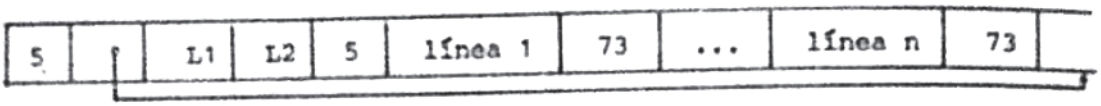


/R,L1,L2

línea 1

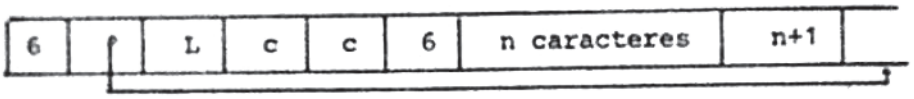
...

línea n

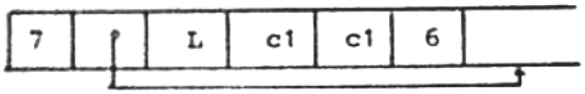


/CI,L,c

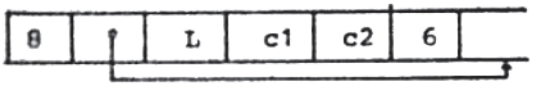
n caracteres



/CD,L,c1

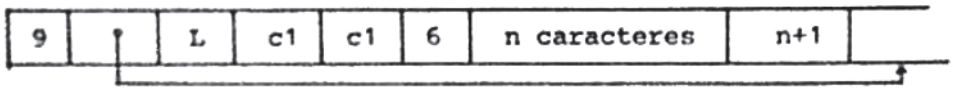


/CD,L,c1,c2



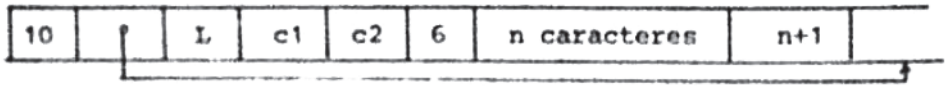
/CR,L,c1

n caracteres



/CR,L,c1,c2

n caracteres



La instrucción /* que permite borrar la línea inmediatamente anterior, plantea un pequeño problema ya que ésta, la línea, puede estar almacenada en memoria (si es una instrucción) o en cinta magnética (texto a editar).

Para saber de donde debemos suprimir la última línea se dispone de una pila en la que se almacenan:

- Donde ha sido almacenada la última línea.
- Informaciones pertinentes: C.O.

La capacidad de la pila ha sido limitada a 5 (o sea, podemos usar 5 veces seguidas la instrucción de borrado /*), teniendo la particularidad de que todo "overflow" significa la pérdida de la información asociada a la línea más profunda (la 5ª) y la entrada en la pila de la información de la última línea. Esto se consigue fácilmente usando una representación continua de la pila (5 palabras consecutivas) y dos variables puntero que marcan el final de la pila. Su actualización se hace siempre módulo 5.

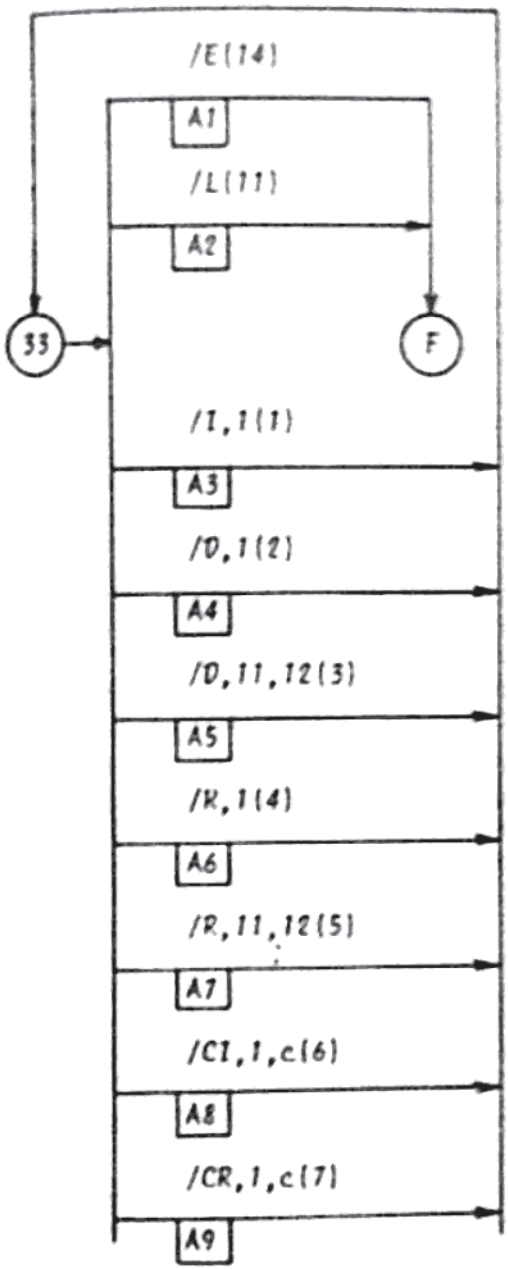
3.3.- PERFORACION Y EDICION DEL TEXTO

El PROGRAMA-2, toma de memoria la forma intermedia del programa fuente, que previamente ha sido clasificada por orden de línea creciente (lo que permite la facilidad primera anunciada en el punto I) y de acuerdo con esta forma intermedia, mediante un proceso de programación conducida por los datos, (en este caso, las instrucciones en memoria) vierte el texto a editar sobre el soporte periférico elegido, realizando las modifica-

nes oportunas.

En esquema el diagrama de transición entre estados es el siguiente:

DIAGRAMA DE TRANSICION ENTRE ESTADOS



NOTAS: - Ai son las acciones asociadas

- Los números entre corchetes se corresponden con los códigos de operación (por mayor claridad se ha añadido el código mnemotécnico).

A modo indicativo, algunas acciones son:

A1. Mientras que ¬ fin de cinta hacer { leer; NLIN ← NLIN + 1; perforar; }

A2. Mientras que ¬ fin de cinta hacer { leer; NLIN ← NLIN + 1; imprimir; }

A3.

A3.1. Mientras que NLIN≤1 hacer { leer: %(I% ← %(I% + 1; perforar; }

A3.2 Mientras que ∃ texto edición hacer { "leer" línea edición; perforar; }

A4.

A4.1. Mientras que %LIN% < 7 hacer { leer: %LIN% ← %LIN% +1; perforar; }

A4.2. leer: %LIN% ← %LIN% + 1;

A5.

A.5.1. Mientras que %LIN% < 11 hacer { leer: %LIN% ← %LIN% + 1; perforar; }

A.5.2. Mientras que 11 ≤ %LIN% ≤ 12 hacer { leer: %LIN% ← %LIN% + 1; fin. }

NOTA: Siendo %LIN el número de línea corriente.

5. CONCLUSIONES

Se han realizado distintas implementaciones de este sistema Editor/Perforador de textos, que varían sobre todo en la forma de representar (lista contigua o ligada) el programa intermedio y también en los algoritmos de clasificación del mismo por orden creciente de línea, debido a que el PROGRAMA-2 actúa sobre un soporte secuencial: la cinta magnética.

Creemos que el precio pagado, debido a las facilidades introducidas frente a los editores clásicos: clasificación intermedia del programa del editor, pila, mayor información en el programa intermedio, no es demasiado elevado: sobre todo en los sistemas que usan cinta de papel perforada como medio de entrada.

Todas las implementaciones se han realizado en el FORTRAN de la HP-2116 B permitiendo una fácil transportabilidad.

LPE

Un lenguaje en español para la programación estructurada

Por ROBERTO MOYA
Departamento de Educación Univac-Madrid

ISIDRO RAMOS
Universidad de Bilbao

RESUMEN

Este artículo presenta la parte algorítmica de un lenguaje en español que ha sido diseñado e implementado para la PE y que incorpora los mecanismos de descomposición de los sistemas complejos en módulos elementales.

La traducción de los programas escritos en LPE se realiza mediante un metaprograma que hemos implementado en un Univac 1100. La entrada a este traductor está constituida por las descripciones estructural y semántica del programa objeto —las especificaciones de control no son requeridas— junto con la descripción de los objetos a ser manipulados por él.

El programa objeto del metaprograma se obtiene escrito en ALGOL, COBOL o FORTRAN, dependiendo de un parámetro de entrada.

1. REVISION DE LA PE

El término «programación estructurada» (PE) es debido a Dijkstra, el cual lo utilizó por primera vez en una serie de notas que circularon informalmente. Estas notas fueron posteriormente ampliadas y publicadas como parte de un libro del mismo autor [1].

Si bien en el momento actual no existe todavía una definición concreta del concepto PE, su objetivo es el aportar posibles soluciones a los problemas que se han planteado en el desarrollo del software.

Estos problemas, referidos comúnmente como «la crisis del software», han sido discutidos extensivamente durante casi una década [2].

Los dos síntomas más claros de esta «crisis» son el costo de producir grandes sistemas y la baja calidad del producto acabado. En realidad, son dos aspectos del mismo problema, puesto que la baja productividad es parcialmente resultado del tiempo consumido en corregir errores previos.

Sintetizando en lo posible la enorme cantidad de publicaciones al respecto (cabe mencionar las citas bibliográficas de los [10] y [11] o el trabajo de recopilación [8]), en las cuales se aportaron ideas tales como:

— programación sin sentencias GO TO;
— el «chief programmer team»;
— prueba de corrección de programas, etc.,

cabe decir que la PE tiende a producir software de mejor calidad y más efectivo que el obtenido por la metodología tradicional, siendo las cualidades más deseables en los programas las siguientes:

— inexistencia de errores,
— facilidad de depuración,
— conformidad con las especificaciones,
— eficiencia de operación,
— robustez frente a posibles errores del medio en que el programa se ejecute,
— adaptabilidad,
— facilidad de conversión,
— claridad de diseño,
— claridad de documentación,

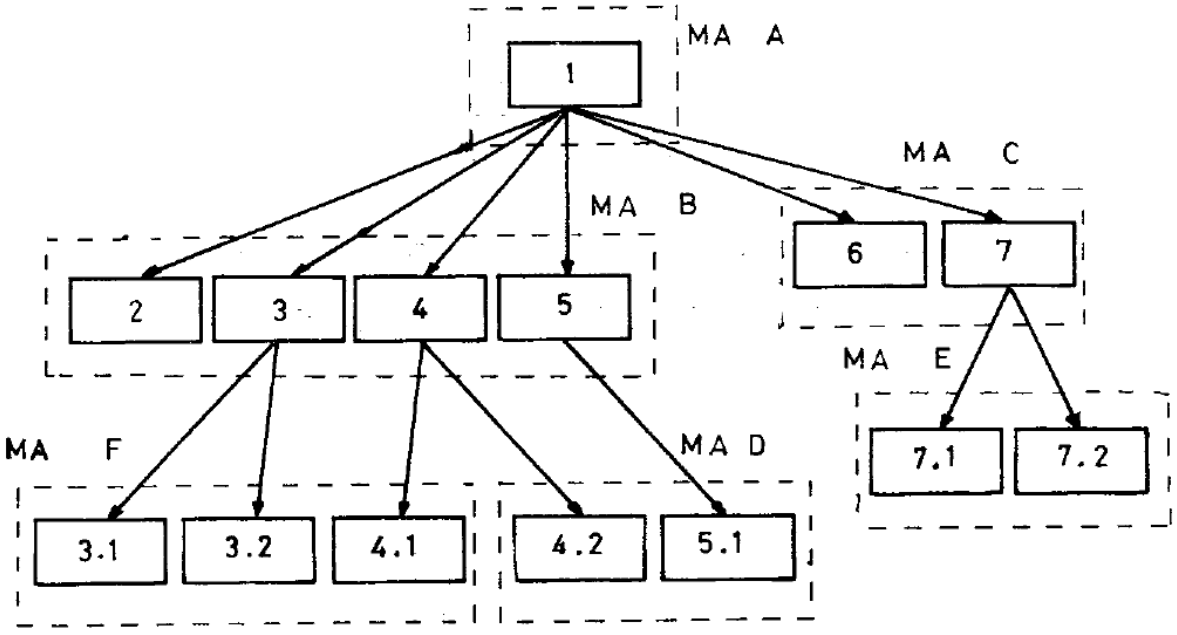
que pueden ser sintetizadas en:

— reducción de la complejidad,
— corrección de errores.

Entre las múltiples publicaciones aportando metodologías en la línea de la PE sobresalen las de:

— Dijkstra [3] y Liskov [4], aportando la descomposición por niveles de abstracción.
— Wirth [5] [6] Baker, aportando la descomposición funcional por refinamientos sucesivos.
— Jackson [7] [8], aportando la descomposición dirigida por la estructura de los datos.

Moya, R. Ramos, I. LPE: Un lenguaje en español para la programación estructurada. Proceso de Datos, no. 70, 1977.



— Naur [9], aportando la descomposición por agrupamiento de acciones.

Variantes de la línea de Jackson (Warnier [12], Bertini [13]) están siendo utilizadas en este momento por los programadores españoles, e incluso han sido adoptadas por casas comerciales de nuestro país [14].

La línea de Dijkstra es válida tanto para la programación como para el diseño; en síntesis, se basa en tomar la **abstracción** como herramienta para dividir y conquistar la complejidad de un diseño. Una abstracción es una descripción de un sistema o parte de él, sin especificar todos los detalles; es, por tanto, una simplificación conceptual, dado que proporciona una perspectiva global de las relaciones y propiedades de los componentes mayores, que sería muy difícil apreciar si todos los detalles estuviesen presentes.

Ahora bien: no podemos hablar de abstracción particular de un sistema, puesto que cualquier sistema tiene múltiples abstracciones posibles. Cada abstracción da una perspectiva diferente de lo que el sistema contiene o hace. Por ejemplo, una abstracción a_2 puede ser un **refinamiento** de otra abstracción a_1, pudiendo extenderse el proceso hasta una a_n.

Cuando las abstracciones se nidifican en esta forma, es significativo hablar de **nivel de abstracción,** es decir, del nivel de detalle en la descripción. Un bajo nivel de abstracción contiene gran cantidad de detalles. En términos informáticos, entendemos que el hardware es de nivel más bajo que el software.

Un segundo concepto, fuertemente relacionado con la abstracción, es el de **máquina virtual** o **abstracta.** Una máquina abstracta es un conjunto de operaciones primarias que describen la operación de un sistema o parte de un sistema. La máquina se denomina también virtual debido a que no necesita ser implementada en términos del hardware. Por ejemplo, la máquina COBOL se implementa generalmente por medio de un compilador.

Un gráfico que expresa estas ideas es el que encabeza esta página, donde las cajas de líneas continuas representan funciones implementadas por módulos; las líneas discontinuas, niveles de abstracción implementados mediante máquinas abstractas (MA), y las flechas, llamadas.

Cabe notar que el nivel B no es necesariamente más o menos abstracto que el nivel C; de otra parte, el hecho de separarlos significa que B y C tienen recursos diferentes; si no, constituirían un mismo nivel o MA.

Nótese que una abstracción se describe por medio de un lenguaje; por tanto, la cantidad de detalles permitidos por dicho lenguaje es un límite para la abstracción. Precisando, el nivel de detalle permitido por un lenguaje de programación impone una frontera entre lo elemental y lo compuesto; por ejemplo, un objeto a ser manipulado por un algoritmo se considerará elemental cuando su estructura, en términos de otros objetos más simples, no es relevante en la descripción del algoritmo en términos de dicho lenguaje de programación.

Las ideas de Wirth también son válidas tanto para el diseño estructurado como para la PE. Wirth propone desarrollar gradualmente el programa mediante una secuencia de pasos. En cada paso, una o varias instrucciones del programa se descomponen en instrucciones más detalladas. Esta descomposición sucesiva o refinamiento de las especificaciones termina cuando todas las instrucciones se han expresado en términos de un computador simulado (lenguaje de programación), y, por tanto, la descomposición debe guiarse por las facilidades disponibles en ese computador (o lenguaje). Obsérvese que desde los primeros estadios del proceso hay que elegir un lenguaje de programación, pudiendo aparecer dicha elección como inacertada en pasos más avanzados del refinamiento.

2. INTRODUCCION AL LENGUAJE LPE

El lenguaje LPE permite la programación en términos de una secuencia jerarquizada de niveles de descomposición, por lo que los programas mantienen los distintos niveles de detalle con toda precisión, a diferencia de los lenguajes tradicionales, que permiten solamente un nivel de detalle.

Otra característica del lenguaje LPE es la de liberar al programador de la codificación del flujo de control dentro del programa, quedando el manejo del control a cargo de un módulo controlador de la ejecución [15]. Esto evita los errores debidos a una mala codificación y supone un acercamiento al campo de la codificación automática de programas.

Una de las ideas fundamentales utilizadas en el diseño del LPE es considerar la ejecución de un programa como una secuencia de bloques que denominaremos terminales, los cuales tienen una única entrada y una única salida. Cada uno de estos bloques simboliza un **tratamiento terminal** o función, que estará implementada mediante un conjunto de instrucciones de un lenguaje de programación (en nuestra implementación, ALGOL, COBOL o FORTRAN).

Con la finalidad de poder hacer explícitos todos los niveles de descomposición de una PE, el LPE permite asociar a una secuencia de bloques terminales un bloque no terminal —también de una sola entrada y de una sola salida— que los englobe, de forma que el programa completo es uno de estos bloques no terminales. Los bloques no terminales simbolizan **acciones no terminales;** esta nidificación se permite incluso recursivamente.

La flexibilidad del método radica en la posibilidad de especificar posibles alternativas para la configuración de un bloque no terminal en el momento de la ejecución; esto se consigue asociando a cada alternativa propuesta un predicado (CONDICION), función de datos y/o resultados, que, por tanto, se evalúa en tiempo de ejecución (dinámicamente).

Para ejemplificar estos conceptos incluimos seguidamente un fragmento de programa (omitimos la parte de datos) en LPE para solucionar el problema trivial de obtener el número de fichas con cantidad par y el número de fichas con cantidad impar de un paquete de fichas; se ha seleccionado para ello la opción COBOL (en el parámetro para especificar el tipo de código objeto deseado al traductor).

```
* TRATAMIENTOS TERMINALES.
LEER SUMPAR SUMIMP ESCRIB A1.
* TRATAMIENTOS NO TERMINALES.
PROGRA CPROG CONT.
* ESTRUCTURA DEL PROGRAMA.
PROGRA. A1 CPROG.
CPROG. LEER CONT.
CONT. ESCRIB /13/
        SUMPAR CPROG /11/
        SUMIMP CPROG /12/.
* CONDICIONES.
/11/ : IF RESTO = 0 MOVE 0 TO : ELSE MOVE
     1 TO : .
/12/ : IF RESTO = 1 MOVE 0 TO : ELSE MOVE
     1 TO : .
/13/ : IF NUMERO = FIN MOVE 0 TO : ELSE
     MOVE 1 TO : .
* TEXTO TRATAMIENTOS TERMINALES.
        A1. OPEN INPUT LECTOR.
        LEER. READ LECTOR END MOVE
              HIGH-VALUE TO NUMERO
              DIVIDE NUMERO BY 2 GIVING
              COCIEN
              REMAINDER RESTO.
        SUMPAR. ADD 1 TO PARES.
        SUMIMP. ADD 1 TO IMPARE.
        ESCRIB. CLOSE LECTOR DISPLAY
        «PARES» PARES
        «IMPARES» IMPARE UPON PRINTER.
* FIN.
```

La ESTRUCTURA DEL PROGRAMA contiene la definición de cada uno de los bloques o tratamientos no terminales, en términos de los bloques a que engloba. Estos bloques componentes, a su vez, pueden ser bloques no terminales, que engloben a otros bloques, o bloques terminales. En nuestro ejemplo, PROGRA engloba a dos bloques: el terminal A1 y el no terminal CPROG, el cual engloba, a su vez, a otros dos, de entre los cuales CONT engloba en dos de sus alternativas a CPROG; aquí tiene lugar, por tanto, una recursividad, dado que en la definición de CPROG interviene el propio CPROG a través de CONT.

El tratamiento no terminal CONT engloba tres alternativas, a cada una de las cuales se le asocia un predicado mediante el label /XX/. El módulo controlador de la ejecución tomará una de ellas en función de que su predicado asociado sea cierto (cuando no se especifica el label /XX/ para una alternativa, se asume que el valor del predicado es siempre cierto, y, por tanto, siempre se ejecutará dicha alternativa).

En tiempo de ejecución, el tratamiento terminal LEER produce, entre otros resultados, un valor para la variable RESTO, el cual servirá para evaluar los predicados (simbolizados por «:» en apartado CONDICIONES). El resultado de dicha evaluación es 0 ó 1 (CIERTO o FALSO, respectivamente), lo que implicará en el momento de ejecutar CONT que se seleccione la alternativa cuyo predicado asociado tome el valor CIERTO.

La implementación actual del LPE no permite la programación paralela, ya que solamente admite un único predicado de valor CIERTO para las distintas alternativas constituyentes de un bloque no terminal (determinismo).

La ejecución del anterior programa en LPE generará la secuencia de tratamientos:

```
A1 LEER SUMPAR LEER SUMIMP LEER ...
ESCRIB
```

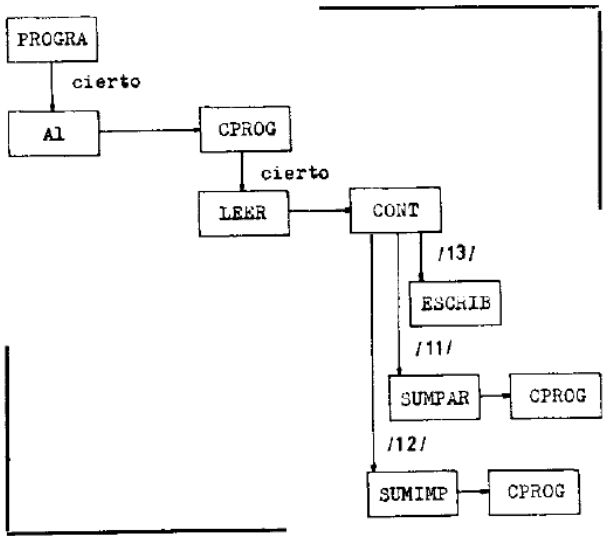
donde la aparición de los tratamientos SUMPAR y SUMIMP está condicionada a los datos de entrada a través de los predicados.

Obsérvese que la ESTRUCTURA DEL PROGRAMA es válida para todos los tipos de problemas que traten un fichero con dos tratamientos alternativos. Esto es posible porque los tratamientos se especifican de forma independiente de la estructura.

El estructuralismo impuesto en un programa escrito en LPE puede ser expresado sin ambigüedad y formalizadamente en términos de las E-gramáticas (E de estructura) [16] [17]. Las E-gramáticas constituyen un algoritmo que describe todas las ejecuciones de un programa dado.

La definición de estas gramáticas constituye el soporte teórico que justifica el mencionado estructuralismo que aporta el LPE. Las E-gramáticas son utilizables también para la prueba de propiedades de programas (finitud, equivalencia, etc.), constituyendo un puente entre la teoría de lenguajes y la PE.

El concepto de árbol programático resulta puramente natural en el estructuralismo del LPE. Para la ESTRUCTURA DE PROGRAMA de nuestro ejemplo correspondería el siguiente árbol programático:



donde las flechas verticales denotan cambio de nivel de abstracción y las horizontales el operador concatenación.

3. ESTRUCTURA DE UN PROGRAMA EN LPE

Un programa LPE consta de las siguientes secciones:

* LENGUAJE OBJETO { FORTRAN / COBOL / ALGOL }
* DECLARACION DE OBJETOS.
* TRATAMIENTOS TERMINALES.
* TRATAMIENTOS NO TERMINALES.
* ESTRUCTURA DEL PROGRAMA.
* CONDICIONES.
* TEXTO TRATAMIENTOS TERMINALES.
* FIN.

y en el orden especificado.

LENGUAJE OBJETO es autoexplicativo y especifica el lenguaje del programa objeto de nuestro traductor.

DECLARACION DE OBJETOS especifica los datos a ser procesados por el programa objeto. En su descripción se ha mantenido un cierto paralelismo con la forma de descripción de la estructura del programa. Los objetos que el lenguaje acepta son: FICHERO, REGISTRO, MATRIZ y ATOMO.

Dado que el propósito de este artículo es presentar la parte algorítmica del LPE, y que por razones de espacio no podemos tratar con detalle la parte de datos, referimos al lector a la publicación [16].

TRATAMIENTOS TERMINALES y NO TERMINALES: estos apartados especifican respectivamente las listas de nombres asignados a los bloques terminales y no terminales.

ESTRUCTURA DEL PROGRAMA, como ya hemos comentado, consiste en la descripción explícita por niveles de abstracción de la estructura del programa, designándose de forma secuencial las acciones a ser realizadas, consiguiéndose así mantener muy próximos los aspectos estáticos (árbol programático) y dinámicos (ejecución) de un programa, lo cual proporciona una mejor legibilidad y documentación del proceso (algoritmo).

CONDICIONES y TEXTO TRATAMIENTOS TERMINALES constituyen la interpretación de los predicados y las acciones terminales, respectivamente, en términos del lenguaje objeto elegido.

4. EL TRADUCTOR DE LPE

Las informaciones más relevantes para el traductor de programas LPE son las descripciones estructural (ESTRUCTURA DE PROGRAMA) y semántica (TEXTO TRATAMIENTOS TERMINALES Y CONDICIONES) del programa objeto; las especificaciones de control no son requeridas, dado que éstas son evaluadas a partir de la descripción estructural, dando como resultado un módulo controlador de la ejecución, el cual mantiene la estructura del programa en forma de árboles implementados mediante vectores.

El programa resultado de la traducción (programa objeto) lleva incorporado el módulo de ejecución configurado a su necesidad. Insistimos en que el programa objeto se obtendrá escrito en COBOL, FORTRAN o ALGOL, según se especifique en la sección CODIGO OBJETO.

Obsérvese que el traductor es un **metaprograma** y que su programa objeto constituye un programa fuente para otro traductor.

Para la codificación del metaprograma se utilizó un sistema de escritura de traductores (TWS) diseñado [18] y experimentado por nosotros, que le permite operar en entornos de tiempo compartido (reentrante).

5. CONCLUSIONES

— LPE constituye una herramienta para la descomposición de sistemas complejos suficientemente versátil y flexible.
— El lenguaje LPE implementado permite la programación estructurada, automatizando la codificación de la secuencia de control dentro del programa.

— La interpretación en términos de un lenguaje de programación de las acciones terminales se demora al último momento, por lo que el lenguaje elegido será siempre el más idóneo.

— La codificación en LPE no requiere la confección de un organigrama, pudiéndose realizar directamente desde el cuadro de descomposición.

BIBLIOGRAFIA

1. Dijkstra, E.
   Notes on structured programming, en
   Structured Programming, Dahl, Dijkstra, Hoare.
   Academic Press, 1972.
2. P. Naur y B. Randell.
   Software engineering: report on a conference sponsored by the NATO.
   Science Committee, 7 a 11 octubre 1968.
   Scientific Affairs Division NATO.
   Bruselas, 1969.
3. Dijkstra, E.
   The structure of the «THE»-multiprogramming System.
   Comunicaciones del ACM, mayo 1968.
4. Liskov, B.
   The design of the VENUS Operating System.
   Comunicaciones del ACM, marzo 1972.
5. Wirth, N.
   Program Development by stepwise refinement.
   Comunicaciones del ACM, abril 1971.
6. Wirth, N.
   Systematic Programming an Introduction.
   Prentice-Hall, 1973.
7. Infosoftware (Infosoftware Procedings).
   Infotech International 1974.
8. Structured Programming.
   Infotech State of the Art Report.
   ISBN 8553-9270-3.
   Infotech International 1976.
9. Naur, P.
   Programming by Action Clusters.
   BIT, vol. 9, núm. 3, 1969.
10. Lecarme, O.
    Structured Programming, programming teaching and the language Pascal.
    SIGPLAN Notices, julio 1974.
11. Knuth, D.
    Structured Programming with go to statements.
    Computing Surveys, vol. 6, núm. 4, diciembre 1974.
12. Warnier, J. D.
    Programación lógica.
    Editores Técnicos Asociados, S. A.
    Barcelona, 1973.
13. Bertini.
    Le COBOL structuré: un modèle de programation.
    Editions d'Informatique.
    Paris, 1973.
14. Lógica de la programación.
    Departamento de Educación Univac.
    Madrid, 1976.
15. Ramos, I., y Moya, R.
    Metaprogramación estructurada.
    Congreso Nacional de Informática y Automática.
    Madrid, 1975.
16. Moya, R.
    Diseño e implementación de un sistema de programación estructurada. (Tesis doctoral.)
    Bilbao, 1977.
17. Moya, R., y Ramos, I.
    SPL (Structured Programming Language), a language for SP.
    IFIP Congress on design and development of algorithmic languages.
    Guidel (France), mayo 1977.
18. Moya, R., y Ramos, I.
    Construcción de un traductor.
    Ministerio de Educación y Ciencia. Madrid, 1974.

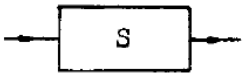
APENDICE

La codificación en LPE de los conjuntos de estructuras de control más salientes en la PE —D y D' (D de Dijkstra)— es como sigue:

Estructuras D:

1. Acciones básicas. Ejemplo: sentencias de asginación, llamadas a procedimientos, sentencias E/S y en general toda primitiva con una sola entrada y una sola salida.

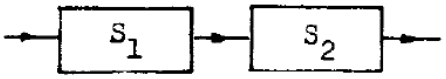
Gráficamente:



LPE: S, siendo S un tratamiento terminal.

2. Concatenación «S_1 ; S_2» de dos estructuras D.

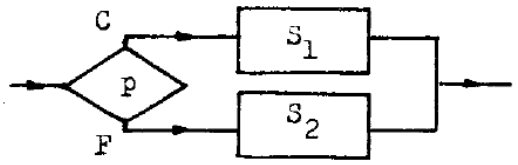
Gráficamente:



LPE: S' . S_1 S_2, siendo S' un tratamiento no terminal.

3. Estructura condicional: **if** p **then** S_1 **else** S_2, donde p es un predicado y S_1, S_2, estructuras D.

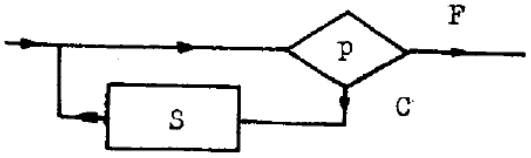
Gráficamente:



LPE: S' . S_1 /p/ S_2 /p/ . , siendo S' un no terminal.

4. Ciclos de la forma: **while** p **do** S, donde p es un predicado y S una estructura D.

Gráficamente:

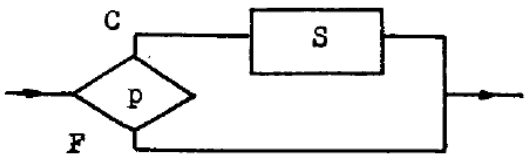


LPE: S' . S S' /p/ . , siendo S' un tratamiento no terminal.

Estructuras D' son una extensión de las D:

1. Cualquier estructura D es una estructura D'.
2. Estructura condicional: **if** p **then** S.

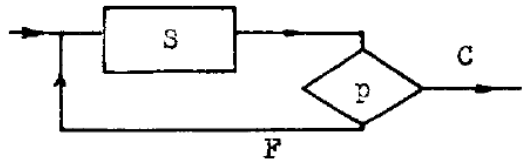
Gráficamente:



LPE: S' . S /p/ ∅ /p/ . ' siendo S' un tratamiento no terminal y ∅ el tratamiento vacío.

3. Estructura cíclica de la forma: **repead** S **until** p.

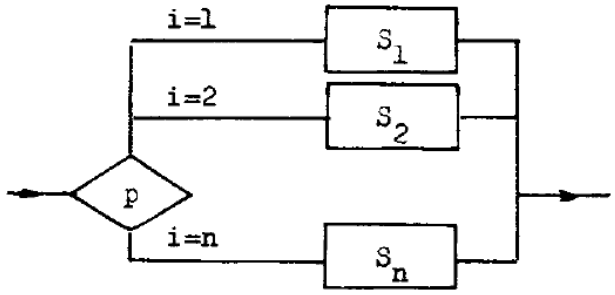
Gráficamente:



LPE: S' . S S'' . S'' . S S'' /p/ ∅ /p/ . , siendo S' , S'' no terminales y ∅ el tratamiento vacío.

4. Estructura case.

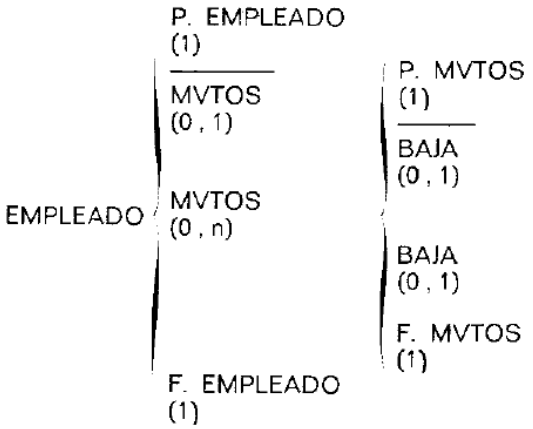
Gráficamente:



LPE S' . S_1 /p_1/ S_2 /p_2/ ... S_n /p_n/.

Como ya hemos indicado en las conclusiones la programación en LPE no requiere la elaboración del organigrama, siendo posible la codificación de la ESTRUCTURA DEL PROGRAMA directamente desde el diagrama de descomposición o cuadro sinóptico.

Seguidamente ponemos un ejemplo:



Una codificación en LPE es:

* ESTRUCTURA DEL PROGRAMA.
EMPLEADO. PEMPLEADO MVTOS FEMPLEADO
MVTOS. MVTOS1 /10/ MVTOS2 /11/.
MVTOS2. TMVTOS MVTOS2 /13/.
TMVTOS. PMVTOS BAJA FMVTOS.
BAJA. BAJA1 /14/ BAJA2 /15/.

DATA DYNAMICS ESPAÑA, S.A.

TERMINALES Y PERIFERICOS

Juan Pérez Zúñiga, 20-B
MADRID-27 - Tel. 408 00 00

Roger de Flor, 49
BARCELONA-13 - Tel. 225 15 26-27

EMPRESA:
DIRECCION:
TELEFONO:
PERSONA DE CONTACTO

"SPL (STRUCTURED PROGRAMMING LANGUAGE) A LANGUAGE FOR STRUCTURED PROGRAMMING"

R. Moya
UNIVAC (MADRID)

I. Ramos
UNIVERSITY OF BILBAO

Key words: Programming languages, Structured programming, Abstraction levels programming.

ABSTRACT.

A programming language together with its translator have been defined and implemented in an UNIVAC 1100 computer. The language is designed following a structured programming approach of abstract levels type.

The translator accepts as input the structure and the semantic description of a given program, together with the objets to be elaborated, and produces as output a FORTRAN, COBOL or ALGOL (depending on programmer's choice) target program following a metaprogramming technologie.

The structure of the program is given by a context free-like grammar (S-grammar) and the functions and predicates interpretation is given in terms of the object language chosen.

1. INTRODUCTION

Program writing following the step-wise refinement technique and using the actual programming languages, is done by a "comments" in natural language recursive substitution by phrases in the programming language chosen, therefore the program description is "refined" succesively up to the description abstraction level is that of a implemented programming language. This programming method does not guarantee the program documentation purposes because in each refinement the previous comments are lost, and what is more important, at the final step of refinement we have only a program description in terms of the programming language chosen, and with the limitation of its descriptive characteristics. Another restriction is that the target language must be chosen since the earlier steps of programming and perhaps will not be the more adecuate language fot the next refinement steps.

Moya, R., Ramos I. SPL (Structured Programming Language). A language for structured programming. Proc. Of the 5th Annual III Conference. 1977.

The implemented language allows a stepwise refinement of program's writing following a hierarchical sequence of abstract levels; in this way the description's accuracy is kept in each level, and the "terminal actions" semantic interpretation in terms of the target language, is delayed until the last refinement is done.
The aim of our method is to keep as near as possible the static and dynamic program aspects (following Dijkstra terminologie). Program execution (dynamic aspect) may be viewed as a sequence of elementary actions that can be described by a certain alphabet of instructions: the corresponding abstract machine instruction repertory.

This instructions sequence that constitues the dynamic aspect of a program description is a string of elements over a finite alphabet. An easier way to describe a subset of strings over an alphabet is a grammar.

This grammar (S-grammar) will describe then the program static aspect and will generate all its possible executions.

The alphabet of actions is anyone, but is the programmer's responsability the description of these actions in terms of the target language chosen.

2. S-GRAMMARS

The program structure is given by a S-grammar (S from structure) that we define following the context-free grammars definition, in the following way:

A S-grammar is the 6-ple $(T_t, T_n, A, \tilde{\pi}, \mathcal{P}, \mathcal{V})$

where $(T_t, T_n, A, \tilde{\pi})$ is a context-free grammar used for describing the program tree-like abstraction level structure, and where:

- T_t: designes the terminal alphabet of the elemental blocs names with have only an entry point and an exit point.

- T_n: designes the auxiliar alphabet of the composed blocs names used in the program structure description. They have only an entry point and an exit point also.

- A: designes the axiom ($A \in T_n$)

- $\tilde{\pi}$: designes the set of rules. $N \to C$

  where $N \in T_n$ and C is a string of elements c_i, with $c_i \in (T_t \cup T_n)$ or is the empty string: ϕ.

- $\mathcal{P}$: is a finite set of names of predicates $\mathcal{P} = \{p_i^n\}$ $i \geq 1$ $n \geq 0$

  where n indicates the n-arity of the predicate.

- $\mathcal{V}$: is a function that for each element of $\tilde{\pi}$ gives an element of $\mathcal{P}$. It is:

$$\forall \pi_j \in \tilde{\Pi} \quad , \quad \mathcal{V}(\pi_j) = p_i^n \in \mathcal{P}$$

Because implementing purposes, the function $\mathcal{V}$ is splited into two other functions:

$$\mathcal{V} = \pi^{\nu} \cdot \nu$$

where the function ν is defined as: $\forall \pi_j \in \tilde{\Pi}$, $\nu(\pi_j) = n \in \mathbb{N}$

(set of natural numbers) and the function π^n is defined as:

$$\forall \pi_j \in \tilde{\Pi} \ , \ \pi^{\nu(\pi_j)} = p_i^n \in \mathcal{P}$$

An example of S-grammar is:

T_t = (INITIALIZATION, READ, COUNT-EVEN, COUNT-ODD, WRITE)

T_n = (AXIOM, PROGRAM, CONT)

A = AXIOM

π = is given by

π1. AXIOM → INITIALIZATION PROGRAM

π2. PROGRAM → READ CONT

π3. CONT → WRITE

π4. CONT → COUNT-EVEN PROGRAM

π5. CONT → COUNT-ODD PROGRAM

φ = is given by:

P1 = TRUE,

P2 = EVEN,

P3 = ODD,

P4 = LAST

ν = is given by:

$\nu(\pi_1) = P_1$, $\nu(\pi_2) = P_2$, $\nu(\pi_3) = P_4$, $\nu(\pi_4) = P_2$, $\nu(\pi_5) = P_3$

Associated with this S-grammar there are the graphic structures:

- Syntactic chart and
- Derivation tree

With its classical meaning. They give a program structure pictorial view (also sometimes called programmation tree...).

The program of our exemple will read a sequence of numbers, count the odd and even numbers separately and print the result, if we suppose defined the terminal actions, and predicates.

For two rules with the same left part the predicates associated with them may be or not, true at a time. In our implementation, we have not alloved it, keeping in this way deterministic the program execution, but we may think also in this possibility with indeterministic or concurrent purposes in mind.

3. THE LANGUAGE SPL

A program in SPL has the following sections:

*OBJECT LANGUAGE { FORTRAN / COBOL / ALGOL }

*DATA DECLARATION

```
*TERMINAL ACTIONS
*NON TERMINAL ACTIONS
*PROGRAM STRUCTURE
*CONDITIONS
*TERMINAL ACTION TEXT
*FIN
```

With the following meaning meanings:

*OBJECT LANGUAGE: is self explanatory. Specifies the target language chosen (three in the implementation done).

*DATA DECLARATION: specifies the data to be elaborated by the target program following a two level schema. The objects (data structures) furnished by the language are:

FILE

RECORD

ARRAY

ATOM

*TERMINAL and *NON TERMINAL ACTIONS: they specify respectively V_t and V_n.

*PROGRAM STRUCTURE: the program structure is given by a S-Grammar-like description. The abstract level structure is explicit.

The actions to be performed by the program are designed sequentially for each level.

*CONDITIONS and *TERMINAL ACTION TEXT: They are respectively the predicate and terminal action interpretation in terms of the target language used.

*FIN indicates the program end.

A complete exemple is given using ALGOL as target language in the Appendix.

4. THE TRANSLATOR

The translator has been implemented using a TWS (Translator Writing System) designed following LL(1) grammar techniques, in a UNIVAC-1100 computer.

The generated object program is a top-down interpreter that uses the tree-like structure created using the S-grammar.

Depending on the target language used the terminal actions may be treated as open or closed subroutines.

The S-grammar trees are represented by a binary tree, one of wich succesion functions is represented by contiguity in the computer store, treating in this way to keep small the storage amount used by the object program.

5. CONCLUSIONS

-The language implemented allows structured programming by decomposition in abstract levels and following a step-wise refinement procedure.

-The program static and dynamic aspects are kept very closed although a grammar is a quite natural tool to describe strings of elements over an alphabet.

-The interpretation target language election is delayed until the last refinement step. It would be possible to do such an election dynamically (by reading the function's interpretation at execution time).

-The use of S-grammars in the programs structure description allows the applications of the Theory of Languages results in solving programs equivalence, determinism, finiteness... problems.

-The iterative while structured bloc recursive definition, produces storage collapsing for large iterations, but it is an implementation solvable problem.

-The Programming System implemented is efectively working in a UNIVAC-1100 computer since November 1976.

6. REFERENCES

Dijkstra E., "Notes on Structured Programming" Academic Press 1972.

Ianov, "The logical schemes of algorithms" Problems of Cybernetics USSR 1960.

Moya R., Tesis Doctoral. Universidad de Bilbao. Marzo 1977.

Pair C., "Compilation" Ecole d'Eté d'Informatique. Neuchatel 1972.

Ramos I., Moya R., "Metaprogramación Estructurada". Proceedings Congreso Nacional de Informática y Automática. Octubre 1975.

APPENDIX

Following we give the complete program of the exemple in paragraph 2. In the example we suppose a card file with a natural number per card, punched in the two initial columns. The end of the file is detected by the figure "99".

The program is:

```
*OBJECT LANGUAGE.ALGOL

*DATA DECLARATION.

FILE DATA DEVICE CARDS.

ESPECIFICATIONS: SEQUENTIAL.

REGISTER CARD 1 CONCEPTS:

1NUMBER A,I2.

FILE LISTING DEVICE PRINTER.

ESPECIFICATIONS: SEQUENTIAL (PRINTER).

REGISTER LINE 2 CONCEPTS:

1 EVEN A, 'EVEN=';I5.

2 ODD      'ODD =';I5.

ATOMO EVEN INTEGER

ATOMO ODD INTEGER.
```

```
ATOMO QUOTIENT INTEGER
ATOMO REMAINDER INTEGER.
ATOMO NUMBER INTEGER.
*TERMINAL ACTIONS
INITIALIZATION READ COUNT-EVEN COUNT-ODD WRITE.
*NON TERMINAL ACTIONS.
AXIOM PROGRAM CONT.
*PROGRAM STRUCTURE.
AXIOM.INITIALIZATION PROGRAM.
PROGRAM. READ CONT.
CONT. WRITE /13/
      COUNT-EVEN PROGRAM /11/
      COUNT-ODD PROGRAM /12/.
*CONDITIONS.
/11/: IF REMAINDER EQL O THEN TRUE ELSE
                          FALSE $
/12/: IF REMAINDER EQL 1 THEN TRUE ELSE
                          FALSE $
/13/: IF NUMBER EQL 99 THEN TRUE ELSE
                          FALSE $
*TERMINAL ACTIONS TEXT.
INITIALIZATION. EVEN=ODD=0$
READ.READ DATA. $
     QUOTIENT=NUMBER //2 $
     REMAINDER=NUMBER-QUOTIENT *2$
COUNT-EVEN. EVEN=EVEN+1 $
COUNT-ODD.  ODD=ODD+1 $
WRITE.WRITE LINE $
*FIN
```

NOTES:

-When a rule has not a predicate associated with it in the PROGRAM STRUCTURE section, it is assumed as TRUE.

- The predicates are numbered in the PROGRAMA STRUCTURE section and explicited in the CONDITIONS section.

-A BNF-like form is used to specifie the S-grammar.

APROXIMACION PROSPECTIVA AL SOFTWARE DE BASE

Isidro Ramos
Prof. de Informática de la
Universidad de Bilbao

Ramos, I. Aproximación prospectiva al software de base. Actas de la Convención Informática Latina. 1977.

1. TESIS

1.1. Consideraciones históricas y analógicas

La Metodología científica halla en la intuición y en la analogía dos útiles poderosos de creación, y nosotros vamos a aprovecharnos de ellos en esta aproximación prospectiva al SOFTWARE de base: Intuición fundada en nuestra experiencia a través de años de trabajo en Software de base y analogía histórica entre la Informática y la Lingüística Clásica (Según las ideas de P.Naur, "Programming languages, Natural languages and Mathematichs" CACM Dec. 1975).

Desde este punto de vista y considerando situaciones homomorfas llegaremos a conclusiones en Informática partiendo de hechos que se producen antes en Matemáticas y en Lingüística, puesto que estas dos ciencias la preceden históricamente.

Dentro de la Informática vamos a ceñirnos al Software de base, aunque, según pensamos, nuestras conclusiones podrían aplicarse a muchos otros aspectos.

En Informática existe una "gran laguna" entre Teoría y Práctica, o sea entre la Informática de nuestras Universidades y la Informática aplicada de nuestras empresas, a pesar de que nosostros nos consideramos todos informáticos y creemos que la "verdadera" Informática se encuentra en la que ejercemos nostros. Para clarificar esta idea es interesante leer el trabajo de S.Gorn "Status of Computer Methodology: When the chips are down" CACM. Nov. 1975.

Ocurre la misma situación entre las Matemáticas llamadas teóricas y aplicadas y por ahora parece que la síntesis está lejos: Muy al contrario, el abismo entre ambas, es cada vez mayor.

Los matemáticos modernos encuentran en la belleza de sus teoremas la justificación de su Ciencia, mientras que los "aplicados" sólo encuentran sentido en la eficacia de sus aplicaciones, e incluso hay otros que alegan que cuanto más se es teórico más se es práctico ("no existe mejor práctica que una buena teoría"), etc.

Pienso que se trata de una polémica bien conocida y sabida por todos los informáticos, como para no hablar más de ella.

De cualquier forma, el hecho de que las Ciencias se dividan en teóricas y prácticas es un resultado de la especialización y de la imposibilidad de que el hombre las abarque completamente después de un cierto tiempo de desarrollo.

En Informática esta duplicidad está clara:

— Los preciosismos sin fin en el desarrollo de los lenguajes de programación (cfr. ALGOL — 68, por no hablar de la Teoría de lenguajes ...) contrastan muy fuertemente con:

— La programación masiva en COBOL realizada en nuestras empresas.

Si aceptamos la analogía Informática-Matemáticas, podríamos obtener el siguiente resultado:
— Esta duplicidad existe y existirá cada vez de forma más clara y explícita.

La segunda disciplina que consideramos próxima a la Informática es la Lingüística Clásica, como todo especialista en Software de base lo conoce. (Basta para ello considerar los trabajos de N.Chomsky).

En esta disciplina es interesante considerar el estudio de las diferencias que la separan de la Informática y en particular las dos siguientes:

1 — Las lenguas naturales se usan fundamentalmente de forma oral, ya que la capacidad de expresión oral está muy desarrollada en el hombre. La puesta a punto interactiva de sus ideas conversando con otros interlocutores es también un proceso normal.

Sin embargo la programación se hace sobre todo de forma escrita y con un grado de interacción muy pobre e ineficaz (basta sólo con reflexionar sobre los lenguajes conversacionales e interpretativos).

2 — Salvo en algunos casos la lengua hablada es vaga e imprecisa, tanto por parte del que habla como por parte del que escucha.

Sin embargo en Informática, las exigencias de precisión son considerables. Esta diferencia no está a favor de los lenguajes de programación puesto que todo lo que puede expresarse con ayuda de un lenguaje de programación, puede también hacerse con una lengua hablada. Mientras que la inversa no es cierta.

Creemos que esta falta de precisión es lo que posibilita el hablar de una forma interminable sobre nuevas ideas: Una lengua tiene un universo de discurso extensible, que permite añadir otras anotaciones (por ejemplo: Los lenguajes de programación).

El análisis de lo que ha pasado con las lenguas que han sobrevivido a sus historias evolucionando y adaptándose y por otra parte el fracaso de las lenguas artificiales (derivadas del ESPERANTO), nos dan dos nuevas ideas:

1 — La simplicidad (puesto que la evolución de una lengua se produce siempre en contra de su complejidad) y

2 — El hecho de haber sobrevivido (nos asegura el consenso sino la corrección de millones de usuarios).

Si trasladáramos estas ideas a la Informática deducimos que tendremos FORTRAN y COBOL para años, puesto que son estos los lenguajes que han quedado, mientras que la complejidad del ALGOL—68 (por perfecto que sea) impide su difusión.

Otra diferencia que hay que señalar es la existencia de un estilo literario (o técnica literaria) en Lingüística que es lo que ha posibilitado la creación de obras maestras en la literatura, mientras que, en programación no tenemos la disciplina homomorfa. En principio aprendemos a **escribir** programas y no a leerlos y menos a hacerlos legibles. Ha sido muy últimamente y sobre todo con la Programación Estructurada, cuando nos hemos ocupado de un cierto estilo de programación que hace más legibles, comunicables los programas ... (la comunicación posibilita el desarrollo de nuestras ideas).

Las conclusiones que podemos sacar son las siguientes:

1 — La separación entre Informática Teórica e Informática Aplicada continuará y se hará más clara.

2 — Es un hecho que programamos en FORTRAN—COBOL y que continuaremos haciéndolo. Será muy difícil introducir nuevos lenguajes, incluso si son muy poderosos. Vemos que la solución no se encuentra de este lado.

3 — El futuro está en las "Artes" que cultivan el estilo de programación, que puede conducir a "obras maestras" en programación (grandes sistemas). Programación estructurada, conversacional.

1.2. El Hardware: Los polisistemas y las redes de ordenadores

La evolución del Hardware se hace para posibilitar:

1 — Sistemas a medida
2 — Sistemas orientados
3 — Sistemas dispersos

siendo además sistemas polivalentes, capaces de realizar "macrofunciones" tanto por Hardware como por Software y susceptibles de combinarse entre ellos para dar lugar a una gran variedad de configuraciones.

También estamos incorporando en el Hardware el Software más costoso: Software de base y aplicaciones standards.

Los polisistemas podrán componerse de forma dinámica y adoptar también una arquitectura adaptada a las necesidades funcionales del usuario, lo que conducirá a redes heterogéneas y cambiantes de ordenadores.

Podemos ver por medio de una tabla la evolución de la Informática distinguiendo cinco generaciones:

1ª. Generación: Automatización de los procesos científicos y comerciales que antes se hacían a mano.

2ª. Generación: Manipulación de grandes cantidades de datos (data processing era).

3ª. Generación: Elaboración de enormes cantidades de datos por ordenador (information processing era).

4ª. Generación: Terminales aislados, Redes (Information on-line era).

5ª. Generación: "Adjoints". Asistencia en tiempo real e "in situ" al hombre en muchas de sus funciones (en particular la Programación).

En un futuro próximo tendremos un aumento de la portabilidad y de los accesos aislados, así como E/S "orales", lo que posibilitará una "verdadera" conversacionalidad.

2. ANTITESIS

2.1. El Software de base: Asíntota ALGOL—68

Según C.Pair (C.Pair, "Les structures de l'Information", Alès 1971) el informático tiene que representar en la memoria de un ordenador **objetos** sobre los cuales quiere efectuar **tratamientos** y propone el esquema siguiente para la Programación (en sentido amplio: Análisis + Programación):

OBJETOS EXTERNOS	→	OBJETOS MATEMATICOS	→	OBJETOS DE UN LENGUAJE DE PROGRAMACION	→	OBJETOS EN MEMORIA

TRATAMIENTOS→ ALGORITMOS→ PROGRAMAS DEL LENGUAJE→ PROGRAMAS MAQUINA

La programación entonces se convierte en una sucesión de representaciones de objetos y tratamientos sobre sucesivas "máquinas virtuales". En cada nivel de abstracción o máquina virtual, hay objetos y tratamientos proporcionados por el "lenguaje máquina" de este nivel.

El proceso de análisis acaba cuando se dispone de una máquina real (por Hardware o Software) que acepta como lenguaje máquina el del último nivel de abstracción.

La evolución de los lenguajes de programación se ha hecho y continúa haciéndose con el ánimo de proporcionar al usuario lenguajes cada vez más poderosos (objetos y tratamientos), pensemos en la evolución:

FORTRAN
COBOL → PL/I → ALGOL — 68

Pero para el "usuario" esta evolución casi no ha existido, siendo el ALGOL-68 una asíntota para su actividad, impuesta por la complejidad del mismo lenguaje. Y sobre todo, la simplicidad de los lenguajes (incluso programar será manipular algunas teclas); el Software de base del futuro ofrecerá servicios adaptados a las necesidades del usuario y no compiladores y sistemas operativos más o menos exitosos.

2.2. El "Programador virtual": Asíntota complejidad.

El Arte de la programación (en la terminología de D.Knuth) no puede definirse como una

disciplina científica, puesto que no se dispone de una medida de su progreso, queda pues por tanto un "arte" (J.T: Schwartz, "What constitutes progress in programming" ACM Forum CACM Nov. 1975), a pesar de que estén empezando a nivel universitario métodos de medida de la "estructura de un programa" (R.Noyes "On a measure of program structure" Tesis PHD, Carnegie Mellon University, Nov. 1973).

Mientras la complejidad del problema a resolver no sobrepase la capacidad de un programador individual, éste aparece históricamente dotado de una serie de útiles de programación que han cambiado su primera aproximación intuitiva. Del programador de máquinas de segunda generación al programador de hoy ha habido una evolución análoga y mimética a la que ha experimentado el Software de base. Hemos pasado por sucesivos "programadores virtuales" que disponen cada vez de lenguajes (organigramas, lenguajes de análisis ....) más potentes para describir su problema. Vemos aquí una interdependencia muy profunda (mimética) entre el hombre y la máquina.

Pero cuando la complejidad del problema impide que un sólo programador sea capaz de resolverlo y se necesita un equipo numeroso de programadores, nos encontramos casi en la situación intuitiva del programador desnudo. Lo que ha conducido, en algunos casos, al abandono de los grandes sistemas.

La programación como actividad comunitaria se encuentra en su segunda generación, gracias a la Programación estructurada, el "programmer steam" ..... lo que configuran "equipos virtuales" de programación.

Por el instante la complejidad de los grandes sistemas continúa siendo una asíntota para los equipos (reales o virtuales) de programación.

3. SINTESIS

Hemos visto que la evolución del Software de base y del papel del programador (individual o colectivo) se realiza en el sentido de una complejidad mayor (objetos y tratamientos más poderosos) de los lenguajes (en general no conversacionales) que usan, lo que difiere totalmente de la situación histórica y de las soluciones aportadas por disciplinas muy próximas como las matemáticas y la Lingüística:

— Simplificación y eficacia (más bien que elegancia) de los lenguajes. Sólo se utilizarán algunas construcciones abstractas.

— Señalar el estilo de programación (como una tecnología) y no los lenguajes, lo que dará origen a una Literatura Informática.

— Conversacionalidad real (para posibilitar el diálogo).

— Evolución de los lenguajes actuales, más bien que introducción de nuevos.

— Software adaptado a las necesidades del usuario y no compiladores o sistemas operativos más o menos acertados.

Todo esto ha conducido a los fracasos de los grandes sistemas del Software de base y de la programación que soportaban.

Pero parece que el número de fracasos no ha sido aún suficiente o que las inversiones no se han amortizado aún suficientemente para cambiar la tendencia de estas disciplinas.

SIMPOSIO 17: Automática e Informática

Martes 3. De 16 a 19 horas

Presidente: Dr. M. Mellado Rodríguez
Secretario: Dr. S. Dormido Bencomo

Comunicación 17 - 4

"MODELO DE REALIZACION DE PRIMIITIVAS DE SINCRONIZACION PARA SISIEMAS MULTIMICROPROCESADOR"

I. Ramos J. Aramberri M. Mellado
Dpto. de Automática e Informática, Facultad de Ciencias, Universidad de Bilbao.

1. INIRODUCCION

En los sistemas multiproceso se suele seguir una estrategia estática de asignación de procesadores especializados que por una parte, permite una sencilla realización hardware pero por otra, tiene el inconveniente de su falta de flexibilidad. En sistemas de tiempo real en los que en función del estado temporal es necesario que determinadas funciones sean desarrolladas con mayor o menor velocidad: vaciado de un buffer próximo al overflow,... esta estrategia estática implica ponerse siempre en el peor de los casos, multiplicando el número de procesadores, los cuales fuera de las condiciones críticas serán superfluos.

La necesidad de una estrategia dinámica de asignación de procesadores y por consiguiente, el hecho de que estos sean recursos equivalentes son dos caracteristicas de nuestro modelo, el cual además, reduce considerablemente las implicaciones hardware mediante soluciones software manteniendo por lo tanto una elevada capacidad de modelización

2. MODELO PROPUESIO

En el modelo propuesto cada microprocesador dispone de una memoria privada con una copia de los recursos software (programas) de los procesos que es capaz de realizar. Además todos los procesadores comparten una memoria común (M.C.) a la que se accede en exclusión mutua asegurada por un árbitro realizado por hardware, y en la que residen los recursos software comunes: variables de control de las primitivas (accedidas mediante un monitor software), listas de representaciones de procesos, variables comunes ... El acceso de un procesador a la M.C. implica:

- que solo él acceda (Arbitro)
- que la M.C. sea "colgada" a ese procesador solo (mediante "Ihree state drivers)

Como el número de procesadores puede ser arbitrario, el Arbitro se ha diseñado solo para dos procesadores. En el caso de un número mayor de procesadores se construye un "arbol árbitro" de tipo binario por repetición del arbitro primitivo. Pese a que esto penaliza la velocidad de respuesta del arbitro, no tiene mayor importancia debido a ser és-

Ramos, I., Aramberri, J. y Mellado, M. Modelo de realización de primitivas de sincronización para sistemas multimicroprocesador. Actas del 75 Aniversario Real Sociedad Española de Física y Química. 1978.

ta muy elevada (circuitos de decisión muy rápidos) siendo sin embargo una aproximación muy modular.

Las características de este modelo son:

1) Exclusión mutua en la ejecución de las primitivas. De esta forma se asegura la coherencia de las variables compartidas. (Asegurada por el Arbitro-Monitor).

2) Uso protegido y restringido de las primitivas y sus variables asociadas (Arbitro-Monitor).

3) Posibilitan el intercambio de la PSW ("representación del proceso").

Como además el contexto es multiprocesador se cumple:

- Un solo proceso a la vez ejecuta una primitiva (Arbitro-Monitor).
- Indivisibilidad en la ejecución de las primitivas (enmascarando las interrupciones).

De esta forma la espera activa de un proceso dura como máximo la ejecución de una primitiva.

Esquemáticamente el MONITOR realiza las siguientes acciones:

```
MONITOR: Deshabilitar interrupciones;
         Emitir señal al árbitro;
         Emitir señal colgar M.C.;
         Primitiva;
         Emitir señal descolgar M.C.;
         Emitir señal al arbitro;
         Habilitar interrupciones;
```

La emisión de señales se hace via las puertas de E/S normales.
La estrategia de asignación de recursos reside en el gestionador de recursos y depende del problema concreto.

3. CONCLUSIONES

Las características del modelo propuesto son:

1) Poca especificidad del hardware extra necesario y modularidad de este.
2) Elevada capacidad de modelización (separa claramente asignación de recursos de la cooperación de procesos).
3) Estrategia dinámica de asignación de procesadores.

El modelo propuesto es parcialmente operativo sobre micro-procesadores INTEL-8080 y está siendo usado en la programación de Sistemas de tiempo real.

4. BIBLIOGRAFIA

- LARSEN, RONY, IXIOS: "The Bugbook V and VI E & L Instruments.

DISEÑO Y DESARROLLO DE UN SISTEMA MULTIMICROPROCESADOR

J. Aramberri, I. Ramos, J.R. Garitagoitia

Dpto. de Automática e Informática

Facultad de Ciencias. Universidad de Bilbao

Aramberri, J., Ramos, I., Garitagoitia, J. R. Diseño y desarrollo de un sistema microprocesador. Actas I Jornadas sobre microprocesadores. 1978.

1.- INTRODUCCION
2.- PROCESOS EN EL SISTEMA

2.1. Estados de los Procesos
2.2. Transiciones entre estados
2.3. Descriptivos de los procesos

3.- ARQUITECTURA DEL SISTEMA

3.1. Estructura de bloques
3.2. Arbitro asíncrono
3.3. Switch I/O

4.- MECANISMO DE SINCRONIZACION (SEMAFOROS)

4.1. Introducción
4.2. Ejecución de las primitivas

5.- ASIGNADOR DE PROCESADORES. POLITICA DEL ASIGNADOR
6.- CONCLUSIONES

1.- INTRODUCCION

Los sistemas multiprocesadores, y en particular los realizados con micros suelen utilizar una estrategia estática de asignación de procesadores especializados. Se utilizan procesadores altamente funcionales, con una realización hardware sencilla, pero con poca flexibilidad, lo que da lugar en algunos momentos a la inactividad de algun procesador que se encuentra en espera activa. Nuestro sistema soporta una estrategia dinámica en función del estado temporal del sistema por medio de unas funciones que modifican la prioridad de ejecución de los procesos.

Tendremos un conjunto de procesos y procesadores, considerando indiferenciados los procesadores, y un mecanismo de asignación de procesadores que establecerá la relación procesador-

proceso atendiendo al estado de ocupación de buffers, necesi-- dad de acelerar ciertos procesos, etc.

El trabajo tiene tres aspectos:

a) Mecanismo de comunicación y sincronización entre procesos, desarrollando unas primitivas de sincronización.

b) Asignador de procesadores, que modifica las relaciones proce sador-proceso del sistema.

c) Aplicación concreta a un Registrador cronológico de estados.

2.- PROCESOS EN EL SISTEMA.

2.1. Estados de los Procesos

En todo momento, un proceso si existe se encontrará en alguno de los siguientes estados.

E1: Activo elegido: Posee todos los recursos que necesita para su ejecución.

E2: Activo dispuesto: Posee todos los recursos que necesita pa ra su ejecución excepto el procesador.

E3: Bloqueado intrínseco: El proceso ha quedado detenido al realizar una primitiva de sincroniza ción.

2.2. Transiciones entre estados

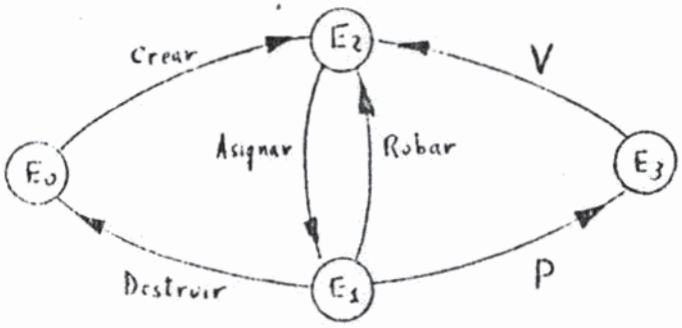
Llamando E_ϕ al estado de "no existencia" de un proceso, podemos considerar la tabla de transiciones entre estados:

→	E_0	E_1	E_2	E_3
E_0	—	—	Crear	—
E_1	Destruir	—	Robar	P
E_2	—	Asignar	—	—
E_3	—	—	V	—

P y V: primitivas de sincronización de Dijkstra.

Asignar y Robar: acciones del asignador de procesadores.

Crear y destruir: primitivas de acción directa.



2.3. Descriptivos de los Procesos

De cada proceso que existe dentro del sistema creamos un descriptivo que lo representa. Constará de

NOMBRE DEL PROCESO	ESTADO	$R_\emptyset..R_N$ (PSW)	S E M	S U C

Siendo $E_\emptyset...E_N$ el contenido de los Registros del procesador cuando el proceso queda bloqueado, o le ha sido robado el procesador S E M el número del semáforo ante el que espera.

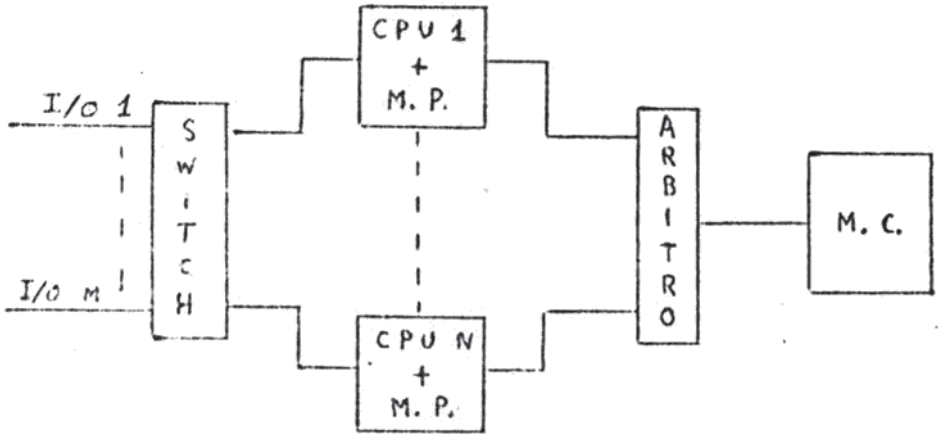
Estos descriptivos se encontrarán formando listas ligadas de procesos activos dispuestos, activos elegidos y de procesos bloqueados ante un semáforo dado.

3.- ARQUITECTURA DEL SISTEMA

3.1. Estructura de bloques

Cada nodo del sistema constará de C P U y memoria privada. En la memoria privada (M.P.) están las copias de los procesos que el procesador de ese nodo es capaz de ejecutar.

La comunicación entre procesadores se realiza a través de una memoria común (M.C.) para lo que es necesario disponer de un arbitro de contención de memoria.



3.2. Arbitro asíncrono

El árbitro de contención de memoria está basado en un circuito que realiza esta función para dos procesadores, creándose un "arbol arbitro" de tipo binario para el caso de un mayor número de procesadores.

Es muy importante la velocidad de respuesta del árbitro, sobre todo en el caso de tener que realizar un "arbol binario" para n procesadores, por lo que utilizan circuitos de decisión rápidos de tipo Schottky o C Mos

En caso de conflicto de acceso simultáneo al acceder al

árbitro, este emplea un sistema de prioridades cambiantes para compensar el servicio.

Igualmente utilizaremos este módulo para asegurar la exclusión mutua en la ejecución de las primitivas de sincronización.

Los buses de datos y direcciones de M.C. y de cada nodo se conectan entre sí por medio de dispositivos 3-states.

3.3. Switch I/O

Cuando un proceso activo elegido desea utilizar un periférico, accede a el en modo protegido, en sección crítica.

Independientemente del procesador en que se esté ejecutando, emite una señal al switch para disponer de conexión con el periférico, a través de puertas 3-states.

4.- MECANISMO DE SINCRONIZACION (SEMAFOROS)

4.1. Introducción

Como mecanismo de sincronización hemos escogido los semáforos de Dijkstra de tipo entero y con lista de espera.

```
INIT  (S,K)      S:= K; lista = vacía;

P(S)             S:= S-1;
                 if  S<Ø then begin bloquear (P corriente);
                                    insertar (P corriente, lista);
                                    end;

V(S)             S:=S+1;
                 if S≤Ø  then begin extraer (P elegido,lista);
                                    desbloquear (P elegido);
                                    end;
```

Para gestionar los semáforos se ha implementado el mecanismo hardware que permita la ejecución en exclusión mutua de las primitivas P y V, e INIT (Arbitro-monitor).

El mecanismo arbitro-monitor garantiza:

a) La exclusión mutua en la ejecución de las primitivas, que han de ejecutarse como procedures indivisibles.

b) Uso protegido de las mismas primitivas y variables asociadas.

c) Se modifican los algoritmos clásicos de las primitivas P y V para introducir además una consulta al asignador de procesadores que puede intercambiar la PSW del proceso activo elegido (acción Asignar) si se dan las condiciones que implican un cambio de asignación del procesador.

La implementación del arbitro-monitor limita la espera activa de un proceso al tiempo que tarde en ejecutarse una primitiva.

4.2 Ejecución de las primitivas

La ejecución de las primitivas se realiza mediante unas macroinstrucciones definidas de la forma siguiente

MONITOR:-Deshabilitar interrupciones
-emitir señal al arbitro (colgar memoria)
-Primitiva de sincronización
-emitir señal al arbitro (descolgar memoria)
-habilitar interrupciones

La emisión de señales se hace via las puertas de I/O.
El arbitro que se utiliza es el mismo de la M.C. La diferencia de uso estriba en el periodo de contención de acceso a los demás procesadores. En el caso de acceso a una palabra de M.C. dura un ciclo máquina, y en la primitiva dura todo el tiempo de ejecución de la misma, con lo que se garantiza la exclusión

mutua pues residen en la M.C. las variables, (semáforos) y listas que maneja la primitiva.

Con el apoyo de las primitivas P y V se implementan los accesos en sección crítica a consulta y modificación de tablas que residan en M.C. y los mecanismos de comunicación entre procesos productor-consumidor con listas de mensajes.

5.- ASIGNADOR DE PROCESADORES

Es uno de los aspectos fundamentales del sistema y soporta la política de Asignación dinámica en función del estado temporal del sistema. Por el momento su construcción depende del problema concreto a tratar, aunque nuestra intención es tipificar las condiciones que supongan cambios en la política de asignación para evitar su reescritura en cada caso concreto de aplicación.

Maneja las representaciones de los procesos que existen en el sistema, y tiene las funciones de Asignar y Robar procesador vistas en el diagrama de transición entre estados.

Dos opciones son posibles dependiendo del tamaño del sistema: que exista un solo ejemplar en M.C., o que exista en todo caso una copia idéntica en cada conjunto CPU+M. Privada. Pero en todo caso su ejecución sería siempre en sección crítica ya que es un recurso único.

6.- CONCLUSIONES

El sistema presentado supera los desarrollados sobre microprocesadores que utilizan monitores de comunicación.

La espera activa en estos últimos puede ser tan larga como la ejecución de una subrutina del monitor, y en nuestro caso se limita a la ejecución de la primitiva de sincronización.

Otro aspecto que mejora el servicio del sistema es la filoso

fía del asignador de procesadores. Por medio del sistema de prioridades cambiantes en función del estado del sistema, se aceleran ciertos procesos, y con ello la respuesta en tiempo real.

La arquitectura del sistema es sencilla y modular, con un importe de circuitería adicional pequeño, pero que garantiza el funcionamiento y la comunicación en el sistema multiprocesador.

BIBLIOGRAFIA

- Asynchronous Arbiters W. Plummer IEEE Trans of Computer Enero 1972.
- Asynchronous Arbiter Module de R.C. Pearce IEEE Trans of Computer eptiembre 1975.
- Analisis of Multiple-Microprocessor System Architectures, de A.J. Weissberger. Computer Design Junio 1977.
- Une Architecture Parallele asynchrone pour les systemes de commande, de M. Courvoisier. AFCET: Redes de Petri, Jornada Mayo 1977, Paris VI.
- Operating System Principles. Per Brinch Hansen. Ed. Prentice-Hall. 1973.
- The logical Design of operating systems. Alan C. Shaw. Ed. Prentice Hall. 1974.
- Operating Systems. Tsichritris-Bernstein. Ed. Academic Press 1974.
- Monitors: An operating System. Structuring concept. C.A.R. HOARE. Communications. ACM Octubre 1974.

SP-Grammars

(Context-Free Grammars for Structured Programming)

R. Moya
Education Dep. of
UNIVAC Spain

I. Ramos
University of
BILBAO Spain

1.- Introduction

From the structured programming point of view, an abstract program (program Scheme) does not reflect the decomposition into refinement steps (abstraction levels, stepwise refinement ...) nor the typical structured programming structures (D, D'...). If we need to prove properties of a computer program while mantaining its structural aspects we can use other devices.

One of such devices is presented here. We call it SP-Grammar (SP for Structured Programming), and it is similar to a Context-free grammar but it uses predicates.

Using SP-Grammars it is possible to perform the proof of properties of structured programs, supported on the theory of context-free languages, which is a well developed field.

The SP-Grammars are an excellent tool among the meta-programming techniques of structured programming. The goal of this paper is to present the properties. Which make these SP-Grammars effective.

2.- SP-Grammars

A program execution (dynamic aspect in the Dijkstra sense) may be viewed as a sequence of elementary actions that can be described by a given alphabet of instructions, i.e. the corresponding abstract machine instruction set.

This sequence of instructions, that constitutes the dynamic aspect of a program description, is a string of elements over a finite alphabet. A grammar is an easy way to describe a subset of strings over an alphabet.

Moya, R., Ramos, I. SP Gramars (Context Free Gramars for Structured Programming). ACM SIGPLAN. Vol.4 Nº1. 1979.

This grammar (SP-grammar) will then describe the static aspect of the program (its structure) and will generate all its possible executions. We define it in the following way:

An SP-Grammar is the 6-ple:

$$\langle T_t, T_n, A, \pi, \mathcal{P}, \nu \rangle$$

where (T_t, T_n, A, π) is a context-free grammar used to describe the tree-like abstraction level structure of the program and where:

- $\mathcal{P}$ is a finite set of names of predicates (in the program scheme sense)
  $$\mathcal{P} = \{P_i^n\} \quad i \geq 1 \quad n \geq 0$$
  where n indicates the n-arity of the predicate

- ν is a function that for each element of π gives an element of $\mathcal{P}$: $\forall \pi \in \pi, \ \nu(\pi) = P_i^n \in \mathcal{P}$

Let SP be an SP-Grammar, and let t, t' be strings: $t = xUy$, $t' = xuy$, where $x, y, u \in (T_t \cup T_n)^*$, $U \in T_n$ (the strings x and y may be empty).

We say that the string t directly produces the string t', written $t \succ\!\!- t'$ if $U \rightarrow u \in \pi$ and $\nu(U \rightarrow u)$ is TRUE.

The language $L(SP)_+$ generated by the SP grammar is the set of sentences: $X \in T_t$ such that:

$$L(SP) = \left\{ x \mid A \overset{+}{\succ\!\!-} x \text{ and } x \in T_t^+ \right\}$$

as usual, we write $t \overset{*}{\succ\!\!-} t'$ if $t = t'$ or $t \overset{+}{\succ\!\!-} t'$ (derivation of length n). In other words, from the point of view of the terminal actions string generation, only the productions having TRUE predicates associated with them are meaingful.

The correspondence between an SP-Grammar and a program scheme can be presented as follows:

Let Σ_S be the alphabet of a program scheme S (2-1). Then an SP-Grammar can describe the program scheme S if:

1) $\mathcal{P}$ is the same set of predicates for both, program scheme and SP-Grammar.

2) A mapping M exists which associates each element in T_t to a set of sentences over Σ_S.

example: For the following program scheme S:

1: $x_1 := f_1$

2: $p_1 (x_2)$ 6,3

3: $x_1 := f_2(x_2,x_1)$

4: $x_2 := f_3(x_2)$

5: $p_1 (x_2)$ 6,3

6: STOP

the Grammar:

$T_t = \{t_1,t_3,t_4,t_6\}$, $T_n = \{A,n_1,n_2,n_3\}$, Axiom: A

$\mathcal{P} = \{p_1^1,p_2^1,\text{TRUE}\}$

Π is the set of rules: ϑ is the function:

1. $A \rightarrow t_1 n_1$ $\quad \vartheta(1) = \text{TRUE}$
2. $n_1 \rightarrow n_2$ $\quad \vartheta(2) = p_1^1$
3. $n_1 \rightarrow t_6$ $\quad \vartheta(3) = p_2^1$
4. $n_2 \rightarrow t_3 t_4 n_3$ $\quad \vartheta(4) = \text{TRUE}$
5. $n_3 \rightarrow n_2$ $\quad \vartheta(5) = p_1^1$
6. $n_3 \rightarrow t_6$ $\quad \vartheta(6) = p_2^1$

describes the schema S making:

$p_2^1 = p_1^1$ and $t_1 = x_1 := f_1$

$t_3 = x_1 := f_2(x_2,x_1)$

$t_4 = x_2 := f_3(x_2)$

$t_6 = \text{STOP}$

Associated with the SP-Grammars there are graphic structures and within them we have a syntactic chart and a Derivation tree, where vertical arrows mean a change of abstraction level and horizontal arrows mean the concatenation operator.

3.- Interpretation

We can define an SP-Grammar interpretation in terms of the program scheme interpretation, that is, given:

1) A nonempty set of elements D (called the domain of the interpretation).
2) To each function f_i^n in Σ_S we assign a total function mapping D^n into D denoted IF.
3) To each predicate p_i^n in Σ_S we assign a total predicate mapping D^n into $\{T,F\}$ denoted $I\mathcal{P}$.

we call interpretation φ of an SP-Grammar the following two mappings.

- φ_1 which associates to each element in T_t a subset of IF.
- φ_2 which associates to each element in $\mathcal{P}$ an element in $I\mathcal{P}$.

In other words, let L be a programming language, and them, the interpretation φ will consist of:

- φ_1 which associates to each element in T_t a set of sentences over L.
- φ_2 which associates to each element in $\mathcal{P}$ a predicate sentence of L.

We shall call Program the couple $p=\langle SP,\varphi\rangle$, where SP is an SP-Grammar and φ is an SP interpretation.

Using SP-Grammar as program structure description it is possible to build a metaprogram accepting the SP-Grammar (Structural aspect) and its interpretation (semantic aspect) as input, and giving the equivalent program in the programming language L as output.

In this way the programmers do not have to specify the control

flow of the program see (3-1). In the appendix we give some examples.

4.- Deterministic program execution

Let $p = \langle SP, \varphi \rangle$ be a program. If we do not want to consider parallel processing, we must have a deterministic condition for all the predicates in one same production of Π.

For each production $\pi \in \Pi$ such as:

$$X \quad /1/C_1 \quad /2/C_2 \quad \ldots \quad /n/C_n$$

where $X \in T_n$ and $/i/\longrightarrow p_i$ is a numbering function of φ, and the C_i are strings of elements $c_i \in (T_t \cup T_n)$ then the deterministic condition is:

$\forall X \in T_n$, X will be the left part of a rule, then the predicates associated with the right parts of this rule verify the conditions:

$$\sum_{OR}^{n} p_i = \text{TRUE and if } p_k = \text{TRUE}; \ \forall p_j \ j \neq k \quad p_j = \text{FALSE}$$

In other words, for two or more rules having the same left part the predicates associated with them may or may not be TRUE at the same time. In this paper we have not considered this question. However we can look ahead to the possibility of having two or more predicats time at the same time with indeterministic or concurrent programming purposes.

REFERENCES

2-1 Manna Z., in Currents in the theory of computing Edited by: Alfred V. Aho; Prentice-Hall 1973

3-1 Moya R., Ramos I., SPL (Structured Programming Language) a language for Structured Programming. IFIP Congress on design and development of algorithmic languages. Guidel (France) May 1977.

A-1 Henry F. Ledgard and Marcotty, A Genealogy of Control Structures CACM November 1975.

A-2 Wirth N., Program Development by Stepwise Refinement CACM April 1971.

APPENDIX

The structural and semantic aspects of the D and D' structures (A-1) can be described using SP-Grammars as follow:

1a. instructions: t where $t \in T_t \vee T_n$

1b. compositions: $t_1 t_2$ where $t_1, t_2 \in T_t \vee T_n$

1c. if - then - else: if P then N_1 else N_2,

Π set, 1: $N \rightarrow N_1$; $\mathcal{P}$ set, 1: p

2: $N \rightarrow N_2$; 2: $\overline{p}$

$N \in T_n$; $N_1, N_2 \in (T_t \vee T_n)^*$

which we can also write as follows in the BNF notation:

$N ::= /p/\ N_1 \quad /\overline{p}/\ N_2$

1d. while-do loops: while p do N_1;

$N ::= /p/\ N_1 N \quad /\overline{p}/\ \phi$; is the empty action

2c. repeat-until loop: repeat N_1 until p;

$N ::= /\text{TRUE}/\ N_1 N_2$ ("TRUE" can be omitted without ambiguity)

$N_2 ::= /p/\ N N_2 \quad /\overline{p}/\ \phi$

2d. case statements: if p_i is TRUE N_i is executed;

$N ::= /p_1/\ N_1 \quad /p_2/ N_2 \quad \ldots \quad /p_n/ N_n$

A SP-Grammar that closely describes the Stepwise refinement followed by Wirth in his paper (A-2) for the generalized 8-Queens Problem is as follows:

(we will use here a convenient notation for input to a metaprogram)

T_t = INICIALIZATION, CONSIDERFIRSTCOLUMN, ADVANCEPOINTER, TESTSQUARE, SETQUEEN, REMOVEQUEEN, NEXTCOLUMN1, NEXTCOLUMN2, WRITERESULTS, RECONSIDERER1, RECONSIDER, GETOUT.

T_n = solutions, iteration, validation, possiblesolution, testcolumn, regress, regre1, regre2, regre3, reconsiderpriorcolumn, reconsider2, considernextcolumn, nextcon.

A = solutions.

T_1 is given by:

solutions. INICIALIZATION CONSIDERFIRSTCOLUMN iteration.

iteration. /1/ testcolumn validation iteration GETOUT.

validation. /2/ SETQUEEN considernextcolumn possiblesolution /3/ regress.

possiblesolution. /7/ WRITERESULTS regress GETOUT.

testcolumn. /4/ ADVANCEPOINTER TESTSQUARE testcolumn GETOUT.

considernextcolumn. /8/ NEXTCOLUMN1 nextcon GETOUT.

nextcon. /8/ NEXTCOLUMN2 GETOUT.

regress. reconsiderpriorcolumn reglel.

regre1. /5/ REMOVEQUEEN regre2 GETOUT.

regre2. /6/ reconsiderpriorcolumn regre3 GETOUT.

regre3. /5/ REMOVEQUEEN GETOUT.

reconsiderpriorcolumn. RECONSIDER1 reconsider2.

reconsider2. /5/ RECONSIDER GETOUT.

where the "/digit/" symbols refer to the predicates of the rules and the rules without this symbol are always associated with the TRUE predicate.

The use of "/digit/" is a simplified way of specifyng $\mathcal{D}$. The interpretation of $\mathcal{P}$ is:

/1/: $(j > 8) \vee (j < 1)$

/2/: safe

/3/: $\overline{\text{safe}}$

/4/: $\overline{\text{safe } \vee \ (i=8)}$

/5/: $\overline{j < 1}$

/6/: $(i=8) \wedge \overline{(j < 1)}$

/7/: $j > 8$

/8/: $\overline{j > 8}$

Finally, the interpretation of T_t in terms of NUALGOL is:

INICIALIZATION. FOR C=(1, 1, 8) DO A(C)=TRUE;

```
                    FOR C=(1, 1, 15) DO RIGHTDIAGONAL (C)=TRUE;
                    FOR C=(1, 1, 15) DO LEFTDIAGONAL (C)= TRUE;
                    I = 0; SAFE = FALSE.

CONSIDERFIRSTCOLUMN.C = 1; X(1) = 0;

ADVANCEPOINTER. X(C) = X(C) + 1; I = X(C);
                EXPRE1 = -1 + C + I; EXPRE2 =8+C-I;

TESTSQUARE. SAFE= A(I) AND LEFTDIAGONAL(EXPRE1) AND
                  RIGHTDIAGONAL(EXPRE2);

SETQUEEN. A(I)=LEFTDIAGONAL(EXPRE1)=RIGHTDIAGONAL(EXPRE2)=
               FALSE;

REMOVEQUEEN. A(I)=LEFTDIAGONAL(EXPRE1)=RIGHTDIAGONAL(EXPRE2)
                =TRUE;

NEXTCOLUMN1. C=C+1;

NEXTCOLUMN2. X(C)=0; SAFE=FALSE; I=X(C);

RECONSIDER1. C=C-1;

RECONSIDER.  I=X(C) ; EXPRE1= -1+C+I; EXPRE2= 8+C=I;

WRITERESULTS. WRITE X; SAFE=FALSE;

GETOUT. COMMENT THIS IS AN EMPTY ACTION;
```

CONCLUSIONS

. SP-Grammars can be considered as a bridge between Structured Programming and the Theory of Languages.

. An SP-Grammar represents the program structure, and together with an interpretation in terms of a programming language, constitutes a valid input to an operating metaprogram (3-1).

4.º CONGRESO
INFORMATICA Y AUTOMATICA
MADRID -16/19 OCTUBRE 1979

SESION
14

MODELO DE SISTEMA MULTIMICROPROCESADOR

por

J. Aramberri*
I. Ramos**
M. Mellado***
J.R. Garitagoitia****

RESUMEN

El propósito de esta comunicación es la exposición y desarrollo de los principios de diseño de sistemas multiprocesadores. La evolución de los circuitos integrados y el bajo coste de las C.P.U. de microprocesadores permite utilizar arquitecturas de sistemas que hasta el momento no eran accesibles desde el punto de vista económico.

El modelo multiprocesador expuesto se puede clasificar como MIMD(1), DSM(2). Es un sistema fuertemente acoplado (loosely coupled), característica propia de la interconexión de procesadores por medio de una memoria compartida.

Para minimizar el efecto de estrangulamiento (botlelneck) generado por el acceso de los procesadores a memoria común, se analizan distintas soluciones (memorias paginadas, duplicación de programas objeto).

*,***Facultad de Ciencias. Universidad de Bilbao
**Facultad de Ciencias Universidad de Valencia
****IKASLAN S.A.

Aramberri, J, Ramos, I., Mellado, M., Garitagoitia, J. R. Modelo de Sistema Microprocesador. Actas del 4º Congreso de Informática y Automática. 1979.

1. INTRODUCCION

Los sistemas multiprocesadores desarrollados últimamente se pueden clasificar en dos tipos principales, según la estrategia de utilización de las CPU.

Por una parte están aquellos sistemas que emplean procesadores especializados, (un proceso-un procesador). Cuando un proceso queda bloqueado por motivos de sincronización se infrautiliza la CPU que lo soporta.

En una concepción distinta, totalmente opuesta, se encuentran los procesadores no especializados. Cada uno de ellos es capaz de ejecutar todo un conjunto de procesos. Pero al precio de un coste elevado en el tiempo de acceso a memoria, o utilizando una parte considerable del tiempo de CPU en el ejecutivo.

El sistema multiprocesador expuesto a continuación dispone de procesadores de ambos tipos, especializados y no especializados. El motivo es la obtención de un diseño que participe de las ventajas proporcionadas por uno y otro tipo de estrategia en la asignación de las CPU.

La existencia de procesadores no especializados, junto con una política de asignación basada en prioridades dinámicas (función del estado del sistema) permite mejorar la respuesta del sistema, acelerando los procesos en trance de saturación etc.

La distinción entre procesadores especializados y no especializados se establece a nivel de Software. Desde el punto de vista Hardware, el sistema es modular, y todos los nodos tienen la misma arquitectura. La distinción entre nodos tiene lugar a la hora de configurar el sistema.

2. ARQUITECTURA DEL SISTEMA

Es un sistema multiprocesador desconcentrado, no distribuido ya que las CPU se encuentran físicamente próximas unas a otras.

Cada nodo dispone de un procesador y una zona de memoria local. A su vez todos ellos tienen acceso a una memoria común o memoria compartida.

2.1 Arbitro de Memoria

Para evitar los conflictos de acceso a la memoria común se ha diseñado un árbitro asíncrono binario de tipo modular (3). Con el circuito árbitro se asegura la exclusión mutua entre los procesadores en el acceso a la memoria común.

Las características más destacables del mencionado circuito son: El establecimiento de una alternancia en las prioridades de los nodos para los casos de conflicto, la modularidad del diseño, y el tiempo de respuesta para el arbitraje muy pequeño (4) (5).

Existen, de todas formas, distintos circuitos árbitros ya sean de tipo síncrono o asíncrono descritos en (6)(7)(8)(9). Pero por sus características presentan inconvenientes como prioridades fijas por Hardware, o la exigencia de circuitos contadores cíclicos para establecer "turno" de acceso, o lentitud de

respuesta.

Las funciones de arbitraje necesarias para el acceso a memoria quedan esquematizadas con el diagrama de la FIGURA 1. Para acceder a una zona de memoria común se emplean las señales Request (I), Grant (I) y Wait (I); I=1, n (n=nº de procesadores del sistema).

En nuestro caso el árbitro está centralizado en un circuito y requiere al menos dos lineas por procesador. Existen árbitros que permiten obtener una descentralización eficaz, disminuyendo el nº de líneas. Por ejemplo, los árbitros síncronos de tipo contador. Pero el coste de disponer de una copia del circuito árbitro en cada nodo y el tiempo de respuesta que puede ser bastante elevado disminuyen su utilidad en sistemas que requieren prestaciones altas.

2.2 Arbitro para Sincronización

Las Primitivas de sincronización del sistema están implementadas por Software. Cada una de ellas está representada por una macroinstrucción o subrutina. Diferentes primitivas acceden a estructuras de datos comunes (semáforos, buffers, descriptivos de procesos...). Según el tipo de las tareas en el sistema. y la longitud de las secciones críticas, se emplean las primitivas más adecuadas (10).

Ahora bien, una primitiva de sincronización ha de tratarse como una instrucción indivisible, para asegurar la coherencia en las Estructuras de información sobre las que opera.

En un sistema monoprocesador basta únicamente con inhibir las interrupciones para obtener una correcta ejecución de las primitivas. Pero en un sistema multiprocesador se requiere un mecanismo de contención para garantizar la indivisibilidad de la primitiva. El problema es similar al de contención en el acceso a memoria, y se resuelve igualmente con un circuito árbitro, que puede ser físicamente el mismo (7).

Con un solo circuito árbitro para el acceso a memoria y las primitivas de sincronización, la espera máxima de un procesador es:

$$T = (n-1).T_{ep}$$

n = nº de procesadores del sistema

T_{ep} = tiempo de ejecución de una primitiva

Es un periodo de tiempo muy grande, ya que la ejecución de una primitiva bloquea no solo la ejecución de otras primitivas por los demás procesadores, sino también cualquier tipo de acceso a la memoria común. Este efecto se presenta como consecuencia de identificar la èxclusión mutua en ambos casos, con la posesión del Bus de memoria. El árbitro, más que la memoria común, se convierte en un cuello de botella que puede disminuir el rendimiento del sistema multiprocesador.

El Modelo presentado en este trabajo discrimina los árbitros de memoria y de sincronización, con el fin de evitar estos efectos negativos.

2.3 Aspecto General

Nuestro sistema multiprocesador se basa en el empleo de una memoria común con un árbitro para establecer la contención.

Se pueden establecer particiones en la memoria común, paginándola. Con un árbitro para cada página de memoria común los conflictos de acceso disminuyen, y se incrementa la velocidad media de acceso a memoria común.

También para cada grupo de primitivas de sincronización que empleen las mismas variables se puede implementar un árbitro. Consecuentemente, las primitivas que acceden a distintos datos (semáforos, colas de mensajes) pueden ejecutarse paralelamente en el tiempo.

Estas medidas suponen un incremento bastante notable en el costo del Hardware, al aumentar el nº de árbitros. Desde nuestro punto de vista, consideramos que la mejor relación costo-eficiencia del sistema se encuentra en el caso de utilizar un árbitro dedicado a memoria y otro a sincronización. El diagrama de bloques que representa el sistema es el de la FIGURA 2. En él quedan reflejados los dos árbitros y su función en el sistema.

Desde el punto de vista Hardware el diseño es totalmente modular. Cada nodo consta de CPU, y memoria privada.

3. ORGANIZACION SOFTWARE

3.1 Tipos de procesos en el sistema

La elección de un sistema multiprocesador supone la descomposición de un proceso complejo en una serie de tareas o subprocesos mas o menos independientes entre si. La ejecución de estos subprocesos se lleva a cabo en las CPU del sistema de forma concurrente.

Cada una de estas tareas consta de:

- Programa objeto, no modificable, almacenado en ROM
- Zona de variables locales a la tarea, con espacio en RAM
- Stack asociado a cada tarea, también en RAM
- Zona de Variables de Sincronización, colas de mensajes, etc. situados en RAM accesible al ejecutivo.

Se va a establecer una relación entre cada una de estas tareas y el conjunto de procesadores (CPU) disponibles, que indica en que procesadores se puede ejecutar la mencionada tarea. Esta información estará contenida en el descriptivo del sistema.

Atendiendo a esta relación proceso-procesador, establecemos una clasificación de los nodos del sistema. Cuando la relación proceso-procesador es inyectiva, diremos que estamos ante un nodo especializado. Cuando no se da esa condición, estamos ante nodos no especializados, FIGURA 3.

3.2 Nodos especializados

Están caracterizados por la asignación estática a un pro

ceso o tarea. Por otra parte, el citado proceso únicamente puede ejecutarse en el procesador al que está asignado, y en cierto modo identificado.

El programa objeto y las variables de los procesos que se llevan a cabo en nodos funcionales se encuentran en su memoria privada.

El bloque de memoria común se utiliza únicamente para intercambio de información y sincronización con otros procesos. El ejecutivo del sistema multiprocesador no actúa sobre los nodos funcionales, ya que son procesadores con una asignación contínua a un proceso. Unicamente se dota al nodo con una serie de funciones, que le permiten realizar primitivas de sincronización y acceder al conjunto de semáforos, colas de mensajes, etc, que se encuentran en la memoria común.

Las ventajas asociadas a los procesadores funcionales son:

- Ausencia de complicados sistemas operativos, con el coste de memoria que suponen. Unicamente se mantienen los mecanismos de sincronización y comunicación con otros nudos.

- La posibilidad de utilizar procesadores heterogéneos, ya que unicamente han de acceder a una memoria común, cuyas únicas características son la longitud de palabra y la velocidad.

- La similitud con las redes de ordenadores, donde se emplea una comunicación serie, enviando mensajes en forma de paquetes con unos protocolos de comunicación. En los nudos funcionales se envían mensajes a través de memoria, con un cierto "protocolo" de primitivas de sincronización.

Como inconvenientes podemos reseñar:

- El más importante a nuestro juicio es que una avería en un procesador especializado normalmente supone la detención de todo el sistema.

- La falta de flexibilidad de un sistema con nudos especializados puede llevar a situaciones tales que un nodo esté saturado, mientras el resto del sistema esta desocupado. Pero esa saturación afecta al vendimiento de todo el sistema.

- Cuando un proceso en ejecución en un nodo funcional queda detenido en una sincronización, empleará comunmente espera activa, desperdiciando tiempo de C.P.U.

Los nodos funcionales se suelen dedicar a tareas específicas, como control numérico, control de procesos, dispositivos de entrada-salida, sistemas de ficheros etc.

3.3 Nodos no especializados

Pueden ejecutar un proceso cualquiera de un conjunto de procesos. Un Sistema Operativo, el asignador de procesadores, se

encarga de establecer dinámicamente cual de esos procesos ha de ejecutar cada nodo no especializado.

- Los procesadores no especializados han de ser idénticos. Debido a la característica de cambiar de un proceso a otro, han de tener accesible la información concerniente a todos ellos. Por lo tanto, las variables locales a los procesos y sus stacks asociados se encuentran en la memoria RAM común. El programa objeto puede también estar en la memoria común, o copiado en la memoria local de cada nodo, para evitar un excesivo uso del arbitro de memoria común.

- Ya que los procesadores son equivalentes, una tarea dada puede ser asignada a cualquier nodo desocupado, y las tareas pueden ser asignadas en caso de fallo de un procesador, produciéndose únicamente una degradación del sistema.

- Los nodos no especializados están especialmente indicados para sistemas en los que cambian constantemente las necesidades de atender uno u otro proceso. Permiten establecer dinámicamente el conjunto de procesos que están en ejecución, para mejorar la respuesta del sistema.(11)

3.4 Sistema operativo

Consta de los módulos de comunicación entre procesos (primitivas de sincronización, monitores) y de un Asignador de procesadores que se encarga de ordenar la ejecución de los procesos. (12)

El estado del sistema está reflejado por las pricridades de los procesos. Estas prioridades se modifican con los requerimientos de vaciado o llenado de colas de mensajes, o con señales externas.

La interrelación del módulo de comunicación con el Asignador es constante, ya que la ejecución de las primitivas de sincronización indican las modificaciones del estado del sistema y producen el bloqueo de procesos.

Por cada proceso, el asignador maneja un descriptivo que almacena el contexto de registros y variables, el estado del proceso e informaciones de gestión, como el nombre, la prioridad...

Según el estado de un proceso, su descriptivo puede estar en la lista de procesos elegidos, o de procesos elegibles, o de procesos bloqueados ante semáforos, monitores o condiciones.

De forma simplificada, la gestión de procesadores sigue este algoritmo en cada procesador.

```
A1: Exclusión entre procesadores
A2: si  proceso elegido = "vacío"
        y nª de elegibles ≠ ∅
        entonces comienzo
                 . buscar proceso elegible Pk
                 . retirar Pk de la lista de procesos elegibles
                 . proceso elegido := Pk
                 . nª de elegibles := nª de elegibles -1
                 end

A3: fin de exclusión
A4: si proceso elegido := "vacío" ir a A1
```

La relación entre los módulos del sistema operativo y con los procesos se establece según la FIGURA 4.

4. CONCLUSIONES

El modelo de sistema multimicroprocesador presentado en este trabajo tiene una estructura totalmente modular que permite la ampliabilidad.

De esta forma se puede dimensionar el sistema para distintas aplicaciones, con el número de procesadores necesarios.

La inclusión de nodos especializados facilita el uso de procesadores heterogeneos que pueden estar mas indicados para funciones específicas.

El conjunto de nodos no especializados mejora la fiabilidad de sistema, y acelera la respuesta a los procesos mas prioritarios.

Los mecanismos de comunicación y sincronización entre procesos proporcionan la posibilidad de expresar de forma concisa en la programación las relaciones que existen entre los procesos en el sistema.

Pensamos que los sistemas multimicroprocesadores pueden sustituir muchos minicomputadores e incluso grandes ordenadores en aplicaciones de control en tiempo real y en sistemas de consulta y modificación de bases de datos, ya que por su flexibilidad permiten incrementar su tamaño con mínimo coste, la capacidad de asignar tareas dinámicamente permite mejorar el rendimiento y el tiempo de respuesta del sistema, y en caso de fallo de procesadores únicamente se produce una progresiva degradación del sistema.

5. BIBLIOGRAFIA

(1) MICHAEL J. FLYNN "Some Computer Organization and their effectiveness". IEEE Transactions on Computer. Vol. C-21, nº 9 Sept. 1972.

(2) GEORGE A. ANDERSON y E. DOUGLAS JENSEN "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples". Computing Surveys Vol. 7 nº 4. Diciembre 1975.

(3) J. ARAMBERRI, I. RAMOS y M. MELLADO "Diseño y desarrollo de un sistema multiprocesador". Primeras jornadas sobre Microprocesadores. Madrid, Octubre 1978.

(4) W. PLUMMER "Asinchronous Arbitros". IEEE Transactions on Computer, Enero 1972.

(5) R. PIERCE "Asinchronous Arbiters Module". IEEE Transactions on Computer. Septiembre 1975.

(6) M. VALERO y E. HERRADA "Estudio de diferentes mecanismos de comunicación entre procesadores de un sistema distribuido multimicroprocesador". Publicación interna del laboratorio de ordenadores de E.T.S. de E.T. Universidad Politécnica de Barcelona. Enero 1978.

(7) G. ADAMS y T. ROLANDER "Design Motivation for multiple processor microcomputer systems". Computer Design. Marzo 1978.

(8) SØE HØJBERG "An Asinchronous Arbiter Resolves Resource Allocation Conflicts on a random priority basis". Computer Design. Agosto 1977.

(9) SØE HØJBERG "One-Step Programable Arbiters for multiprocessors". Computer Design. Abril 1978.

(10) H. WETTSTEIN "The Implementation of Synchronizing Operations in various Environments". Software-Practice and Experience. Vol. 7 pag. 115-126. 1977.

(11) ALAN J. WEISSBERGER "Analysis of Multiple-Microprocessor Systems architectures". Computer Design. Junio 1977.

(12) C. GIRAUD, R. TERRAT "Curso sobre concepción de sistemas informáticos". Bilbao Marzo 1978.

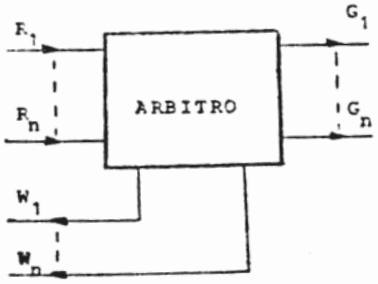


FIGURA 1

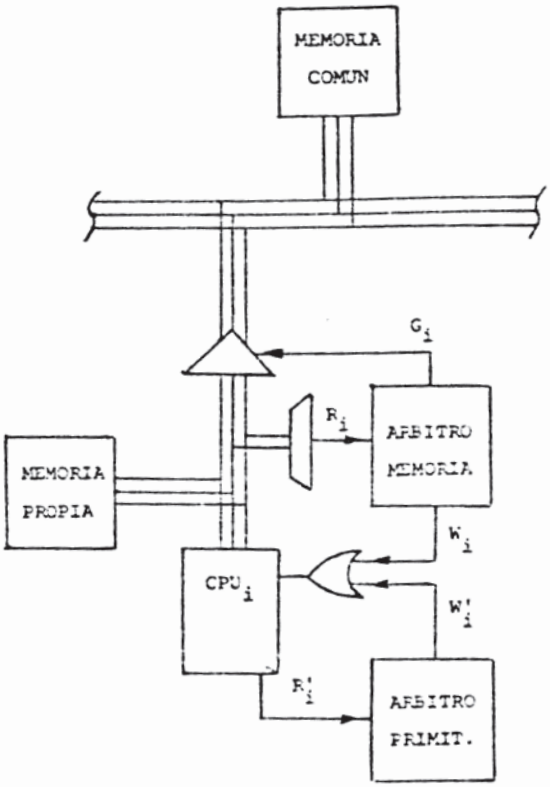


FIGURA 2

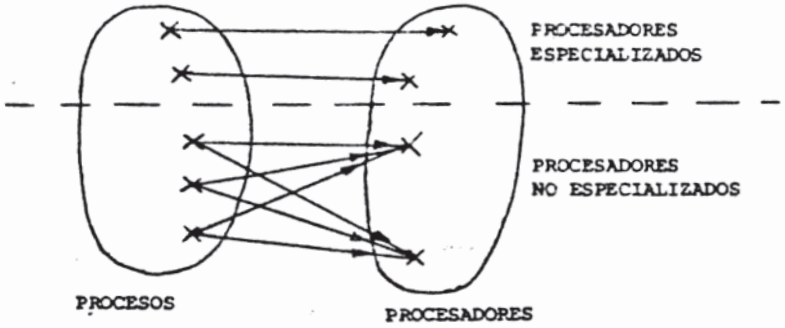


FIGURA 3

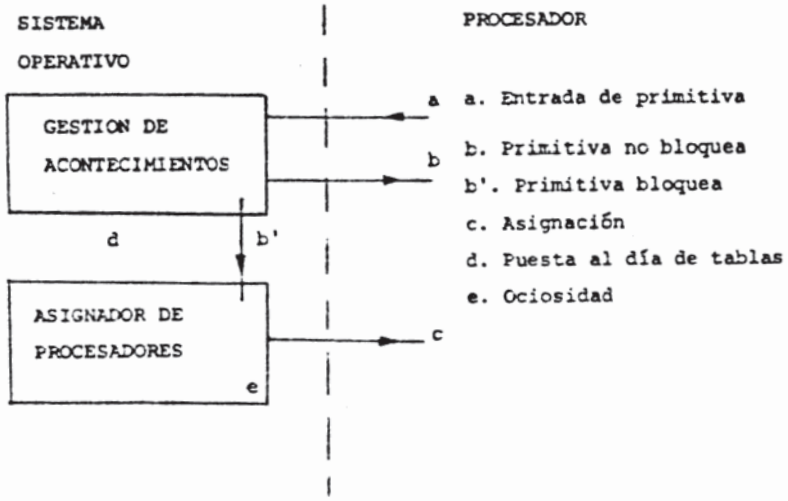


FIGURA 4

4.º CONGRESO
INFORMATICA Y AUTOMATICA
MADRID -16/19 OCTUBRE 1979

SESION
II

SISTEMA DE CONTROL EN TIEMPO REAL DE UNA RED VIARIA

por

Luis de SOPEÑA
Isidro RAMOS S.
UNIVERSIDAD AUTONOMA DE BILBAO
LEJONA-VIZCAYA

RESUMEN

Se describe un sistema basado en un conjunto de miniordenador-microprocesadores para efectuar el control general, en tiempo real, de cada una de las estaciones de peaje de una autopista.

Este sistema de control debe colaborar con el personal humano permitiendo conocer y registrar al instante cada una de las incidencias que se produzcan en la estación.

Sopeña, L., Ramos, I. Sistema de control en tiempo real de una red viaria. Actas del 4º Congreso de Informática y Automática. 1979.

1. INTRODUCCION.

El progresivo abaratamiento y gran difusión alcanzados por la t logía de microprocesadores han permitido que sus aplicaciones se extiende áreas de todo tipo. En concreto, su utilización en los sistemas de peaje las autopistas va a hacer posible un control rápido, eficaz y seguro en t po real.

La finalidad es múltiple: por un lado, dicho control se refiere to a vehículos (su número, clase, posibles infracciones), como al aspecto nómico (recaudación por vehículos, por día), y de forma fundamental al el to humano que interviene en el proceso (cobradores, peajistas); por otro es interesante disponer de datos estadísticos actualizados sobre tráfico estación, rendimiento de las instalaciones, etc.

El sistema consta de un conjunto de microprocesadores y un mini nador por estación de peaje, que controlará lo que en ella ocurra, manter do informando al usuario de todas las incidencias. El propósito de esta c nicación es presentar esta última fase: el Control en tiempo real por par de un miniordenador de un conjunto de microprocesadores en estrella, reco do sus mensajes y peticiones, así como las de un supervisor que, eventual y a través de teclado, atenderá el funcionamiento del sistema.

2. EL PEAJE Y SU CONTROL AUTOMATICO

2.1. Organización del Peaje.

El propósito del sistema de peaje de una autopista es fundament mente el cobro, a los vehículos que circulan por ella, del importe corres diente al recorrido efectuado y a su categoría. La base del sistema la va constituir las estaciones de peaje: en ellas va a realizarse la recogida datos. Cada estación consta de un número variable de calles o vías de ent da y/o salida de la autopista.

En general, los sistema de peaje son de dos tipos: los llamados je cerrado y peaje abierto. En peaje cerrado los vehículos deben pasar po dos estaciones: una de entrada (en la que se distribuyen los tickets y se sifica al vehículo) y otra de salida (en la que el vehículo es nuevamente sificado, de forma manual y automática, y efectúa el pago del importe). C peaje abierto los vehículos son controlados por una única estación de pea no existen tickets.

Dichos tickets disponen, además de una información impresa, de soporte magnético que es codificado a la entrada e interpretado por la ma naria de salida. Como vemos, existe un doble proceso en paralelo para cad vehículo: por un lado, por parte de los peajistas; por otro, por elementa automáticos.

2.2. Organización general del sistema de control.

Se han dispuesto los elementos del sistema en tres niveles de t miento:

1.- En el nivel inferior se encuentran los elementos de detección automática de cada una de las vías: detectores de paso de vehículos, de longitud, altura, semáforos, etc.

2.- En el nivel intermedio, los microprocesaroes (uno por vía) constituyen los órganos de control periférico, llevando a cabo un compejo conjunto de actividades: concentración de señales externas, emisión de señales a la vía, gestión de contadores, recepción y emisión de mensajes al miniordenador.

3.- En el nivel superior, un miniordenador por estación lleva a cabo el control de todo el proceso. Esta es la fase que vamos a desarrollar.

Así pués, son características del sistema:

- La distribución local de funciones (descentralización de tareas) gracias a la independencia y relativa autonomía de los micros.

- El control centralizado por parte del mini, gestor y organizador del proceso. La figura 1 muestra esquemáticamente la disposición del sistema.

El miniordenador, además de atender los mensajes de cada uno de los microprocesadores a los que controla, está conectado a un teletipo y a una perforadora de cinta de papel. A través del teclado del teletipo, un operador podrá realizar determinadas peticiones de información. Además se obtendrá un listado impreso y una cinta perforada donde estarán registradas las incidencias que se produzcan durante el funcionamiento.

3 CARACTERISTICAS DEL SISTEMA DE CONTROL.

3.1. Incidencias importantes a registrar.

El sistema es capaz de identificar y registrar un cierto número de incidercias previamente tipificadas. Estas incidencias son detectadas por el equipo de vía y los micros, y van a afectar a los distintos elementos que intervienen en el peaje:

- por un lado a los peajistas: Control de turnos de trabajo (apertura y cierre de vía) infracciones (no coincidencia de clasificaciones manual y automática, tráfico vía cerrada), etc.

- por otro lado, a los vehículos en tránsito: clasificación automática, número según su clase, infracciones (abandono de vía sin recoger ticket, o con semáforo en rojo,...)

- asímismo, al equipo de vía, controlando acciones tales como el cambio en el depósito de tickets, emisión de recibos, cambio en el cofre de monedas, etc.

- por último, al admitirse el pago mediante una tarjeta de crédito especial, magnéticamente codificada, se debe disponer de una lista de los números que no deban admitirse como válidos; dicha lista se ac-

3.2. Comunicaciones y lenguaje de comunicaciones.

El funcionamiento del sistema exige un intercambio constante de información entre el órgano central y los periféricos.

Podemos distinguir entre:

- Comunicaciones interior-interior: las que tienen lugar entre el miniordenador y los micros. A su vez, las comunicaciones en el sentido mini-micros pueden ser unidireccionales (del mini a un micro) o multidireccionales (del mini a todos los micros).

- Comunicaciones exterior-interior e interior-exterior: aquellas en las que interviene el operador a través del teclado y las que producen un resultado externo, respectivamente.

Por otra parte, en dichos mecanismos de comunicación existen procesos automáticos (que tienen lugar sin la intervención del usuario y son disparados por algún acontecimiento externo: una hora determinada, una incidencia en la vía) y procesos eventuales, de usuario (quien exige al sistema una determinada información).

Para satisfacer los diferentes flujos de información se ha diseñado un lenguaje de comunicaciones que pasamos a describir brevemente.

* Mensajes de microprocesadores a ordenador central: se subdividen en mensajes largos y mensajes cortos (ver figura 2).

- Los mensajes largos constan de un número variable de registros (de 1 a 4) de 21 bytes de longitud cada uno, precedidos por un carácter de "principio de información", y cuyo último byte es de comprobación de que la recepción ha sido correcta.
  La finalidad de estos mensajes es proporcionar al mini la información de las incidencias que se producen en las vías. Su contenido depende del tipo y responde a las circunstancias que han originado su emisión: apertura o cierre de vía, infracciones, estado de la vía, informe horario, transacciones, etc.

- Los mensajes cortos constan solamente de 5 bytes, precedidos por un "principio de información y donde también el último byte sirve como comprobación.
  Estos mensajes tienen tres finalidades:
  a) Pedir información horaria al ordenador central, al principio de la conexión y periódicamente, con objeto de estar sincronizado
  b) Pedir información sobre un número de tarjeta de crédito para saber si se encuentra en la lista negra o no;
  c) Responder a un mensaje del test del mini.
  A los mensajes a) y b) el ordenador debe responder con la información solicitada.
  La recepción de todos estos mensajes es selectiva: el mini acepta o rechaza cada registro recibido para que vuelva a ser emitido según la recepción haya sido correcta o incorrecta.

* Mensajes del ordenador central a microprocesadores: son más breves que los anteriores, por ser su finalidad diferente.

Existe un único mensaje largo de 6 bytes, el llamado "mensaje horario".
El resto son mensajes cortos de un solo caracter y tienen distintas misiones:
- Informar al microprocesador si el registro que ha enviado ha sido correcta o incorrectamente recibido;
- enviar un test a un micro cuando lleva un cierto tiempo sin emitir información;
- responder a un micro diciendo si un número de tarjeta se encuentra o no en lista negra;
- pedir información a un micro sobre su estado (como consecuencia de una orden de teclado).

3.3. Mensajes a través de teclado.

Son los que el supervisor introduce a través del teclado de teletipo. En general su misión es visualizar sobre el teletipo alguna incidencia que ocurra en las vías y de otro modo no sería impresa por el ordenador.

Estos mensajes son: petición de fecha y hora, petición del estado de una vía determinada, y petición de que se impriman las transacciones (pago por vehículo) de la estación controlada.

3.4. Mensajes impresos y perforados:

Estos mensajes van a permitir disponer de una información codificada en cinta perforada cuyo posterior tratamiento hará posible conocer gran cantidad de datos sobre el funcionamiento de la estación; al mismo tiempo, el sistema imprime mensajes referentes a cada incidencia, lo cual permite conocer al instante y de forma continua todo aquello que sucede en la estación.

Se ha tratado de reducir en lo posible la longitud de datos y formatos a imprimir, llevando a la cinta, más rápida, la mayor cantidad de información pero haciendo al mismo tiempo legibles y fácilmente identificables los mensajes escritos.

Los mensajes impresos constan de tres letras iniciales de identificación, y después el número secuencial del mensaje, códigos de estación, y vía, fecha, hora, número de cobrador; a continuación va la información específica para cada mensaje, según el tipo. Todas las incidencias que se produzcan en una vía a lo largo de su funcionamiento van a estar parentizadas por los mensajes de apertura y cierre de la misma.

Los mensajes perforados se corresponden en líneas generales con los impresos. Las diferencias fundamentales son la mayor cantidad de información codificada y naturalmente el formato.

3.5 Funcionamiento del sistema.

Al producirse la conexión de los distintos elementos se lleva a cabo su sincronización; a partir de este momento empieza con regularidad el funcionamiento del sistema. El mini se encuentra inicialmente en un estado que podemos llamar de espera activa: espera la llegada de mensajes procedentes

de los micros y del teclado; y esta espera es activa porque hace constantem
te un test de la hora (ya que debe enviar cada hora un mensaje horario de p
ción de datos a todos los microprocesadores), y también un test del estado
los micros.

La llegada de un registro de un mensaje largo produce el análisi
de dicho registro, la composición del mensaje y la respuesta al micro sobre
la recepción ha sido correcta o no. Al llegar el último registro, tiene lug
además la impresión y perforación del mensaje correspondiente.

Si se trata de un mensaje corto o procedente del teclado, se ana
za y se genera la acción que corresponda de acuerdo con la información pedi

Hay también un estado que puede llamarse de alerta, que tiene lu
cuando uno de los micros no envia información durante un tiempo. Se le envía
un mensaje llamado de test para conocer su estado y saber si la comexión co
núa o bien se ha producido alguna anomalía.

Existe también la posibilidad de funcionamiento en modo local de
los micros: la transición se produce cuando un registro es recibido incorrec
mente por el mini por tres veces seguidas. El micro pasa a local y el mini a
sa al supervisor de dicha transición.

La vuelta a funcionamiento "on line" puede producirse automáticam
te por medio de mensajes horarios o de test por parte del mini, o bien por
iniciativa del micro afectado, en caso de que se quiera realizar una apertur
o cierre de la vía en modo local.

3.6. Implementación.

Se ha dado al sistema de control una estructura modular al objeto
de que sea flexible a posibles cambios dependiendo de las necesidades y de s
adaptación a diferentes estaciones. Con este objetivo se ha distribuido el
sistema de la forma siguiente:

1.- Un Programa supervisor dirige y controla el proceso, encargándos
de:
- aceptar y supervisar las interrupciones procedentes de los term
nales,
- Comprobar la corrección de los registros recibidos,
- gestionar las colas de mensajes de entrada y salida,
- llevar un registro con el estado de cada periférico, avisando d
posibles anormalidades,
- controlar periódicamente a los micros por medio de mensajes, ho
rarios y de test,
- llamar a los subprogramas que lleven a cabo la labor necesaria
en cada momento.

2.- Rutinas de interrupción, encargadas de recibir cada uno de los da
tos procedentes de un microprocesador, así como de controlar la
llegada de órdenes por teclado.

3.- Rutinas de tratamiento de mensajes largos, que toman de ellos los
datos necesarios para imprimirlos o perforarlos.

4 - [illegible] de tratamiento de mensajes cortos, que dan como resultado el listado o perforación, o una respuesta al micro correspondiente, según el tipo.

5 - [illegible] de envío de mensajes de mini a micros, que se encargan de componer el mensaje de acuerdo con las especificaciones establecidas y de remitirlo al micro que corresponda.

6 - [illegible] de perforación y control de perforadoras, que comprueban que no existe ninguna anormalidad en la perforadora que está activa en cada momento, informando, en caso contrario, de tal situación y dirigiendo la perforación hacia la otra unidad. También se encargan de la emisión por cinta de los caracteres a perforar.

4.- CONCLUSIONES

El sistema desarrollado, del que aquí se ha presentado una síntesis permite [illegible] en tiempo real del tráfico que circula por una autopista. Ello tiene aplicaciones muy variadas, de tipo económico, personal, estadístico, que pueden permitir una organización más efectiva del propio peaje.

Los resultados obtenidos demuestran la efectividad del lenguaje de comunicaciones empleado y del método de programación seguido.

Se contempla como una ampliación a las actuales limitaciones del sistema la posible unión de los miniordenadores a un nivel superior con la ayuda de un ordenador más potente, que mejoraría considerablemente la gestión del sistema.

5.- BIBLIOGRAFIA

- [illegible] de Sopeña / "Sistema de control para el peaje de una utopista" / Tesina de Licenciatura Universidad de Bilbao. / junio 77.

- [illegible] de Sopeña / "Une application d'un ensemble mini-micros au [illegible] en temps réel" / Proceedings del 5º Symposium internacional sobre mini y microcomputadores y sus aplicaciones / Zurich / Mayo 1.979.

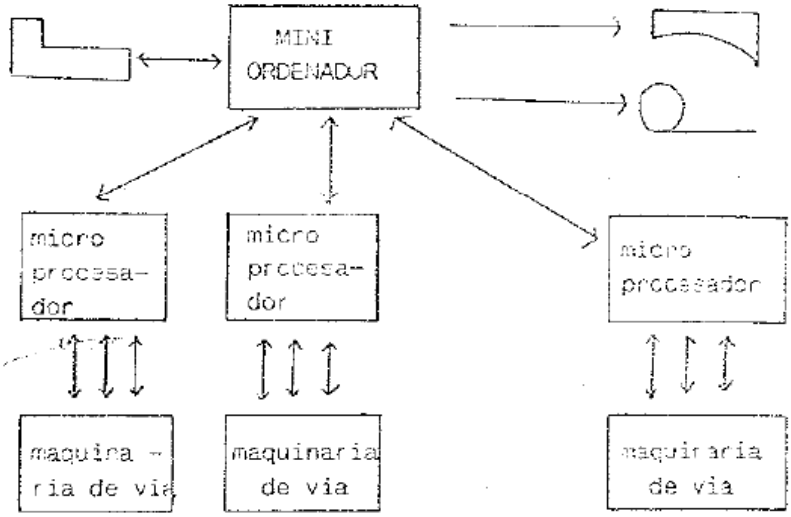


FIGURA - 1

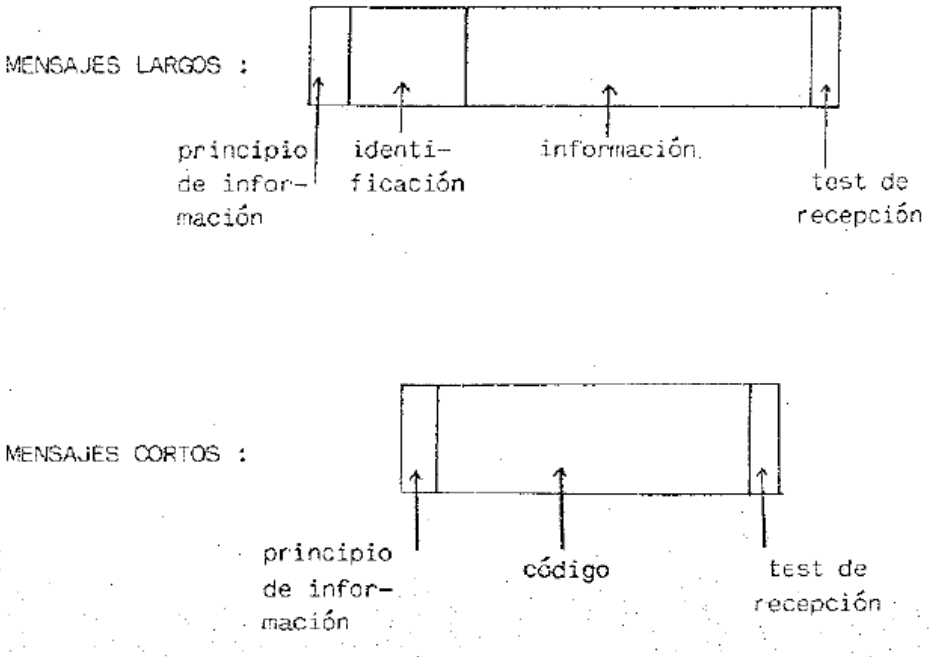


FIGURA - 2

4.º CONGRESO
INFORMATICA Y AUTOMATICA
MADRID -16/19 OCTUBRE 1979

SESION
8

SISTEMA DE PROGRAMACION ESTRUCTURADA CON VUELTA-ATRAS

por

Juan AGUIRRE A.
e
Isidro Ramos S.
UNIVERSIDAD AUTONOMA DE BILBAO
LEJONA-VIZCAYA

RESUMEN

La finalidad de la Programación Estructurada de dar una mayor fiabilidad, legibilidad, rapidez ue diseño, ... se ve abordada por distintas metodologías cuyas implementaciones llevan a metodos de programación conocidos algunos de ellos incluso comercialmente.

En el presente trabajo se da un nuevo paso dentro de este campo, viéndolo desde el punto ue vista de la Teoría de Lenguajes, en particular con el diseño e implementación de un Sistema de Programación Estructurada con vuelta - atrás.

Para ello es fundamental el conocimiento del concepto de E-Gramática y Metaprograma descrito en trabajos anteriores, así como el discerni - miento de la parte estática y dinámica de un programa, entrando en esta última parte la aplicación de la vuelta-atrás por evaluación dinámica de predicados.

Con ello la programación gana en versatilidad a la vez que en convertibilidad y posibilidad de refinamiento de los programas.

8

Aguirre, J., Ramos, I. Sistema de programación estructurada con vuelta atrás. Actas del 4º Congreso de Informática y Automática, Madrid, 1979.

1. INTRODUCCION.

Con el desarrollo actual de la programación el volumen de programas cuya puesta a punto y posteriores modificaciones deben realizarse, ha llegado a tal nivel que obliga a plantear diversas soluciones.

Una de estas soluciones trata de dar la Programación Estructurada, siendo quizás hoy en día, el estructuralismo una de las principales metas a la hora del diseño y puesta a punto de los programas.

No cabe duda que la formación de equipos de trabajo para el desarrollo de programas, fuerza a tomar más en cuenta esta vía.

A su vez la posibilidad de obtener una mayor legibilidad en los programas y modificarlos de forma sencilla da otro aliciente al estudio de la Programación Estructurada.

Es por esto que han ido apareciendo diversos métodos cuyas filosofías no abordamos aquí pero que pueden encontrarse en la bibliografía que de los mismos existe.

Parece interesante resaltar, que todo intento de estructurar un programa en un lenguaje de programación (Fortran, Cobol, ...) se encuentra con las limitaciones propias de dichos lenguajes.

Por ello es conveniente enfrentarse a dichos problemas desde sus cimientos, comenzando el mismo por los aspectos básicos de la teoría de lenguajes, para lo cual definimos el concepto de E-Gramática.

2. DESCRIPCION TEORICA.

2.1. E - Gramática.

Definimos el concepto de E - Gramática (E de estructura) de forma pareja a las gramáticas de tipo 2 de Chomsky.

Una E - Gramática es la sextupla

$$\langle T_t,\ T_n,\ A,\ \Pi',\ \mathcal{P}',\ \mathcal{S} \rangle$$

en donde:

T_t Tratamientos terminales que son nombres de bloque de una sola entrada y una sola salida.

T_n Tratamientos no terminales, usados en la construcción de la estructura que son nombres de bloque de una sola entrada y una sola salida.

Π producciones, conjunto de reglas

$$N \longrightarrow C \qquad \text{donde}$$

$N \in T_n$ y C es una cadena de C_i

tal que $C_i \in (T_t \cup T_n)$ pudiendo ser la cadena vacia

$\mathcal{P}'$ conjunto finito de nombres de predicado

$$\mathcal{P}' = \left\{ P_i^{\,n} \right\} \qquad i \geqslant 1 \qquad n \geqslant 0$$

n indica el grado n-ario

Asocia a cada Π un elemento de $\mathcal{P}'$. Por tanto, el lenguaje que se genera de una E-Gramática será

$$L_E = \left\{ x \mid A \overset{*}{\longrightarrow} x \wedge x \in T_t^{+} \right\} \qquad \text{donde}$$

$$T_t^{+} = T_t^{1} \cup T_t^{2} \cup \cdots \cup T_t^{n} \qquad \text{y}$$

$A \overset{*}{\longrightarrow} x$ es la forma sentencial de la cadena x (Generable desde el axioma A).

- Hasta aquí los pasos seguidos han sido describir lo que es una E - Gramática que a su vez nos ha llevado a ver el lenguaje por ella generado.

Pasemos seguidamente a describir el concepto de Metaprograma.

2.2. Concepto de Metaprograma.

Decimos que L es un Metalenguaje de L' si L puede describir a L'.

Diremos que M es un Metaprograma de P si:
- M puede aceptar la descripción de P.
- M puede generar P.

Dentro de la Programación Estructurada diremos que el:
- Metaprograma:
  - acepta la E-Gramática de todo programa P.
  - acepta la interpretación de la E-Gramática.
  - para toda inicialización genera P.

es decir, el Metaprograma recorre la E-Gramática de P desde el axioma A hasta los elementos terminales T_t de forma descendente, evaluando todos los predicados.

Puesto que se realiza una evaluación de los predicados, analicemos las condiciones de indeterminismo.

$$x \longrightarrow C_i \,/\, 1 \,/\, C_2 \,/\, 2 \,/\, \ldots\ldots\ C_n \,/\, n \,/$$

donde $x \in T_n$ $\quad C_i \in (T_t \cup T_n)^*$ $\quad /i/ \rightarrow P_i \in \mathcal{P}$.

lacondición de indeterminismo nos dice

$$\sum_{i=1}^{n} P_i = \text{cierto si } \exists \text{ uno o más } P_j \text{ tales que}$$

OR

$$P_j = \text{cierto}$$

con lo cual más de un predicado puede ser verdad (en tiempo de ejecución) para una producción dada.

Este indeterminismo de los predicados es tratado por medio de un sistema con vuelta-atrás, por cuya mediación, se evaluan todos y cada uno de los predicados ciertos para cada producción.

Debe resaltarse que no solo los predicados pueden ser causa de un proceso de vuelta-atrás, también los bloques terminales pueden ser origen de un retroceso por evaluación a falso del mismo.

3. SISTEMA IMPLEMENTADO .

3.1. Descripción del Sistema Implementado.

Primeramente se ha definido un metalenguaje estructurado en el que caben destacar las siguientes partes:

* DECLARACION DE OBJETOS.
* TRATAMIENTOS TERMINALES
* TRATAMIENTOS NO TERMINALES
* ESTRUCTURA DEL PROGRAMA
* CONDICIONES
* TEXTO DE TRATAMIENTOS TERMINALES
* FIN

El bloque de DECLARACION DE OBJETOS describe los objetos que van a ser usados en el programa.

El apartado de TRATAMIENTOS TERMINALES describe la lista de los nombres que van a etiquetar los TEXTOS DE TRATAMIENTOS TERMINALES.

El apartado de TRATAMIENTOS NO TERMINALES nos describen los nombres que nos permiten reflejar la estructura del programa.

El apartado ESTRUCTURA DEL PROGRAMA nos define las producciones (de la Gramática). Representa la descripción del flujo de control

(reglas de la E - Gramática). Representa la descripción del flujo de control del programa.

El apartado CONDICIONES nos describen los predicados.

Los TEXTOS DE TRATAMIENTOS TERMINALES nos definen el bloque de acciones a realizar como interpretación al nombre del tratamiento terminal asociado.

FIN indica el final del Metaprograma.

Como resultado se obtiene un programa escribo en Algol. Este lenguaje, es el que debe emplearse para la descripción de los distintos apartados (Declaración de objetos, Condiciones y Texto de tratamientos de terminales.

Dicho programa Algol representa la ventaja de generarse sin errores sintacticos, y la representación del mismo a través del Metaprograma en forma estructurada.

4. CONCLUSIONES.

Con la implementación del Sistema se ha visto la posibilidad de una mayor facilidad en la concepción de programas, y de reflejar los algoritmos en el apartado de Estructura del Programa.

Es de notar la posibilidad de modificar para una misma Estructura del Programa las interpretaciones de los Textos de Tratamientos Terminales (interpretaciones) modificar la Estructura del Programa obteniéndose una mayor versatilidad que en los sistemas convencionales.

Puede, por busqueda exaustiva, buscarse soluciones óptimas a problemas, de forma elemental, así como la introducción de euristicos que permitan eliminar caminos.

5. BIBLIOGRAFIA.

PAIR, C Compilation / Ecole d'été d'Informatique / Neuchatel / sept. 1972

WIRTH, N. Systematic Programming and Introduction / Prentice-Hall / 1973

FOSTER, J.M. Automatic Syntactic Analysis

MOYA, R. Diseño e Implementación en un sistema de Programación Estructurada Universidad de Bilbao / 1977

RAMOS, I. Metaprogramación Estructurada / Congreso Nacional de Informática y Automática / 1975

AGUIRRE, J. Sistema de Programación Estructurada con vuelta-atrás / Universidad de Bilbao / 1973

4.º CONGRESO
INFORMATICA Y AUTOMATICA
MADRID -16 19 OCTUBRE 1979

SESION
6

ALGUNAS SOLUCIONES AL PROBLEMA DEL INTERBLOQUEO DE PROCESOS

por

* I. Ramos
** J.R. Garitagoitia
***J. Aramberri

RESUMEN

Cuando un conjunto de procesos ocupan diversos recursos,de forma que la ejecución de estos no puede continuar, se dice que ha llegado a una situación de interbloqueo entre procesos (Dead lock).

Diversos autores han dado distintos Algoritmos de Detección del Interbloqueo. A partir de este punto se pueden tomar dos caminos: Dejar a un operador que tome las decisiones para salir de esta situación, o construir un Algoritmo de Destrucción que nos de la solución mejor de acuerdo con una función de costos.

Este segundo camino es el seguido en nuestro trabajo desarrollando una serie de algoritmos de orden creciente de optimización que resuelven en cierta forma el problema.

Los Algoritmos basados en uno de detección no solucionan con plena satisfacción el problema, por ello nos replanteamos la situación y presentamos en la segunda parte un Algoritmo de Prevención.

* Centro de Cálculo. Universidad de Valencia
** IKASLAN (ARTECHE) S.A.
*** Facultad de Ciencias Universidad de Bilbao

Ramos, I. Garitagoitia, J.R., Aramberri, J. Algunas soluciones al problema de interbloqueo entre procesos. 4º Congreso de Informática y Automática. 1979.

1. INTRODUCCION

En todo sistema con varios procesos ejecutándose a un mismo tiempo y con recursos compatibles simultáneamente por estos, puede aparecer el problema del interbloqueo entre procesos (Dead-lock).

Los sistemas actuales evolucionan de forma que los recursos son cada vez más compartidos simultáneamente por distintos procesos, lo que hace que el problema del interbloqueo entre estos sea cada vez más importante.

El estudio se ha realizado para el caso en que los recursos sean un conjunto o sistema de ficheros, ya que es en este punto, después del gran desarrollo de las bases de datos, donde el problema es más crítico.

Los resultados obtenidos se pueden generalizar a cualquier tipo de recurso.

El objetivo perseguido es no mermar en gran medida la compatibilidad de un conjunto de operaciones sobre diversos ficheros en un momento dado. Por esta razón excluimos de antemano las soluciones evidentes, tales como ocupar todos los ficheros a usar a la vez, o siguiendo una determinada secuencia, etc. Que en definitiva, imponen restricciones en el uso del sistema.

2. ALGORITMO DE DETECCION

Son evidentes las ventajas de la compartición simultánea de ficheros por un conjunto de procesos, que se traducen básicamente, en relación con los sistemas donde no hay compartición simultánea, en una reducción de los tiempos de espera.

Sin embargo existen operaciones que no pueden ser realizadas simultáneamente sobre un mismo fichero, se dice que son operaciones incompatibles. (1). Será necesario de alguna forma restringir ciertas actividades paralelas produciéndose esperas entre procesos con operaciones incompatibles.

Supongamos $F = \{F_1 \ldots F_m\}$ un conjunto de ficheros simultáneamente compatibles. Sea $O = \{O_1 \ldots O_r\}$ conjunto de operaciones que se pueden realizar sobre los elementos de F. Sea $P = \{P_1 \ldots P_n\}$ conjunto de procesos que comparten los ficheros de F.

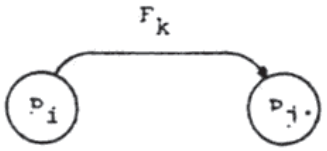
Decimos que el proceso P_i espera al P_j cuando ha pedido la ocupación del fichero F_k para efectuar una operación O_p y esta petición no puede ser aceptada, por el hecho de que el proceso P_j ha pedido la ocupación del mismo fichero para efectuar una operación O_q incompatible con O_p.

Definimos así la relación "P_i espera a P_j por F_k", que puede ser representada por un grafo orientado y "etiquetado" $G=(P,\Gamma,\phi)$ llamado Grafo de Esperas donde:

$P=\{P_i \ldots P_n\}$ es el conjunto de nodos.

ϕ es la función de etiquetado $\phi : \Gamma P_i \in G \longrightarrow F_k \in F$

$\Gamma(F_k)P_i = P_j \Leftrightarrow P_i$ espera a P_j por F_k.



Teorema I (Según J.F. Murphy)

Existe situación de interbloqueo sí y solo si el grafo de esperas posee circuitos

El anterior teorema de fácil demostración nos indica un camino fácil a seguir para la detección de la situación de interbloqueo. Existen diferentes algoritmos que nos dan los circuitos de un grafo.

3. ALGORITMO DE DESTRUCCION

Una de las soluciones que se pueden tomar para salir de una situación de interbloqueo es la construcción de una serie de algoritmos que tengan en cuenta la situación general del sistema y nos den la salida más barata para cada caso.

Hemos desarrollado una serie de teoremas fácilmente convertibles en algoritmos que en general nos demuestran:

1) Siempre podemos salir de una situación de interbloqueo sin cambiar el entorno de ficheros ocupados y esperados por los diferentes procesos.

2) El algoritmo obtenido a partir del teorema resuelve ciertamente la situación.

El caso en que el fichero pueda ser usado directamente por el proceso que lo pide no tiene interés. Estudiaremos el caso en que el fichero no pueda ser ocupado y por lo tanto habrá de ser adicionado un nuevo elemento al grafo de esperas.

Teorema 2

Sea un grafo de esperas $G = \{P,\Gamma,\phi\}$ en la forma definida anteriormente y de cardinal m, sin circuitos. Siempre es posible obtener un grafo $G^* = \{P,\Gamma,\phi\}$ de cardinal m + 1 y tal que $G^* \cap G = G$ sin circuitos.

La demostración de este teorema nos induce el algoritmo de destrucción correspondiente.

Supongamos el elemento nuevo es $(A,B,X) \notin G$ y tal que $A,B \in P$ y $X \in F$ y supongamos $G' = G \cup \{(A,B,X)\}$ tiene circuitos:

$$\exists\,(B,a_1,x_1)\ldots\ldots(a_n,A,x_n) \in G \text{ tal que: } \begin{vmatrix} a_1 \ldots a_n \in P \\ x_1 \ldots x_n \in F \end{vmatrix}$$

$(B,a_1,x_1)\ldots\ldots(a_n,A,x_n)(A,B,X)$ es un ciruito.

Entonces el grafo $G'' = G \cup \{(B,A,X)\}$ no tiene circuitos, pues en caso contrario:

$$\exists\,(A,b_1,z_1)\ldots\ldots(b_r,B,z_r) \in G \text{ tal que: } \begin{vmatrix} b_1 \ldots b_r \in P \\ z_1 \ldots z_r \in F \end{vmatrix}$$

$(A,b_1,z_1)\ldots\ldots(b_r,B,z_r)(B,A,X)$ es un circuito

con lo que se tendría:

$(B,a_1,x_1)\ldots\ldots(a_n,A,x_n)(A,b_1,z_1)\ldots\ldots(b_r,B,z_r) \in G$ es un circuito lo que va en contra de la hipótesis de partida.

Un segundo teorema da lugar a otro algoritmo que aprovecha mejor las posibilidades de ejecución de procesos en paralelo.

Teorema 3

Sea un grafo de esperas $G = \{P,\Gamma,\phi\}$ de cardinal m, sin circuitos. Supongamos el grafo G' de cardinal m + 1 y tal que $G' \cap G = G$, formado por la adición de un elemento nuevo al grafo G. Sea (A,B,X) el elemento nuevo. Supongamos G' posee circuitos, el grafo G^* de cardinal m + 1 formado a partir de G' de la forma siguiente:

$\forall a \in \Gamma$ $a \neq B$ y tal que $B \in \Gamma^+_{G'}(a)$ tenemos

$$\Gamma^+_{G^*}(a) = \Gamma^+_{G'}(a) - \{B\}$$

$$\Gamma^-_{G^*}(a) = \Gamma^-_{G'}(a) + \{B\}$$

$\forall a \in P$ $a \neq B$ y tal que $B \in \Gamma^{+}_{G'}(a)$ tenemos

$$\Gamma^{+}_{G^*}(a) = \Gamma^{+}_{G'}(a) - \{B\}$$

$$\Gamma^{-}_{G^*}(a) = \Gamma^{-}_{G'}(a) + \{B\}$$

$\forall a \in P$ $a \neq B$ y tal que $B \notin \Gamma^{+}_{G'}(a)$ tenemos

$$\Gamma^{+}_{G^*}(a) = \Gamma^{+}_{G'}(a)$$

$$\Gamma^{-}_{G^*}(a) = \Gamma^{-}_{G'}(a)$$

Además: $\Gamma^{+}_{G^*}(B) = \Gamma^{+}_{G'}(B) \cup \Gamma^{-}_{G'}(B)$

$\Gamma^{-}_{G^*}(B) = \emptyset$ (conjunto vacío)

El grafo G* no tiene circuitos.

NOTA.- $\Gamma^{+}_{G^*}(a)$ es "el conjunto de siguientes" del modo a en el grafo G*.

$\Gamma^{-}_{G^*}(a)$ es "el conjunto de precedentes" del modo a en el grafo G*.

El teorema se puede demostrar fácilmente como el anterior por reducción al absurdo.

Es ahora el propio enunciado el que nos induce el algoritmo de destrucción.

Igualmente se puede definir el teorema para el otro nodo del elemento nuevo, la elección de un modo u otro vendrá marcada por el cardinal de los conjuntos $\Gamma^{-}_{G'}(A)$ y $\Gamma^{-}_{G'}(B)$ que nos da idea del número de arcos que hay que modificar en el grafo de esperas (función de coste).

El teorema es directamente aplicable con la condición:

$\forall a_i, a_j \in P$ $a \neq B$ tal que $B \in \Gamma^{+}_{G'}(a)$ con $x_i, x_j \in F$

$$\left.\begin{array}{l}\Gamma(x_i)a_i = B \\ \Gamma(x_j)a_j = B\end{array}\right| \qquad a_i \neq a_j \Rightarrow x_i \neq x_j$$

En la realidad esta condición no siempre se cumple y el algoritmo debe modificarse para admitir esta posibilidad.

El nuevo algoritmo puede llevarnos de una situación de interbloqueo a otra nueva.

Al realizarse la petición de un fichero x ocupado por un proceso a_o por un proceso b_o de forma que caemos en una situación de interbloqueo. Al deshacer este bloqueo original, el proceso an tiene posibilidad de caer en interbloqueo, con otro proceso al menos, si existe un camino que va de a_n a a_o tal que se cumple:

Sean a_n , a_{n-1}, a_{n-2},....., a_2, a_1, a_0 los nodos de ese camino

$\forall a_i \in P$ .... $\exists a'_i$ tal que $x_k \in F$

1) Existen (a_i, a_{i-1}, x_j) y (a'_i, a_{i-1}, x_j) con $x_j = x'_j$
2) Coste de a'_i > coste de a_i.
3) Existe un camino de a'_i - a_i.

El coste total de destrucción de un interbloqueo es la suma de los costes de destrucción de los diferentes interbloqueos que se pueden producir a partir del original. Este número es siempre finito por trarse de conjuntos finitos ya que la aplicación del algoritmo quita al nodo las condiciones necesarias para volver a caer en un nuevo interbloqueo.

Resumiendo, las acciones a seguir son:

- Detección del circuito a la adición de un nuevo elemento en el grafo de esperas.
- Medida del número de interbloqueos que se generaría al aplicar el algoritmo a los nodos involucrados en el circuito.
- Flección del nodo al que se le va a aplicar el algoritmo en función de este número y del cardinal del conjunto de precedentes.
- Aplicación del algoritmo de destrucción al nodo elegido.

4. ALGORITMO DE PREVENCION

Los algoritmos anteriores y todos aquellos que se basan en el algoritmo de detección del interbloqueo son difícilmente aplicables en sistemas de ficheros ya que obligan a echar atrás operaciones ya comenzadas.

Se hace necesaria la definición de una nueva instrucción en nuestro sistema de ficheros para poder construir un algoritmo de prevención. Introducimos así la instrucción "Declarar" ficheros a utilizar por un proceso.

Supongamos $P = \{p_1 \ldots\ldots p_n\}$ conjunto de procesos en ejecución, $F = \{f_1 \ldots\ldots f_m\}$ conjunto de ficheros declarados y $O = \{O_1 \ldots\ldots O_r\}$ conjunto de operaciones posibles sobre los elementos de F. Definimos el grafo $G = \{D, \Gamma\}$ llamado "Grafo de ficheros declarados" en el plano formado por el conjunto producto cartesiano de P y F. Sea $\Phi = P \times F$.

Tenemos $G = \{D, \Gamma\}$

D conjunto de nodos : $D = D_1 \cup D_2 \cup \ldots\ldots \cup D_r$ y $D \subset \Phi$

D_k $(k=1\ldots\ldots r)$ conjunto de $(p_i, f_j) \in \Phi$: p_i ha declarado f_j para realizar sobre él una operación O_k

Diremos que $(p_i, f_j) \in D_w$ y $(p_k, f_r) \in D_v$ son nodos opuestos si $f_j = f_r$ y O_w y O_v son operaciones incompatibles.

Γ función de sucesión: $\Gamma = \{\Gamma_h, \Gamma_v\}$ tal que

sea (p_i, f_j), $(p_k, f_r) \in D$

$\Gamma_h(p_i, f_j) = (p_k, f_r) \iff (p_i, f_j)$ y (p_k, f_r) son nodos opuestos.

$\Gamma_v(p_i, f_j) = (p_k, f_r) \Rightarrow p_i = p_k \in P$

Es evidente: $\Gamma(p_i, f_j) = (p_k, f_r) \Rightarrow \Gamma(p_k, f_r) = (p_i, f_j)$.

El nodo (p_k, f_r) es siguiente del (p_i, f_j) si existe Γ^*

siendo $\Gamma^* = \underbrace{\Gamma_x \cdot \ldots\ldots \cdot \Gamma_v \cdot \Gamma_h \cdot \Gamma_v \cdot \Gamma_h}_{\text{k-veces}}$ $\left|\begin{array}{ll} \Gamma_x = \Gamma_h & k=n^\circ impar \\ \Gamma_x = \Gamma_v & k=n^\circ par \end{array}\right.$

Tal que $\Gamma^*(p_i, f_j) = (p_k, f_r)$.

Un grafo $G = \{D, \Gamma\}$ tiene circuitos si al menos existe un nodo $(p_i, f_j) \in D$ que es siguiente de sí mismo.

El nodo $(p_i, f_j) \in D$ es un Nodo Activo si p_i ocupa f_j

El nodo $(p_i, f_j) \in D$ es un Nodo Pasivo si p_i no ha intentado la ocupación de f_j.

El nodo $(p_i, f_j) \in D$ es un Nodo en Espera si p_i ha intentado la ocupación de f_j y esta aún no ha podido realizarse.

Los nodos (p_i, f_j) y $(p_k, f_r) \in D$ son Nodos del Mismo Proceso si $p_i = p_k$.

Debido a la incompatibilidad de operaciones solo uno de dos nodos opuestos puede ser nodo activo.

El Teorema siguiente nos da la condición necesaria y suficiente para que un circuito en el grafo de ficheros declarados se traduzca en un circuito en el grafo de esperas y por lo tanto en una situación de interbloqueo.

Teorema 4

Sea $C = \{D', \Gamma\}$ un circuito del grafo de ficheros declarados $G = \{D, \Gamma\}$, $D' \subset D$ y $C \subset G$.

Existe circuito en el grafo de esperas GF construido con los mismos conjuntos F, P en la forma ya definida, sí y solo si:

$\nexists (p_i, f_j)$, (p_k, f_j) , $(p_i, f_j) \in D'$ se cumple:

(p_i, f_j) es un nodo activo
(p_i, f_j) es un nodo en espera
(p_i, f_k) es un nodo en espera

Los circuitos del grafo G existen desde la declaración de los ficheros independientemente del orden de ocupación de estos. El grafo G contiene todos los circuitos que pueden implicar situaciones de interbloqueo, cuando las condiciones de existencia se cumplan, desde el momento de la declaración de los ficheros.

El algoritmo construido asegura que en un circuito el conjunto de nodos de un determinado proceso al menos están ambos activos o no, rompiendo una de las condiciones de existencia.

Este algoritmo permite por otra parte que se realicen operaciones compatibles en paralelo ya que estas no producen nodos opuestos y por lo tanto circuitos.

En la figura 2 se representa el comportamiento del algoritmo de prevención en el caso que

F = {A,B,C,D}
P = {1,2}
O = { lectura, escritura} Lect y Esc suponemos son incompatibles.

Supongamos:

El proceso 1 declara A,B,C,D en escritura, lectura, escritura, lectura respectivamente.

El proceso 2 declara B,C,D en lectura, lectura, escritura.

La fig. 1 muestra el Grafo de ficheros Declarados correspondiente, en el que se puede observar la existencia de un circuito.

Supongamos que la ocupación de los ficheros se produce en la forma siguiente: (fig. 2)

El proceso 1 en t_1 ocupa A, en t_2 ocupa B, en t_3 ocupa C, en t_4 ocupa D en t_5 libera D y en t_6 libera c.

El proceso 2 ve todos los ficheros libres hasta el momento t_3 en el cual ve los ficheros C y D ocupados cuando en realidad solo lo está el C. Por otra parte el proceso 2 podrá ocupar en cualquier momento el fichero B así como el D a partir de t_5.

BIBLIOGRAFIA

(1) "Une solution au probleme du verrou mortel en partage simultane de fichiers".
Tesis de la Universidad de Paris VI (1.974).

(2) J.E. Murphy "Ressource allocation with interlock detection in a multi-task system".
AFIPS FJCC - Volumen 33-2 (1.968).

(3) A.N. Habermann "Prevention of System Deadlocks".
Communications ACM - Volumen 12 n° 7 (1.969).

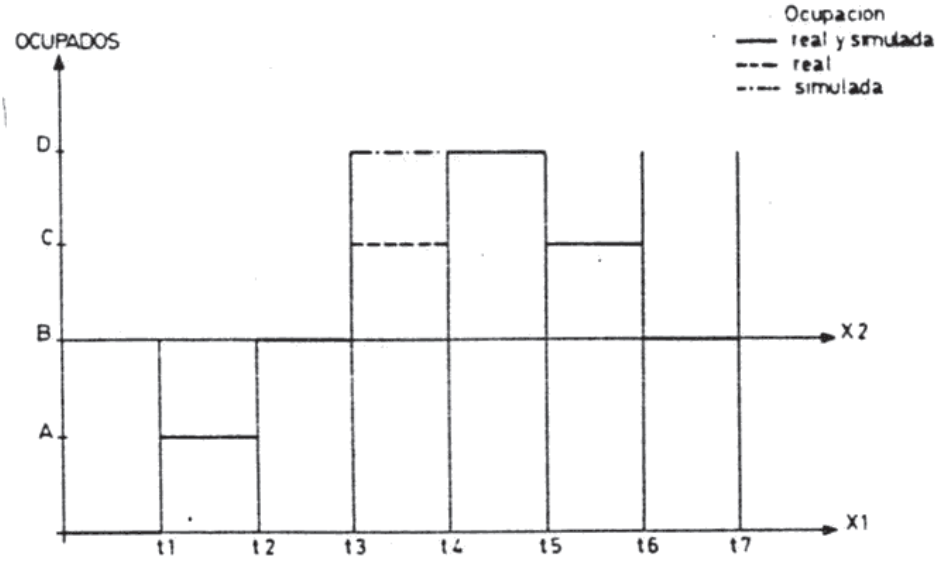

OCUPADOS
Ocupacion
real y simulada
real
simulada
D
C
B
A
X2
X1
t1
t2
t3
t4
t5
t6
t7

FIG. 1

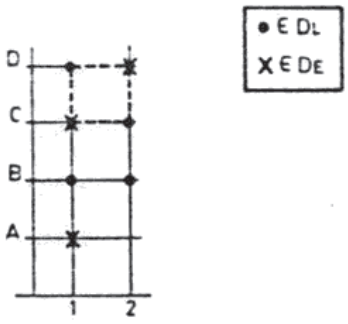

• ∈ DL
X ∈ DE
D
C
B
A
1
2

FIG. 2

4.º CONGRESO
INFORMATICA Y AUTOMATICA
MADRID -16/19 OCTUBRE 1979

SESION
11

A SOFTWARE CHRONOLOGICAL STATE CONTROLLER

por

J.M. Guaresti, J.R. Garitagoitia e
I. Ramos.
Departamento de Automática e Informática
Universidad de Bilbao

RESUMEN

A chronological state controller has been implemented on a MCS-80 microcomputer. It is a program that accepts as data the state of a set of input ports and reports its changes along the time. It has a great flexibility because its software nature and allows easily future enhancements . Comparatively to the hardware implemented state controllers it is cheaper and smaller. Between its features we remark the important number of messages it is able to accept and the possibility of dialog with an operator using a simple language.

Guaresti, J.M., Garitagoitia, J.R. y Ramos I.A software chronological state controller. Actas del 4º Congreso de Informática y Automática. 1979.

1. INTRODUCTION

In the electrical networks stations and substations it is important to know the changes in the states of different switches, protections, alarms ... but it is still more important to know the relative time sequence of these changes since the perturbation that motivates them.

Up the date the usual state controllers had been implemented using hardware techniques, but the arrival of microprocessors and the increasing need of flexibility in their use have led to change to software techniques in their implementation.

The state controller implemented detects and reports the changes along the time of the states of many input ports, and in this way it is possible to analyse and interprete the causes that have produced these changes.

The functions performed are:

- Dynamic storage allocation of the output messages FIFO LIST.
- Test of the ports status.
- Parsing and interpretation of the operator's orders.
- Output of the messages.

2. CONTROL LANGUAGE

A teletype is used as input device to give the operator's orders and to output the state controller's messages.

The orders given by the operator are of the following kinds:

- Beginning of operation orden.
- Report of the actual state of ports order. Its general format is
  R ((Type,) (nº initial/nºfinal,) (state)) CR

The parentheses indicate optional fields.

- Disable of ports order. The ports disabled are not considered in the process.
- Enable of ports order. Is the counter part of the order below.
- Stop of process order.
- Operator's interrupt order. Allows the absolute control of the process by the operator.
- Clock set and display orders.

The messages given by the state controller are of two kinds:

- Messages reporting a change of state in a port.

- Message reporting the state of a port (result of a Report request).

The parser detects the syntax errors.

The language is described by the following Type-3 grammar given in BNF (the terminal symbols are underlined, the nonterminal symbols appear as left parts and the axiom is the first left part):

<orden> : : = R <Resto R> | A <Resto A> | I <Resto I> | H <Resto H> | C <Resto C> | F CR | P CR

<Resto R> : : = ¥ <Resto R> | CR | ↓ CR | = CR | letra <Resto 1R> | dd <Resto 5R>

<Resto 1R> : : = letra <Resto 2R>

<Resto 2R> : : = letra <Resto 3R>

<Resto 3R> : : = CR | , <Resto 4R>

<Resto 4R> : : = dd <Resto 5R> | ↓ CR | = CR

<Resto 5R> : : = dd <Resto 5R> | / <Resto 6R>

<Resto 6R> : : = dd <Resto 7R>

<Resto 7R> : : = dd <Resto 7R> | CR | , <Resto 8R>

<Resto 8R> : : = ↓ CR | = CR

<Resto A> : : = ¥ <Resto A> | ↓ CR | = CR | letra <Resto 1R> | dd <Resto 1A>

<Resto 1A> : : = dd <Resto 1A> | CR | , <Resto 2A> | / <Resto 6R>

<Resto 2A> : : = dd <Resto 1A>

<Resto I> : : = ¥ <Resto I> | letra <Resto 1R> | dd <Resto 1I>

<Resto 1I> : : = dd <Resto 1I> | CR | , <Resto 2I> | / <Resto 6R> | ↓ <Resto 3I> | = <Resto 3I>

<Resto 2I> : : = dd <Resto 1I>

<Resto 3I> : : = CR | , <Resto 2I>

<Resto H> : : = ¥ <Resto H> | CR

<Resto C> : : = ⊬ <Resto C> | CR | dd <Resto 1C>

<Resto 1C> : : = dd <Resto 2C>

<Resto 2C> : : = ⊬ <Resto 3C> | CR

<Resto 3C> : : = dd <Resto 4C>

<Resto 4C> : : = dd <Resto 5C>

<Resto 5C> : : = ⊬ <Resto 6C> | CR

<Resto 6C> : : = dd <Resto 7C>

<Resto 7C> : : = dd <Resto 8C>

<Resto 8C> : : = ⊬ <Resto 9C> | CR

<Resto 9C> : : = dd <Resto 10C>

<Resto 10C> : : = dd <Resto 11C>

<Resto 11C> : : = ⊬ <Resto 12C> | CR

<Resto 12C> : : = dd <Resto 13C>

<Resto 13C> : : = dd <Resto 14C>

<Resto 14C> : : = ⊬ <Resto 15C> | CR

<Resto 15C> : : = dd <Resto 16C>

<Resto 16C> : : = dd <Resto 17C>

<Resto 17C> : : = letra <Resto 18C> | CR

<Resto 18C> : : = dd <Resto 19C>

<Resto 19C> : : = dd <Resto 20C>

<Resto 20C> : : = dd <Resto 21C>

<Resto 21C> : : = CR

Note: We consider letra and dd as terminal symbols.

Obviously:

<letra> : : = A | B | C | ... | Z

dd : : = 0 | 1 | 2 | ... | 9

3. IMPLEMENTATION

The implemented program has two parts:

- Initialitation part: states the initial values of different parameters and stores in the port-state-vector the initial status of the ports.

- Body: with three independent processes and monitoring:
  - Operator's process: routines of input, parse and execution of the operator's orders.
  - Output process: routines of output and management of the output buffer.
  - Test of the ports-state-process: routine of test.

A clarifying work on the subjet is reference (2).

The process priorities are showed by the flow-chart of the fig. 1.

The input/output is performed in a character by character way.

A message is complete when the characters CR/LF are typed.

The end of a order starts the initialitation phase that has three

steps:

-State vector initialitation: a first reading loop is performed to set the state vector to the initial state of the ports. A type is assigned to each port following the port type table (given at the state-controller generation).

- Clock variables initialitation: these variables are initialized following the operator's time specifications.

-Pointers, variables ... initialitation.

The state vector structure is the following:(see fig. 2).

The priorities of the processes are, in decreassing order, the following:

1.- Operator's interrupt: The operator has control, the output of messages is temporarily suspended but the test-loop continues.

2.- Message output in the output buffer.

3.- Operator's orders parse (in the input buffer).

4.- Output buffer loading with "change of state" messages.

5.- Output buffer loading with "report of state" messages.

6.- Operator's orders input.

If none of the previous items exists, the test loop is performed (cycle time 2.5 mseg per 1000 ports).

The routines are the following:

- Test of state subroutines: perform for each enabled port the following actions:

. Read the actual state.

. Compare with previous one; if different:

- update the port byte in the port-state-vector.
- set up the change of state vector.
- store the message in the output queue.

The corresponding flow-chart is in the Fig. 3.

The output queue is a FIFO circular list.

The 'change of state' vector is formed by the actual status byte and the clock variables (time).

- Input/Output subroutines : perform the buffer management asyncronously the test loop.

The output messages are of two kinds:

- Change of state message : the format is:

HH MM SS MSMS : nº PORT STATE TYPE CR/LF

HH = hour, MM = minutes, SS = seconds, MSMS = miliseconds.

This message is generated following the actions:

. Clock variables and nº of port conversion to ASCII characters

. Store the external type representation in the output buffer.

. State of port conversion to ASCII characters.

- Report messages: with the following format:

HH RESUMEN : nº PORT STATE TYPE E/D CR/LF

E/D = enabled/disabled.

The other fields are self explanatorys.

- Order's parser : is a program data driven because the efficiency of this approacer. The recognized language is given by the previous Type-3 grammar. The arrival at the final states of the corresponding associate automaton produces the order's execution.(See Fig. 4)

4. ENHANCEMENTS.

Because its software nature the state-controller allows easily, different enhancements.

The first one considered is to establish priorities between the ports to report its change of state (an alarm may be more prioritary than a switch). This implies in our design to change the FIFO output list to a multi-level FIFO list.

The second one is the introduction of a interrupt system that minimizes the test loop tp the interrupt routine.

The third one is to use a multiprocessor system, this implies in our design the introduction of synchronizing primitives.

These enhancements will reduce the cycle time that in spite of the software character of the state controller implemented compares favorably with others state controllers implemented following hardware techniques.

5. CONCLUSIONS

1st.- The state controller implemented following software techniques is more flexible than the state controllers built by hardware techniques without penalizing excessivily the cycle-time characteristics (max. cycle-time 5 mseg, min. cycle-time 2.5 mseg).

2nd.- Using asynchronous processes the cycle-time is not increased by the output decives rates.

3rd.- The storage capability is bigger and easier expandable than in the hardware state controllers.

4th.- The operator interacts really with the system via a simple language.

5th.- The state controller implemented is cheaper and smaller than the hardware implemented ones.

6th.- Different enhancements are possible in a straigth-way form.

6. REFERENCES.

Arteche, Registrador Cronológico de Estados.

Brinch Hansen, A Real-time Scheduler (California Institute of Technology, 1975).

Intel 8Ø8Ø, "Assembly Language Programming Manual".

Intellec 8/MOD 8Ø, "Operators Manual".

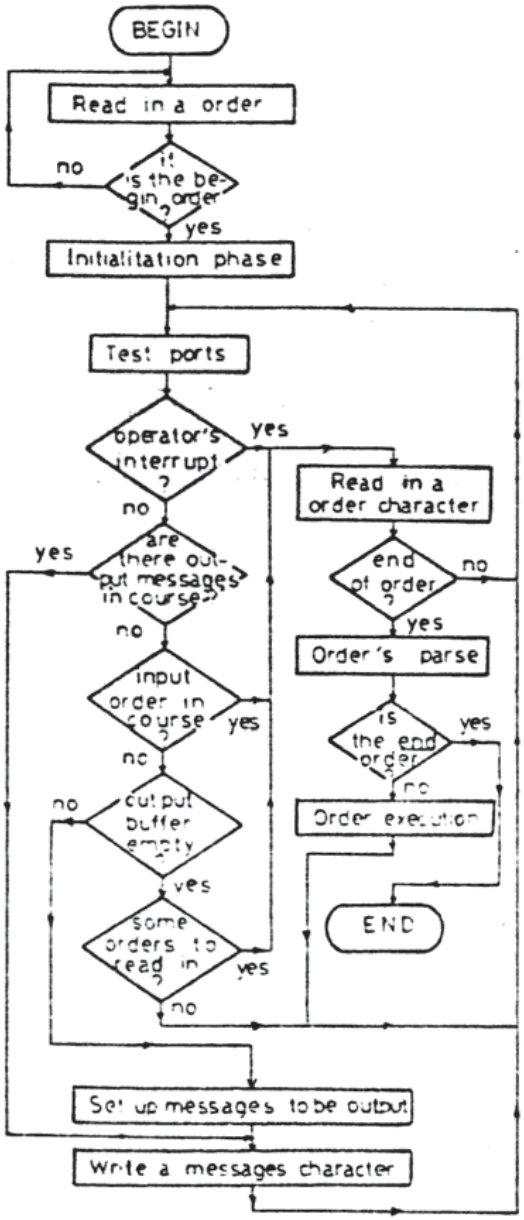

BEGIN
Read in a order
it is the begin order ?
no
yes
Initialitation phase
Test ports
operator's interrupt ?
yes
no
Read in a order character
are there output messages in course ?
yes
no
end of order ?
no
yes
Order's parse
input order in course ?
yes
no
is the end order ?
yes
no
output buffer empty ?
no
yes
Order execution
END
some orders to read in ?
yes
no
Set up messages to be output
Write a messages character

Fig. 1

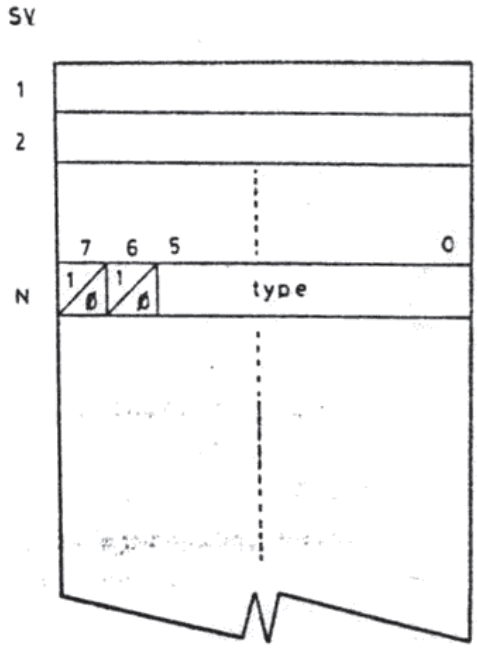


Fig. 2

bit 7: enable/disable port bit.

bit 6: status bit.

bits 5-0: type bits (64 posible types).

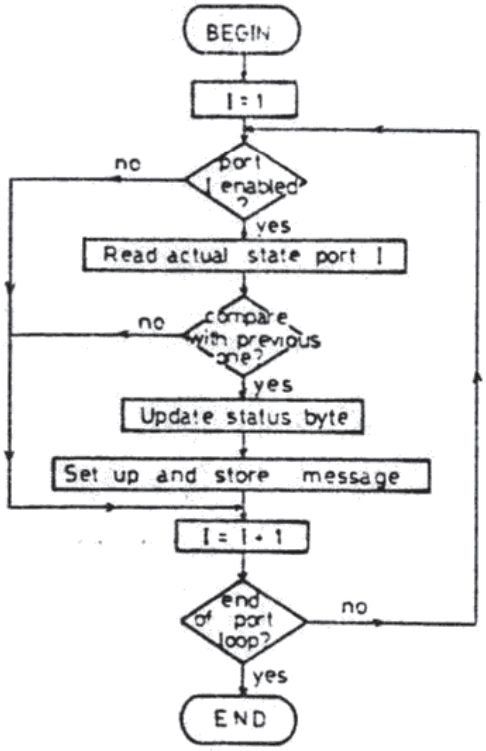


Fig. 3

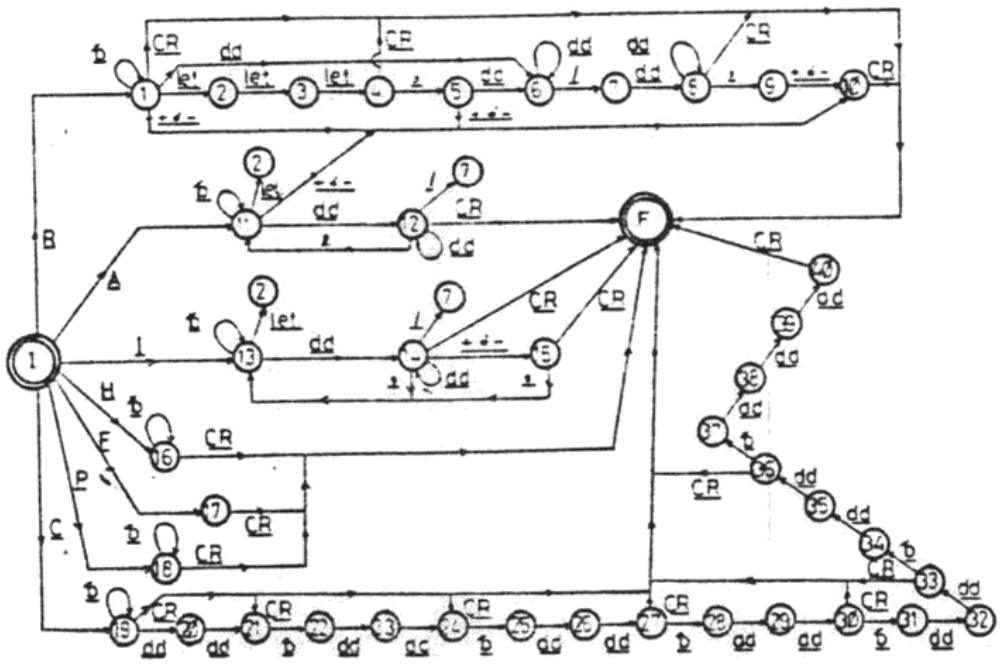


Fig. 4

CALCULADORA DE MESA: UN PROYECTO DE CLASE SOBRE TEORIA DE AUTOMATAS Y LENGUAJES FORMALES.

por: Isidro Ramos Salavert

Departamento Automática e Informática.
Universidad de Bilbao

En el trabajo se expone un proyecto de clase sobre teoría de lenguajes y autómatas que consiste en la realización de una calculadora evolucionada de mesa (acepta notación infija) usando un miniordenador. El proyecto se realiza en Fortran.

Dado el contexto docente (alumnos de 2º siclo) y las restricciones temporales, el proyecto permite introducir en forma clara y operativa, conceptos que de otra forma resultan difíciles.

1.- INTRODUCCION.

La enseñanza de la Informática y en particular de la teoria de Lenguajes Formáles y Autómatas, en contextos docentes con una clara polarización hacia el uso de esta Disciplina como herramienta, plantea dificultades que podemos resumir en los siguientes puntos:

- Al hacer hincapié en los aspectos formales(demostración de los teoremas básicos ...) se produce una ruptura con el alumnado que ni por formación, ni por intención se sientete motivado a este tipo de curso.
- Es discutible el interés de un planteamiento formal en la exposición de esta disciplina en ese contexto.
- Es impensable el abandonar la enseñanza de estos tópicos que luego le resultarían imprescindibles en aplicaciones como el diseño de Sistemas de Supervisión, controladores programables... o sea en aplicaciones en las que medie un lenguaje(por elemental que este sea) y en general en todos los campos de aplicación de la programación conducida por Datos(Data Driven Programming).
- Ausencia de Textos adecuados....

La ausencia de un planteamiento sistemático unida a las dificultades expuestas anteriormente, hacen que los proyectos o trabajos prácticos(1)sean la fórma usual de introducir algunos conceptos que así pierden su carácter formalista y a veces, esotérico.

En este trabajo resumimos lo que constituye un proyecto de clase sobre la materia. Este proyecto ha sido realizado con éxito por alumnos con una preparación mínima en Informática.

2.- EXPOSICION DEL PROYECTO.

El proyecto se desarrolla con alumnos de segundo ciclo (4º curso) de

Ramos, I. Calculadora de mesa. Un proyecto de clase sobre teoría de Lenguajes y Autómatas. Revista de Informática y Automática. 1979.

$F = (k_o)$ conjunto de estados finales.

La función $\delta(x,y,z)$, de transición entre estados, con

$$x \in K$$
$$y \in T^*$$
$$z \in T^*$$

se define como

$\delta(k_o, a\alpha, \beta b)$= si $b \in T$ entonces si $a = b$ entonces (k_o, α, β) sino error sino $(k_o, a\alpha, \beta\tilde{\gamma})$.

Nota: b ocupa la cumbre de la pila.

Donde $\tilde{\gamma}$ es el simétrico especular de la parte derecha de una regla:
b::= γ que se elige de acuerdo con las condiciones de determinismo de D. E. KNUTH (2).

Esta solución, discutida en clase fue tomada por algunos equipos.

2ª). Soluciones ad-hoc.

, Entre ellas y por su simplicidad transcribimos una posible.

Se usa un autómata de estados finitos determinista:

A1 = (K, k_o, F, T, δ).
K = (1, 2) dos estados.

Estado 1: Si acepta una letra pasa al segundo estado o acepta un paréntesis de apertura e incrementa un contador (inicializado a cero) y permanece en el estado 1.

Estado 2: Acepta un operador y pasa al estado 1 o acepta un paréntesis de cierre, disminuye el contador en una unidad (si se hace negativo se emite un error) y pasa al estado 2 o acepta el final de la frase.

k_o = Estado 1

F = {Estado 2}

T = (letra, operador, (,))

δ : K x T → K la función de transición entre estados viene dada por el diagrama de transición entre estados al que se le asocian las acciones semánticas específicas (programación conducida por los datos).

Obviamente el proceso termina con la aceptación de un carácter fin de frase: (" ").

ETAPA2 Paso a notación polaca inversa.

Después de verificada la corrección de las frases se les comentan en clase los métodos de precedencia mediante ejemplos y se pide que formalicen un autómata a pila traductor que realice el paso a notación polaca inversa en la forma clásica y que damos a continuación.

A2 = (T, r, G, K, k_o, F, δ, ϕ).

Con:

T = {A, ..., Z, +, *, -, /, (,), ↑,)

r = (A, ..., Z, +, *, -, /, ↑,)

σ = carácter inicial en la pila

K = (k_o) conjunto de estados formado por un solo estado k_o.

$k_o = k_o$

$F = \{k_o\}$

$\delta(k_o, x\alpha, \beta y)$ =
- si x = '(' e y = ')' entonces (k_o, α, β)
- sino si x = '⊢' e y = '⊣' entonces fin
- sino
- si prioridad (y) > prioridad (x)
- entonces $(k_o, x\alpha, \beta)$
- sino $(k_o, \alpha, \beta y\ x)$;

con $k_o \in K$

$x \in T$, $\alpha \in T^*$, $y \in r$, $\beta \in r^*$.

y prioridad (x) indica la prioridad del carácter x.

$\phi(k_o, x\alpha, \beta y)$ = si x = '(' e y = ')' entonces ε
- sino si x = '⊢' e y = '⊣' entonces ε
- sino
- si prioridad (y) > prioridad (x) entonces ε
- sino x;

siendo ε la cadena vacía.

Que es la función de salida del autómata. Este autómata se sugiere en clase comentando unicamente la tabla de

na Facultad de Ciencias Físicas y des pues de 3 meses de enseñanza a razón de 2 horas diarias.

nicialmente los alumnos desconocen otalmente los tópicos de la Informáti ca y tienen la formación típica dada por el primer ciclo de una Facultad de Ciencias.

Las enseñanzas que reciben están con cebidas de forma ascendente en cuan to al planteamiento de los temas(delo particular a lo general) aunque en la resolución de problemas y ejercicios se use una aproximación descendente clásica.

Las materias que se imparten en estos tres meses son las siguientes:

- Aprendizaje de un miniordenador concreto.
- Lenguaje máquina de un miniordenador concreto. Ejercicios.
- Lenguaje ensamblador de un miniordenador concreto. Ejercicios.

Esta primera etapa se realiza en 15-20 dias. A continuación y solapados en el tiempo se exponen los tópicos de:

- Estructuras de Datos.
- Estructura de Calculadoras.
- Fortran.Ejercicios numéricos y no numéricos.

Con una duración de dos meses.

Conocido ya el concepto de ordenador, dato y lenguaje se aborda el tema de:

- Teoria de Lenguajes y Autómatas y es dentro de esta parte en la que se plantea el proyecto de la calculadora de mesa.

Agrupados los alumnos en grupos de 3 como máximo, se les pide que elaboren un programa que evalue expresiones aritméticas que permitan paréntesis, blancos, operandos (variables), con identificadores de una sola letra (para obviar el problema del análisis lexicográfico)...

Se les sugiere como gramática indicativa expresada en B N F la siguiente:

⟨E⟩ :: = ⟨E⟩ ± ⟨T⟩ | ⟨E⟩ ∓ ⟨T⟩ | ⟨T⟩

⟨T⟩ :: = ⟨T⟩ * ⟨P⟩ | ⟨T⟩ / ⟨P⟩ | ⟨P⟩

⟨P⟩ :: = ⟨v⟩ | [⟨E⟩]

⟨v⟩ :: = A | B | ... | Z

que describe expresiones aritméticas bien parentizadas, pero que tiene el defecto de no ser determinista (concepto de gramática determinista LL (1) explicado en clase) por lo que no se le puede asociar el autómata equivalente que de forma descendente y predictiva nos indica si una frase es correcta o no.

Los alumnos buscan una gramática débilmente equivalente a la dada. Entre las distintas soluciones encontradas se dió la siguiente:

⟨E⟩ :: = ⟨L⟩ ⟨R1⟩ | [⟨E⟩] ⟨R1⟩

⟨R1⟩ :: = ⟨O⟩ ⟨E⟩ | ε

⟨L⟩ :: = A | B | ... | Z

⟨O⟩ :: = ± | ∓ | * | /

gramática que se amplía trivialmente a la potenciación sin más que incluir el operador " ↑ ".

Los elementos de la gramática son pues:

G = (T, N, A, ::=)

T = (A, ..., Z, + , -, * , /, ↑ ,(,))

N = (⟨E⟩ , ⟨R⟩ , ⟨L⟩ , ⟨O⟩).

A = ⟨E⟩

y las reglas son las indicadas anteriormente.

ETAPA1 Reconocimiento de las frases.

Varias soluciones son posibles para el reconocimiento de las frases (expresiones aritmeticas).

1ª). Solución clásica del autómata a pila equivalente:

$A = (T, \Gamma, \sigma, K, k_o, F, \delta)$

Siendo,

T = alfabeto de la cinta de entrada que coincide con el alfabeto terminal de la gramática.

Γ = T U N (siendo N el alfabeto no terminal de la G)

$\sigma \in \Gamma$ símbolo inicial de la pila y que coincide con el axioma de la gramática.

K = conjunto de estados que se reduce a un único estado k_o

k_o = estado único.

prioridades y realizando en la pizarra varios ejemplos manualmente de paso a notación polaca inversa (NPI) siguiendo este método.

Los equipos formalizan el algoritmo en forma del autómata A2 (o equivalente).

Y en particular resuelven el problema de eliminación de paréntesis y fin.

ETAPA3 Evaluación de las expresiones.

Análogamente a la ETAPA2 se comenta en clase mediante ejemplos el intérprete evaluador de expresiones escritas en NPI.

Se formaliza mediante un autómata a pila:
$A3 = (T, \Gamma, \sigma, K, k_o, F, \delta)$

con $T = (A, \ldots\ldots, Z, +, x, -, /, \uparrow, \dashv)$.
$\Gamma = (A, \ldots\ldots, Z, \mathbb{N})$.

$\mathbb{N}$ = conjunto de números naturales, relacionado con el almacenamiento de resultados intermedios en variables temporales (son los índices de un vector de variables temporales).

$\sigma = \vdash$
$K = (k_o)$

k_o
$F = (k_o)$
$\delta(k_o, x\alpha, \beta yz) =$ si $x \in$ operadores entonces $(k_o, \alpha, \beta n)(1)$
sino $(k_o, \alpha, \beta yzx)$;

$k_o \in K$, $x \in T$, $\alpha \in T^*$, $\beta \in T^*$, $y \in \Gamma$, $z \in \Gamma$

siendo $n \in \mathbb{N}$ el valor del índice que nos apunta a la primera posición vacía del vector de variables temporales.

Obviamente además de estas transiciones entre estados en (1) se realizan operaciones del tipo

```
TEMP(n):= y * z;
      n = n + 1;
```

y análogas.

Los tres autómatas anteriores constituyen la base del proyecto que se programa en FORTRAN (único lenguaje que conocen los alumnos).

Hay que añadir los siguientes aspectos concretos (caso del proyecto presentado):

1) Se dispone de un buffer de lectura (I BUF). Gestionado por dos punteros de entrada sobre el que se lee carácter a carácter la expresión. Un buffer de salida ISAL y una pila IPIL.

2) Se realiza la eliminación de blancos caso de que existan y se detectan los caracteres de control ' @ ' y '$' (correcciones efectuadas) con el siguiente significado:

- ' @ ' caso de aparecer este símbolo en IBUF se elimina el carácter previo (carácter de borrado): disminución del puntero de salida: ISAL.
- 'b': se elimina incrementando el contador de entrada IPT.
- '$': iniciar el proceso nuevamente. Caso de aparecer este símbolo el programa pasa al punto inicial quedando en disposición de volver a leer.

3) El carácter '⊣' se representa por -1.

4) Todo error provoca una reinicialización del proceso abandonando las operaciones previas. El error es único.

5) Una vez realizado el paso a NPI se solicitan los valores de las variables mediante preguntas

variable =

obviamente si una variable aparece más de una vez en la expresión su valor solo es solicitado una vez (Tabla de variables incipientes: ILET).

6) Se opera la expresión (A3) y se escribe el resultado. Se reinicia el proceso.

3.- CONCLUSIONES.

Después de finalizado el proyecto los equipos entregan un documento con la descripción detallada de la solución propuesta y del programa realizado.

Creemos que el grado de dificultad del problema propuesto es grande pero gracias a una asistencia continuada y

no imperativa por parte del profesor se llega a:

- Resolver felizmente el proyecto con un porcentaje de proyectos en funcionamiento muy elevado.

- Los conceptos de lenguaje, autómata ... aparecen claros y próximos.

- La talla del proyecto permite que sea resuelto en unas cuatro semanas.

Se sigue una política de puerta abierta en el uso del miniordenador.

En un proyecto clásico de Programación no numérica con manejo de caracteres, colas, pilas...

- Permite de forma inmediata el abordar proyectos más ambiciosos sobre: traductores, intérpretes...

4.- BIBLIOGRAFIA

1. SIGCSE BULLETIN, Vol 8,nº1 y SIGCSE TOPICS Vol.2. "Papers of ACM SIG CSE-SIG CUE TECHNICAL Symposium on computer Science and Education". ACM 1975.
2. KNUTH D., "Top-Down Syntax Analysis" Acta Informática. Enero 1971.
3. HOPCROFT J.E., ULLMAN J. D., "Formal languaje and their relation to automata". Addison-Wesley 1969.
4. HEWLETT-PACKARD, "A pocket guide to HP computers: Specifications and Basic Operation manual".

SISTEMA DE CONTROL LOGICO PROGRAMABLE SOBRE MICROORDENADOR

por: J.L. Zuloaga
R. Santamaria
L. Emaldi
I. Ramos

Universidad de Bilbao, Centro de Cálculo
y Dpto. de Automática e Informática

El diseño de automatismos para el control de procesos y en especial su puesta a punto, representa uno de los campos en que la informática, y más particularmente las técnicas de microprocesadores, tiene una aplicación más evidente.
Un sistema de control lógico programable sería entonces una herramienta eficaz para reducir el tiempo (costo) de diseño y puesta a punto de un automatismo, así como de las distintas modificaciones que éste debe sufrir según el proceso a controlar.
En este papel se presenta un sistema que acepta como entrada la descripción de un circuito y produce como salida:
ó bien: código interno intérpre que servirá para una simulación y puesta a punto.
ó bien: código máquina para un µP de forma que éste sustituya en tiempo real al automatismo descrito.
Palabras clave: Automatismo, descripción, simulación, macrodefinición, macroexpansión, control, microprocesador.

I.- INTRODUCCION

La idea de sustituir la lógica cableada por la lógica programada para el control de un proceso no es nueva ni original. Sin embargo es difícil llegar a dotar al usuario (diseñador de automatismos) de una herramienta nueva que no le venga ancha.

Cuando se diseña y se construye un automatismo cableado (p.ej. de relés), el diseñador escribirá sobre un papel una descripción del circuito, planos, ... etc., de forma que el que deba construirlo tenga la información necesaria para ello.

En el caso de la lógica programada, el automatismo final será un programa máquina que explorará las entradas, las procesará y emitirá unas señales de salida.

Estas tres operaciones en un automatismo cableado se producen de forma con currente, mientras que en un sistema programado el procesador que debe efectuarlas es estrictamente secuencial.
De esta última consideración nace la siguiente pregunta:

¿El diseñador deberá decirle al sistema como tiene que efectuar estas operaciones o simplemente describir las componentes del automatismo? .

El sistema que aquí se presenta ha optado por la segunda solución pues parece la más natural: describiendo cómo es el automatismo, el sistema ya se encargará de generar el programa secuencial que trabaje como lo haría cableado.

Por lo tanto la forma de comunicación del usuario con el sistema es un lenguaje para describir automatismos.

Describir un automatismo consistirá en describir cada uno de sus componentes

REVISTA DE INFORMATICA Y AUTOMATICA

Zuloaga, J.L., Santamaria, R., Emaldi, R., Ramos, I. Sistema de Control lógico programable sobre microordenador. Revista de Informática y Automática. 1979.

dando criterios para su posterior enlace.

II.- EL LENGUAJE

El lenguaje que soporta la descripción del automatismo es un lenguaje de descripción por bloques.

Cada bloque vendrá descrito por:
- nombre de las señales de entrada.
- nombre del bloque (función que realiza.
- nombre de la señal (es) de salida.
- parámetros internos en caso de que los tenga (contadores, temporizadores ... etc.).

Con excepción del bloque "Registro de desplazamiento" todos los demás bloques tienen una sola salida.

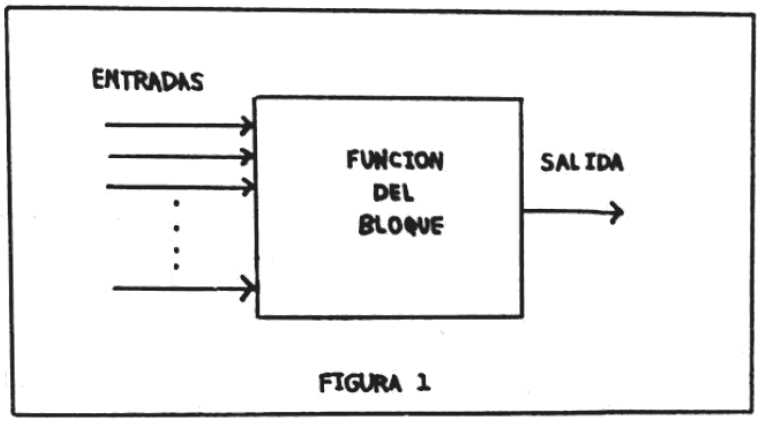


FIGURA 1

El lenguaje dispone de trece bloques básicos a partir de los cuales el usuario mediante macrodefiniciones podrá formar bloques más complejos.

La estructura general de descripción de un bloque es la siguiente:

nombre-de-la-señal-de-salida. función que realiza y parámetros internos . nombre entrada. ..... nombre-entrada. ;

Las señales se identifican mediante una letra seguida de un número comprendido entre Ø y 127. Quedando para el usuario de Ø a 99 y de 100 a 127 para generar variables internas el macroexpansor, así por ejemplo un bloque que realiza la operación AND con las señales A7,A11,E23 y F87 para dar la señal S 12 se escribirá:

S12. AND. A7,A11,E23,F87;

Un bloque que representa un temporizador que tiene dos entradas, reset R1, habilitación H1 y que debe dar como señal de salida S1 cuando haya contado 1328 pulsos de tiempo mientras estaba habilitado se describirá:

```
S1 . TIME-1328. R1,H1;$ H1
     ES HABILITACION

                              R1
     ES RESET $
```

lo que está encerrado entre los símbolos $, son comentarios que se pueden hacer para una mayor legibilidad de la descripción.

Otro tipo de sentencias son las que asocian las señales externas con identificadores para el sistema.

Por ejemplo si nosotros queremos que la señal A7 se lea del sensor 3, la señal A11 del 4, la señal E23 del 15, la señal F87 del 5, y que la señal S12 salga por el dispositivo 20 pondremos:

INPUT(A7,3) (A11,4) (E23,15) (F87,5);

OUTPUT (S12,20);

Como se ha dicho anteriormente lo que se hace es una descripción del automatismo, por lo tanto para el sistema el orden en que se describan los bloques es totalmente irrelevante.

El repertorio básico de funciones que están directamente implementadas en el sistema es:

INPUT Asignación de dispositivos de entrada a las señales.

OUTPUT Asignación de dispositivos de salida a las señales.

AND)
OR) bloques combinacionales
XOR)
NOT)

SHIFT Registro de desplazamiento, accesibles todos sus bits.

JKFF Flip - Flop J - K

DFF Flip - Flop D

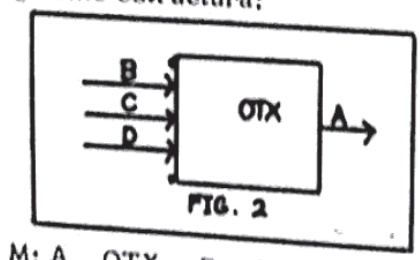
CONT Contador (tres entradas, pulsos)

TIME Temporizador (2 entradas: RESET Y HABILITACION)

PASS Relé de paso: produce un pulso de tamaño fijo, es como un monostable

WINK Relé de parpadeo mientras la entrada esté alta produce una onda cuadrada de anchura dada (multivibrador)

Además de estos bloques básicos se pueden definir bloques más complejos mediante definiciones que tienen la siguiente estructura:

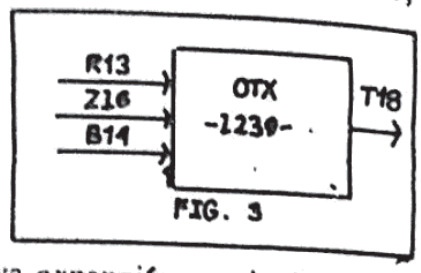


M: A . OTX . B , C , D;
A . AND . @12, @6, B;
@12 . TIME . B, B ;
@6. XOR . C, D ;
ENDM ;

Las señales que empiezan por @ son internas al cuerpo de las macros y se cambiarán por etiquetas A100 ...A127, B100 .....Z 127.

Una referencia a la anterior macro sería:

T18 . OTx - 1239 . R13, Z16, B14;



Cuya expansión quedaría:

T18 . AND . A100 , A101 , B14;

A100 - TIME - 1239 - R13, R13;
A101 - XOR . Z16 , B14;

No se ha hecho más que sustituir cada señal por su correspondiente en la macrollamada

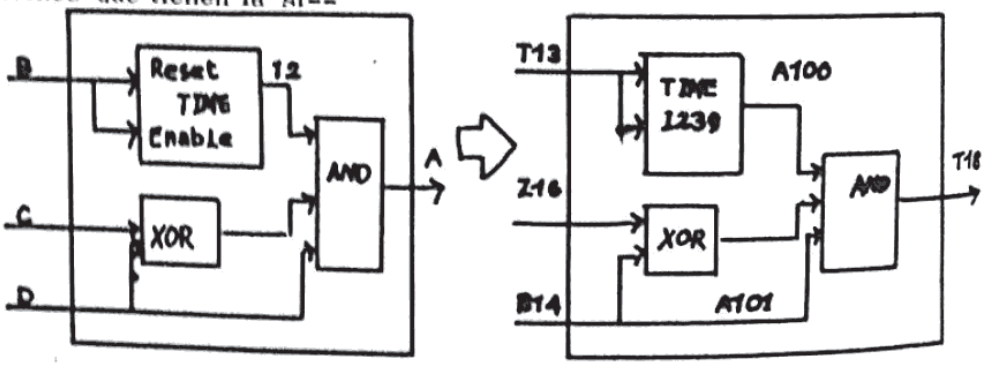


FIGURA 4

A, B, C y D, son los parámetros formales de la macro, @12, y @6 sus variables internas ficticias.

Las reglas para definir un macrobloque son las siguientes:

1 - debe empezar por M: definición; cuerpo; ENDM;

2 - El nombre de un macrobloque estará formado por un conjunto de hasta 3 letras que no coincidan con las tres primeras de ninguno de los bloques básicos.

3 - Los parámetros formales constan de una sola letra.

4 - Un macrobloque puede tener hasta 25 entradas y una sola salida.

5 - Las variables internas ficticias empezarán por @ y deberán estar definidas dentro de la macro, es decir, deberán ser señales de salida de algún bloque interno del macrobloque.

6 - Dentro de la definición de una macro no se puede definir ni usar otra.

7 - Son bloques prohibidos en una macro: INPUT, OUTPUT, y SHIFT.

El uso de esta facilidad de las macrodefiniciones puede servir para un diseño modular en los automatismos.

A continuación se presenta formalmente el lenguaje mediante una gramática en notación BNF:

⟨PROGRAMA⟩ ::= ⟨SENTENCIAS⟩ FIN; | ⟨MACROS Y SENTENCIAS⟩ FIN;

⟨MACROS Y SENTENCIAS⟩ ::= ⟨MACROS Y SENTENCIAS⟩⟨MACRO⟩ ENDM; | ⟨MACROS Y SENTENCIAS⟩ | ⟨SENTENCIAS⟩

⟨SENTENCIAS⟩::=⟨SENTENCIA ⟩; ⟨SENTENCIAS⟩ | ε

⟨MACRO⟩ ::= M ⟨DEFINICION⟩ ⟨CUERPO⟩

⟨DEFINICION⟩ ::= ⟨LETRA⟩ . ⟨D.RESTO⟩

⟨D. RESTO⟩::=⟨NOMBRE DE MACROS⟩ . ⟨PARAMETROS⟩

⟨PARAMETROS⟩::=⟨LETRA⟩ ⟨P.RESTO⟩

⟨P. RESTO⟩ ::= , ⟨LETRA⟩ | ; ⟨CUERPO⟩ ::= ⟨SENTENCIA DE MACRO⟩ ⟨CUERPO⟩ | ε

⟨SENTENCIA DE MACRO⟩ ::= ⟨IDENTIFICADOR DE MACROS⟩ ⟨S.M. RESTO⟩

⟨S.M. RESTO⟩ ::= , ⟨NOMBRE DE BLOQUE DE MACRO⟩ . ⟨ENTRADAS DE MACRO⟩

⟨NOMBRE DE BLOQUE DE MACRO⟩ ::= OR | AND | XOR | NOT | JKFF DFF | CONT | TIME | PASS | WINK

⟨ENTRADAS DE MACRO⟩ ::= ⟨IDENTIFICADOR DE MACROS⟩⟨E.M. RESTO⟩

⟨E.M. RESTO⟩ ::= , ⟨IDENTIFICADOR DE MACRO⟩ ;

⟨IDENTIFICADOR DE MACRO⟩ ::= ⟨LETRA⟩ | @⟨NUMERO 2⟩

⟨SENTENCIA⟩ ::=⟨IDENTIFICADOR⟩ ⟨S.RESTO⟩ | ⟨MACRO LLAMADA⟩ | INPUT ⟨PUERTA⟩ | OUTPUT ⟨PUERTA⟩ | ⟨COMENTARIO⟩

⟨COMENTARIO⟩ ::= $ ⟨CADENA DE CARACTERES⟩ $

⟨CADENA DE CARACTERES⟩ ::= ⟨CARACTER⟩⟨CADENA DE CARACTERES⟩ | ε

⟨CARACTER⟩ ::=⟨LETRA⟩ | ⟨NUMERO⟩ | . | , | ; | = | @ | ' | % | = | + | * | ? | !

⟨PUERTA⟩ ::= (⟨IDENTIFICADOR⟩ , ⟨DIGITO 2⟩) ⟨PV. RESTO⟩

⟨PV.RESTO⟩ ::= (⟨IDENTIFICADOR⟩ , ⟨DIGITO 2⟩) ⟨PV.RESTO⟩ | ε

⟨S. RESTO⟩ ::= , ⟨NOMBRE DE BLOQUE⟩ . ⟨ENTRADAS⟩

⟨ENTRADA⟩ ::= ⟨IDENTIFICADOR⟩ ⟨E.RESTO⟩

⟨E.RESTO⟩ ::= , ⟨ENTRADAS⟩ | ;

⟨NOMBRE DEL BLOQUE⟩ ::= OR | AND | XOR | NOT | SHIFT- ⟨LETRA⟩ (⟨DIGITO 2⟩) | JKFF | DFF | CONT- ⟨NUMERO 5⟩ | TIME-⟨NUMERO 5⟩ | PASS- ⟨NUMERO 5⟩ | WINK- ⟨NUMERO 5⟩

⟨MACRO LLAMADA⟩ ::=⟨IDENTIFICADOR⟩⟨M.LL. RESTO⟩

⟨M.LL. RESTO⟩ ::= . ⟨NOMBRE DE MACRO LLAMADA⟩ . ⟨ENTRADAS⟩

⟨NOMBRE DE MACROLLADADAS⟩ ::= ⟨NOMBRE DE MACRO⟩⟨CUENTAS⟩

⟨CUENTAS⟩ ::= = ⟨NUMERO 5⟩ ⟨CUENTAS⟩ | ε

⟨NOMBRE DE MACRO⟩ ::=⟨LETRA⟩ ⟨N.M. RESTO⟩

⟨N.M. RESTO⟩ ::=⟨LETRA⟩⟨N.M.R. RESTO⟩ | ε

⟨N.M.R. RESTO⟩ ::=⟨LETRA⟩ | ε

⟨IDENTIFICADOR⟩ ::= ⟨LETRA⟩ | ⟨IDENTIFICADOR⟩ ⟨NUMERO 2⟩

⟨DIGITO 2⟩ ::= ⟨NUMERO⟩ ⟨NUMERO 1⟩

⟨NUMERO 5⟩ ::= 2 ⟨NUMERO 4⟩ | 1 ⟨NUMERO 4⟩ | Ø ⟨NUMERO 4⟩

⟨NUMERO 4⟩ ::= ⟨NUMERO⟩ ⟨NUMERO 3⟩ | £

⟨NUMERO 3⟩ ::= ⟨NUMERO⟩ ⟨NUMERO 2⟩

⟨NUMERO 2⟩ ::= ⟨NUMERO⟩ ⟨NUMERO 1⟩

⟨NUMERO 1⟩ ::= ⟨NUMERO⟩

⟨LETRA⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

⟨NUMERO⟩ ::= Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .

III.- DESCRIPCION DEL FUNCIONAMIENTO DEL SISTEMA.

El sistema físico está compuesto de dos partes:

1 - Unidad de desarrollo y generación de programas: Aquí es donde se traducen, se prueban y se generan los programas máquina de las descripciones.

2 - Unidad de control: Es en este donde se ejecutan los programas equivalentes de los automatismos y sus posibilidades son solo de: cargar programas, ejecutar programas, y cambiar parámetros de contadores, temporizadores ....etc.

Veamos mediante un organigrama la forma de operar el sistema de desarrollo.

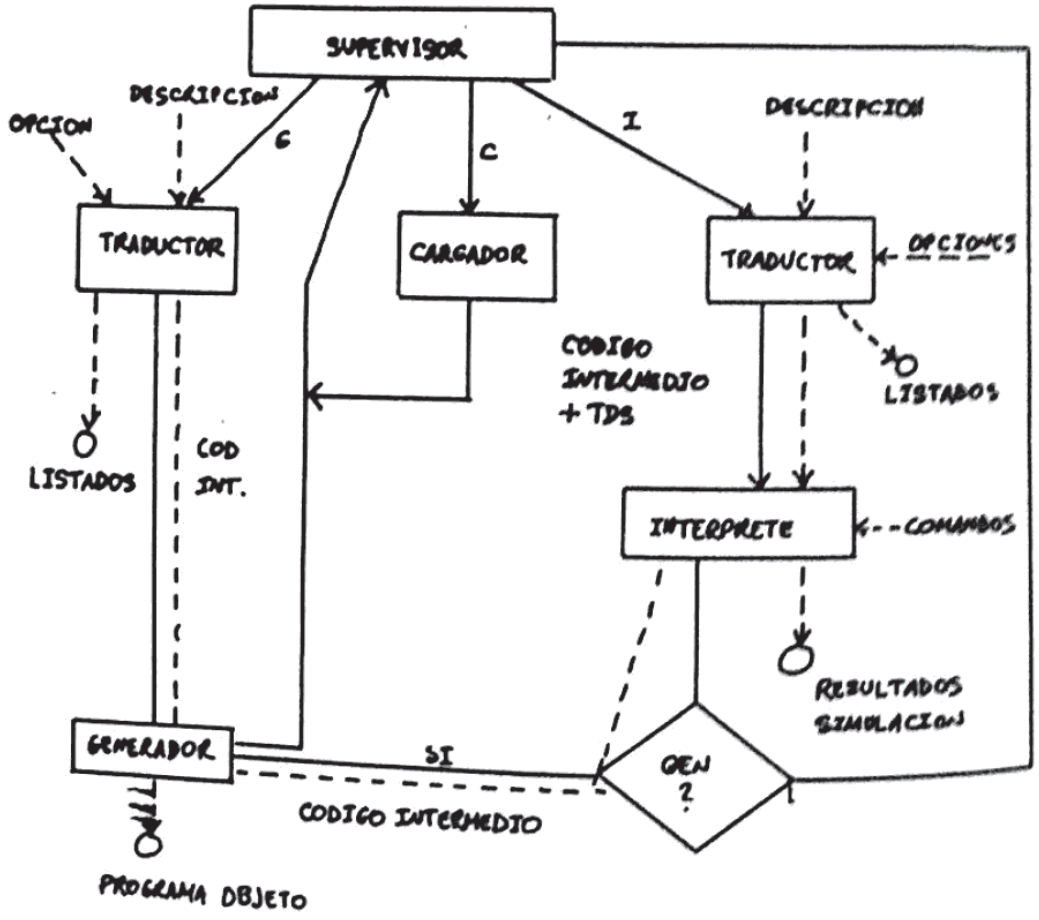


FIGURA - 5

En la figura vemos que el control del sistema la realiza un monitor que analiza los comandos del operador y realiza la operación pedida.

Estas operaciones pueden ser:

C: cargar y ejecutar un programa máquina.

I: traducir y simular (Interpretar) una descripción de automatismo.

Para la interpretación se dispone de los siguientes comandos:

INICIALIZAR NS = (Ø (, ..., (1 (

NS (Ø ((1 (. (NS = Nombre de señal)

DISPLAYAR NS, NS, ..., NS.

REPETIR.

FIN.

El usuario entonces podrá hacer un testeo todo lo completo que quiera del circuito descrito.

Al final del testeo si se desea se puede generar código máquina de ese automatismo.

G : Traducir y generar código máquina de una descripción de automatismo.

Veamos a continuación en que consiste cada una de las operaciones reseñadas anteriormente.

III. 1. TRADUCCION

Es la operación que a partir de la descripción de un automatismo general un código interno que servirá o bien para hacer una simulación y testeo del mismo o bien para generar código máquina para el micro-procesador.

Esta operación se realiza en tres pasos:

1 - El primer paso consiste en expandir las macrollamadas a partir de las definiciones que se hayan dado de macrobloques.

Deja la descripción reducida a bloques elementales dando un listado de los errores que se hayan producido.

2 - El segundo paso efectúa un análisis del texto de la descripción dando el listado de los errores sintácticos que se produzcan y produciendo como resultado una estructura de información a base de listas ligadas que será la entrada al siguiente módulo.

3 - Este módulo efectúa a partir de los bloques en código interno los enlaces que entre ellos formando el grafo del circuito y dando la secuencia correcta de ejecución de los bloques. Detectará incoherencias tales como: señales que son salidas de bloques pero que no se utilizan como entradas a otros o como salida del automatismo, señales que se definen como entradas a algún bloque pero que no han sido previamente definidas como entradas del automatismo o salidas de algún otro bloque.

Además se puede pedir un listado del código intermedio producido, un listado de la tabla de símbolos, un listado de las direcciones relativas de las cuentas de cada temporizador y contador de cara a poder modificarlas en tiempo de ejecución. Tras este paso ya queda toda la información de como opera el circuito lista para, o bien generar código, o bien efectuar una simulación y testeo.

III. 2. - SIMULACION Y TESTEO.

Antes de poner en funcionamiento un automatismo programado, el usuario necesitará probar su correcto funcionamiento. Para ello preparará una batería de test que consistirá en configuraciones de las señales de entrada (o de algunas de ellas) y ver que señales se obtienen tras un ciclo de ejecución.

Los comandos que admite el intérprete son los siguientes: (lo que está en minúsculas lo escribe el sistema):

Comienzo
Inicializar:
Displayar:
Repetir
Fin

Veamos con un ejemplo como funciona el simulador:

El circuito es:

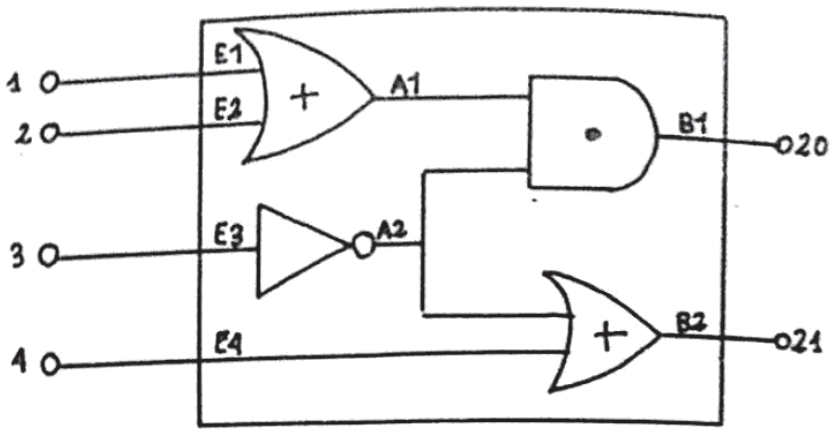


FIGURA - 6

```
$ ESTA SERIA SU DESCRIPCION $
 B1 . AND . A1 , A2 ; A1 . OR . E1 ,
EZ ;
 C1 . OR . A2 ,
          E4 ;
 A2 . NOT . E3 ;
 INPUT (E1 , 1) (E2 , 2)
       (E3 , 3)  (E4 , 4) ;

 OUTPUT (B1 , 20) (C1 , 21) ;

 FIN
```

El traductor dejaría a partir de esta descripción el código interno dispuesto para utilizar por el intérprete.

Ahora entra en juego el intérprete, una posible ejecución quedaría:

```
- INTERPRETE-
COMIENZO
 INICIALIZAR:E1=1, A2=1, E4=1 .
DISPLAYAR: A1, B1, C1.
 REPETIR
```

Se ejecuta un ciclo del circuito dando como resultado el valor de las señales que se ha pedido displayar.

```
**LA SALIDA A1 NO ESTA DEFI-
  NIDA**

** LA SALIDA B1 NO ESTA DEFI-
  NIDA* *

   C1 = 1
```

y se vuelve a admitir comandos de inicialización o de display.

```
INICIALIZAR  E1=1, EZ=Ø, E3=Ø ,
             E4=1.

DI SPLAYAR  A1, A2, B1, C1.

FIN

A1 = 1
A2 = 1
B1 = 1
C1 = 1
```

y aquí preguntará si se desea generar código máquina. Puesto que si hubieramos deseado efectuar más test debíamos haber puesto el comando R.

III.3.- GENERACION DE CODIGO

Cuando habíamos puesto el comando G del monitor, o después de poner FIN en el intérprete respondemos afirmativamente a la pregunta de si deseamos generar código, entra en funcionamiento el generador de código.

Este programa pregunta la dirección a partir de la cual el programa debe ser generado y a partir del código intermedio genera un programa objeto (para Intel-808Ø) que sustituirá en tiempo real al automatismo descrito.

Además genera la rutina de interrupción del operador para cambiar las cuentas de los temporizadores y contadores mientras el automatismo está ya funcionando.

La cantidad de código generado dependerá del tamaño y complejidad del automatismo descrito, por ejemplo se genera una cantidad de código considerablemente mayor para un shift-register que para una puerta "and".

Se lleva a cabo también una optimización del código generado, de una forma sencilla pero que ahorrará del orden de un 10% de instrucciones máquina.

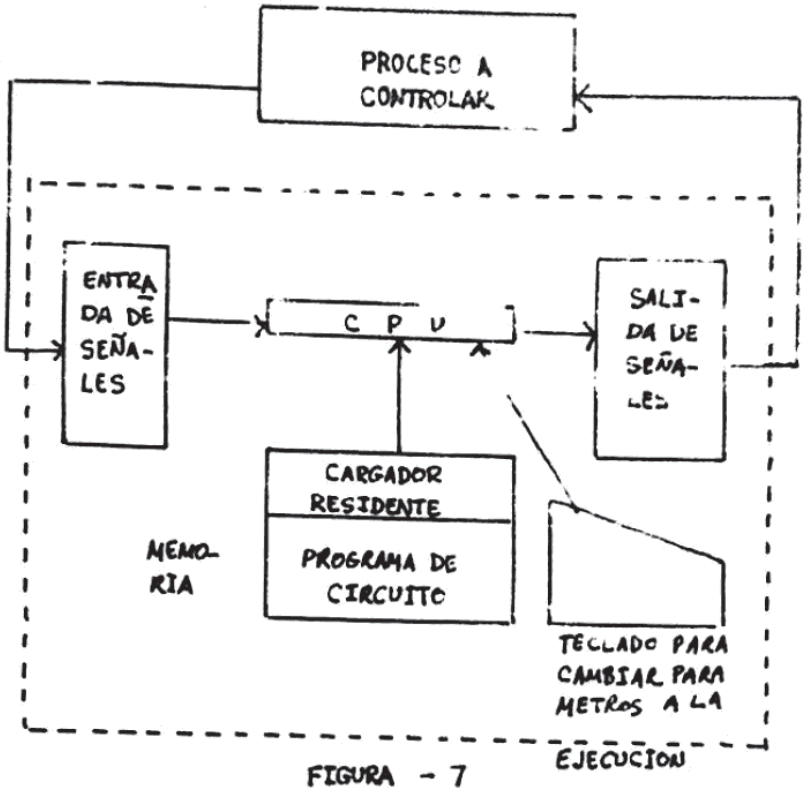


FIGURA - 7

BIBLIOGRAFIA

1) BRINCH HANSEN, "Operating Systems Principles". Prentice - Hall Englewood Cliffs, N.J., 1973.

2) CROCUS, "Systèmes d'explotation des ordinateur". Dunod - 1975.

3) FOSTER, J.M. "Automatic Sytactic Analysis". Macdonald/Elsevier Computer Monographs. 1971.

4) ICOM, inc. "F DOS - II / MOD 80 OPERATOR'S MANUAL". oct. 1975.

5) INTEL CORP. "PL/M Programming manual" - 1975

6) INTEL CORP. "8080" Assembly language, programming manual" 1974.

7) INTEL CORP. "Intellec 8/MOD 80, Microcomputer Systems operator's manual" - 1974.

8) KESSELS J.L.W.,"Parallel programa ming conceps in a difinitional language" - A.C.M. Sigplan - oct. 1976.

9) PAIR, C., "Compilation". Ecole d'été d'informatique. Neuchatel, sept. 1972.

10) WHEAVER, A.CH. , "VIPTRAN 2 Au improved programming language and Its compiler for process control equations". I.E.E. Comp. Soc. 1975.

11) WIRTH, N. , "Systematic programming, and introduction". Prentice Hall - 1973.

12) WIRTH, N., "Toward a discipline of real-time programming". Com. A.C.M. - Aug.1977 - Vol. 20 nº 88, pp. 577-583.

MIKBUS: UN BUS PARA PLACAS DE MICRO-COMPUTADORES COOPERANTES

Jose Ramon Garitagoitia *
Carmelo Martinez **
Isidro Ramos ***

* Dpto. Automatica e Informatica
Facultad de Ciencias - Universidad del Pais Vasco
Lejona (Vizcaya)

** MIKROA S.A.L.
c/ Juan Bautista Zabala 12,6
Algorta (Vizcaya)

*** Centro de Informatica - Universidad de Valencia
c/ Doctor Moliner s/n.
Burjasot (Valencia)

Garitagoitia, J.R., Martinez,C., Ramos, I. MIKBUS: un bus para placas de micro-computadores cooperantes. Actas del II Simposio Español de Informática Repartida. 1982.

- INTRODUCCION

En este articulo se describe el hardware y el software de un sistema de conexion de placas de microcomputadores, no necesariamente homogeneos y sin prioridades.

El sistema junto con el monitor construido, permite escribir las tareas independientemente de su posterior distribucion en las estaciones de la red.

La asignacion de tareas a los nodos se puede realizar en funcion de las necesidades de calculo, velocidad de respuesta, capacidad de memoria, etc.

El esquema de conexion esta pensado para aplicaciones en tiempo real, fundamentalmente de control industrial, donde el recurso procesador puede ser critico.

Bajo esta filosofia, se ha realizado un sistema de control numerico para maquina herramienta con tres microcomputadores, que se encuentra en fase de comercializacion.

- DEFINICION GENERAL

El sistema de comunicacion esta basado en un unico bus del que cuelgan las distintas unidades de proceso (Fig. 1).

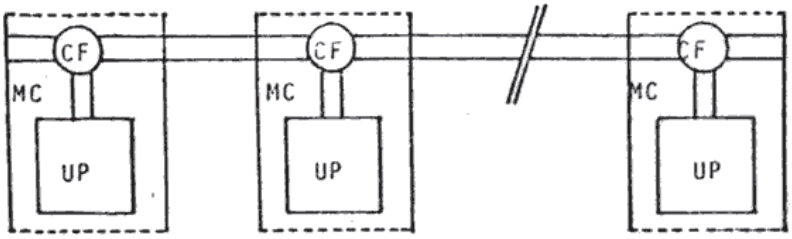


fig.1 MC Sistema microcomputador
UP Unidad de proceso
CF Controlador de flujo

El multiplexado y ocupacion del bus se realiza por los componentes del sistema mediante el correspondiente controlador de flujo (*Switch*).

El esquema presentado se puede clasificar como bus compartido multiplexado, de control distribuido y de transferencia indirecta (*Bus Window*) (1,2).

- CARACTERISTICAS DEL SISTEMA

El bus

El bus es mixto, pudiendose determinar dos zonas que se suceden alternativamente:

- *Bus real:* Constituido por las lineas hardware que unen un sistema micro-computador (MC) con el siguiente o el anterior.

- *Bus simulado:* El paso de datos a traves de un nodo, gobernado por software.

La union (interface) de las dos zonas definidas se realiza mediante puertas, que pueden ser programadas alternativamente como de entrada o salida (Fig. 2).

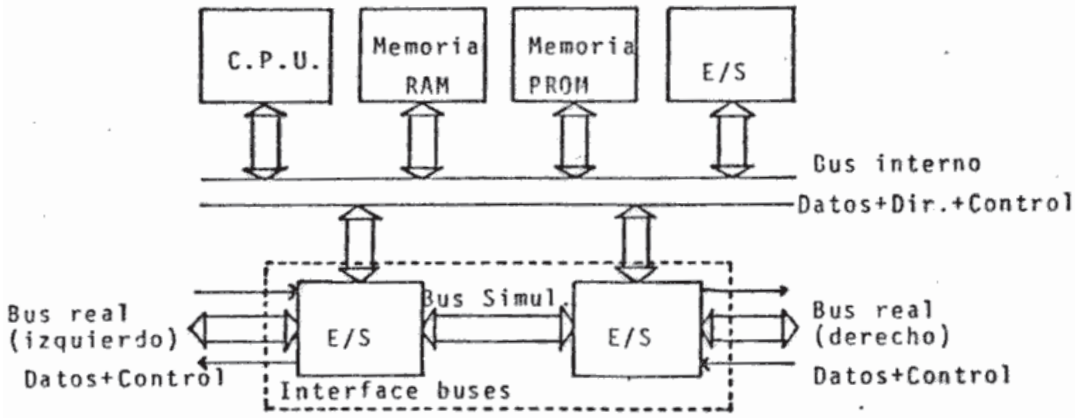


fig.2

El bus consta de ocho lineas de datos y dos lineas de control.

Ocupacion y multiplexado del bus

Cada uno de los tramos del bus real, es ocupado alternativamente por los dos MC de sus extremos durante la transmision de un mensaje. Existe un mensaje nulo con el fin de no bloquear la comunicacion en sentido contrario.

El multiplexado del bus se realiza independientemente para cada una de las zonas por las distintas unidades acopladas al bus.

No es necesario que el bus uniendo el origen y destino de un mensaje, esté por completo asignado para comenzar su envio. De esta forma, una zona puede contener datos de un mensaje diferente al contenido en otra parte del bus e incluso puede transmitirse en sentidos diferentes.

Tipo de comunicacion y mensajes.

La comunicacion se realiza virtualmente entre las tareas de los distintos MC por medio del controlador de flujo que permite que la localizacion en la red de la tarea destino sea transparente para la origen.

En el nivel virtual los mensajes no cumplen regla alguna, solo se debe especificar la tarea destino.

El mensaje real esta dividido en dos partes:

- La *cabecera* consta de tres caracteres (Bytes): Tarea destino, longitud del texto tarea origen.

- El *texto* coincide con el mensaje virtual.

No se ha incluido una *cola* con un codigo de deteccion de errores pues se supone que los MC estan fisicamente proximos y unidos por una placa madre.

La comunicacion es paralelo y asincrona con velocidad no homogenea a lo largo del bus: La velocidad de transmision es igual y muy elevada (Del orden de MHz) en los tramos de bus real y menor a traves de cada MC, dependiendo incluso de su carga.

Estructura logica

Los nodos del sistema conectados al bus estan numerados de izquierda a derecha en orden creciente de tal forma que dado un MC, las unidades con numeros menores se encuentran a su izquierda y al reves.

Una tarea esta identificada por un caracter (Byte) compuesto por el numero del nodo en que corre y un numero que la distingue del resto de tareas del mismo procesador.

La aplicacion limita el numero de nodos y el de tareas por estacion a 16.

- FUNCIONAMIENTO

Los bytes de que se compone el mensaje son transmitidos uno a uno a traves de las zonas del bus.

Un caracter del mensaje no es transmitido a una zona determinada del bus hasta que el anterior ha pasado a la zona siguiente.

Un mensaje puede estar en uno, dos o tres de los estados siguientes:

- Saliendo del nodo origen.

- En una zona de bus comprendida entre el origen y destino. Los datos del mensaje se encuentran en sucesivos tramos del bus.

- Llegando al nodo destino.

La transmision esta basada en el controlador de flujo y sus funciones:

- Direccionamiento de mensajes (*Routing*).

- Cruce de mensajes.

Tres son las causas que necesitan un direccionamiento de mensajes:

- Envio de un mensaje por una tarea del sistema MC al que pertenece el CF: Segun sea la tarea destino, se ocupa el bus real izquierdo o derecho correspondiente.

- Comienzo de recepcion de un mensaje para una tarea de otro nodo: Se ocupa el bus real contrario al que se ha hecho la recepcion.

- Recepcion de un mensaje para una tarea del nodo: Recoge el mensaje para la tarea correspondiente.

Puede darse que un CF reciba dos mensajes que circulan en sentidos diferentes a un mismo tiempo, produciendose una colision. El problema se ha resuelto dando prioridad a la transmision en un sentido y dotando al CF de un buffer auxiliar (Via muerta) donde descarga el mensaje que circula en sentido de baja prioridad (Fig. 3).

El multiplexado de una zona del bus real se consigue mediante el envio de un mensaje nulo o de conmutacion entre los dos MC de sus extremos.

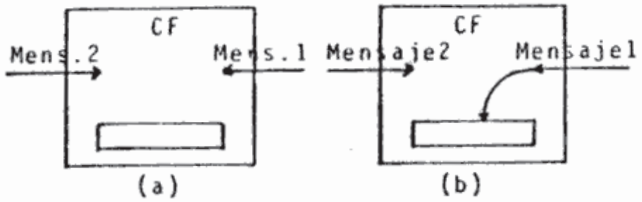
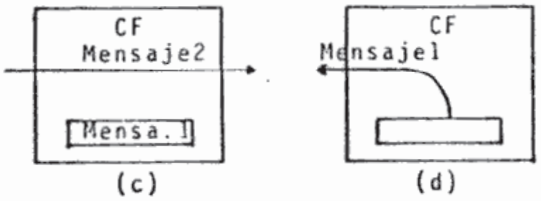


fig.3



El nodo que recibe el mensaje de conmutacion ocupa el bus para transmitir un mensaje pendiente si lo hay o devolver el mensaje nulo.

- ESQUEMA HARDWARE

La interface con el bus (Fig. 2) se ha realizado con un 8155 de Intel (5).

La puerta A comunica al bus real derecho y la B al izquierdo. Ambas puertas trabajan en el modo *'strobed'* y pueden ser programadas sucesivamente como de entrada o salida.

El diseño sigue las lineas de la fig. 4.

En la fig. 5 se muestra el diagrama de tiempos para la transmision de un caracter de la puerta A a la B de otro MC.

La transmision de un dato (y su recepcion) queda asegurada por el juego de señales.

El paso de un caracter de una puerta a otra se realiza de forma independiente del programa en un tiempo que oscila entre 50 ns. y 800 ns. maximo.

- ESQUEMA SOFTWARE

El controlador de flujo se considera dividido en dos: el que atiende al bus derecho y el que lo hace al izquierdo. Cada una de estas partes sigue el diagrama

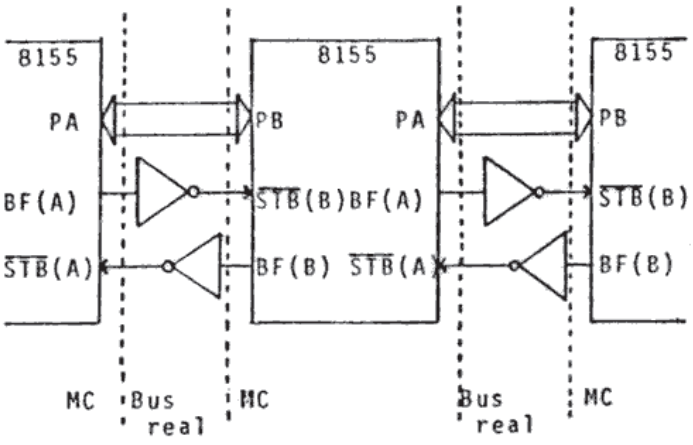

8155
8155
8155
PA
PB
PA
PB
BF(A)
STB(B) BF(A)
STB(B)
STB(A)
BF(B) STB(A)
BF(B)
MC
Bus real
MC
Bus real
MC

fig.4

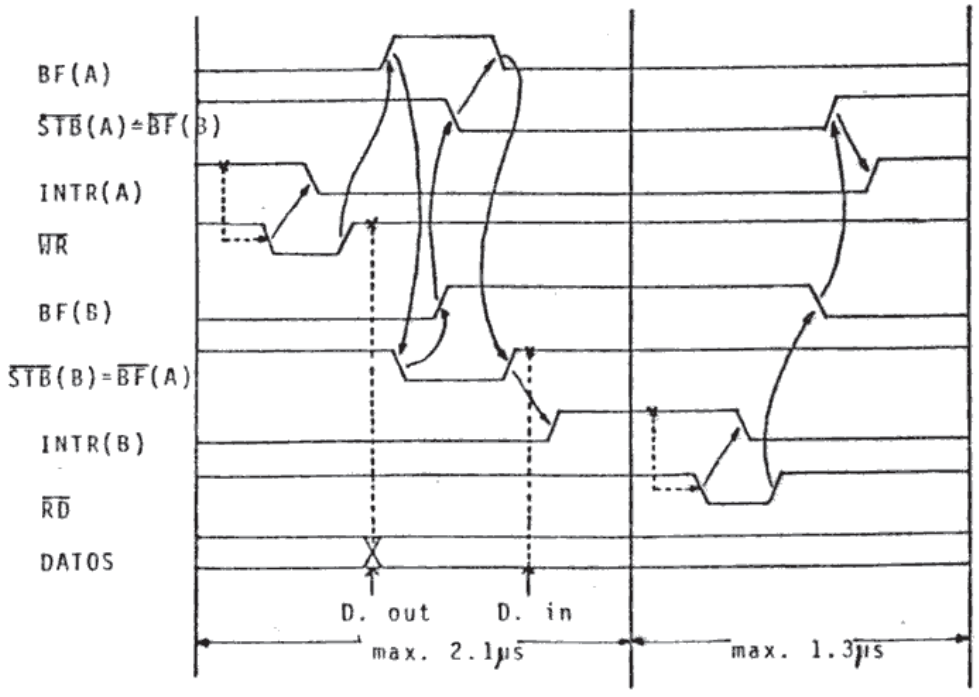

BF(A)
STB(A)=BF(B)
INTR(A)
WR
BF(B)
STB(B)=BF(A)
INTR(B)
RD
DATOS
D. out
D. in
max. 2.1μs
max. 1.3μs

fig.5

de estados de la fig. 6.

La transicion (b) solo existe en la interface izquierda y la (e) asi como el estado 9, en la parte derecha de acuerdo con la prioridad establecida.

- EVALUACION DEL SISTEMA

El esquema presentado se ha clasificado dentro del grupo de bus compartido, multiplexado, de control distribuido y de transferencia indirecta (*Bus Window*), por lo tanto, tendrá las caracteristicas del grupo.(1,2)

- Costo modular (*Cost modularity*): Puede ser bueno si en la placa madre se ha previsto una futura expansion.

- Tolerancia a fallos (*Fault tolerance*): Esta caracteristica varia con respecto al grupo. Un mal funcionamiento en una placa microcomputador divide al bus en dos, siendo posible la comunicacion en cada una de las partes, pero no entre ellas. Es facil la deteccion de fallos por cualquiera de los nodos en funcionamiento (Timeout en la conmutacion del bus).

- Complejidad local (*Local complexity*): Es muy buena. El numero de test para encaminar la informacion no aumenta al añadir nuevos nodos.

- Embotellamiento (*Bottleneck*): Los embotellamientos son posibles, pero la posibilidad es menor que la del grupo al permitir circular mas de un mensaje incluso en sentidos diferentes.


1
LISTO PARA TRANSMITIR
Pendiente: 'SEND' desde la tarea
2
ENVIO DEL MENSAJE
Envio de tarea destino
3
ENVIO DEL MENSAJE
Envio longitud del mensaje
4
ENVIO DEL MENSAJE
Envio de tarea origen
5
ENVIO DEL MENSAJE
Envio de dato del texto del mensaje
Fin envio del mensaje
6
FIN DE TRANSMISION
Reset flags
9
CONTROL DE ESTADO
(a)
1
LISTO PARA TRANSMITIR
Pendiente: envio del mensaje en el buffer auxiliar
7
PREPARACION DEL ENVIO
Actualizacion flags y variables
2
ENVIO DEL MENSAJE
(b)
1
LISTO PARA TRANSMITIR
Pendiente: mensaje en recepcion por el otro bus
8
RETRANSMISION DEL MENSAJE (REALIZACION)
Fin recepcion y envio del mensaje
6
FIN DE TRANSMISION
(c)
1
LISTO PARA TRANSMITIR
Pendiente: nada
9
CONTROL DE ESTADO
Buffer auxiliar ocupado
Buffer auxiliar libre
10
CAMBIO DE MODO: RECEPCION
Envio mensaje de conmutacion
11
RECEPCION
Recepcion mensaje de conmutacion
12
CAMBIO DE MODO: TRANSMISION
1
LISTO PARA TRANSMITIR
Recibe tarea destino
13
RECEPCION MENSAJE
Destino: procesador propio
14
RECEPCION MENSAJE
Reserva espacio memoria para el mensaje
15
RECEPCION MENSAJE
Recibe longitud mensaje
16
RECEPCION MENSAJE
Recibe dato del mensaje
Fin recepcion del mensaje y envio a tarea destino
11
RECEPCION
Destino: otro procesador
17
RETRANSMISION MENSAJE (PREPARACION)
Bus libre (poner retransmision pendiente)
18
RETRANSMISION MENSAJE (PREPARACION)
Fin recepcion y envio del mensaje
11
RECEPCION
(d)

fig. 6

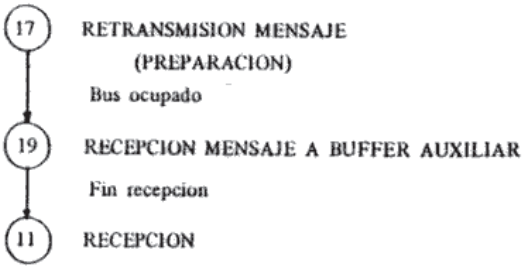


fig. 6 (e)

- BIBLIOGRAFIA

1.- G.A. Anderson, E.D. Jensen

'Computer Interconnection: Taxonomy, Characteristic, and Examples'

ACM Computer Surveys (1975)

2.- K.J. Thurber, G.M. Masson

'Distributed- Processor Communication Architecture'

Lexington Books (1979)

3.- P.B. Hansen

'Operating Systems Principles'

Prentice- Hall (1973)

4.- Madnick and Donovan

'Operating Systems'

Mc. Graw- Hill (1974)

5.- Intel

'MCS-80/85 Family User's Manual'

ALGORITMO DE PREVENCION PARA UN SISTEMA DE GESTION DE FICHEROS

Jose Ramon GARITAGOITIA*, Isidro RAMOS**

*) Fac. Ciencias - Univ. del Pais Vasco
**) Fac. Ciencias - Univ. de Valencia

RESUMEN

Los sistemas informáticos actuales, evolucionan de forma que el grado de compartición simultánea de recursos por distintos procesos, es cada vez mayor. En estas circustancias el problema del interbloqueo va adquiriendo importancia.

El algoritmo presentado, se ha realizado para el caso en que los recursos sean un conjunto o sistema de ficheros, ya que es en este punto, después del gran desarrollo de las bases de datos, donde el problema es mas crítico.

El artículo presenta las distintas etapas seguidas en nuestro trabajo: Definición de un modelo del sistema de ficheros, caracterización del interbloqueo sobre el modelo, estudio de las condiciones en las que aparece el problema y concepción del algoritmo de prevención.

Para finalizar se hace una evaluación del algoritmo y se presenta un ejemplo de su comportamiento.

El presente trabajo es continuación de otros anteriores de los autores, las soluciones que se intuian quedan ahora formalizadas.

Garitagoitia, J.R., Ramos, I. Algoritmo de prevención para un sistema de gestión de ficheros. Actas de la Convención Informática Latina. 1983.

ALGORITMO DE PREVENCION PARA UN SISTEMA DE GESTION DE FICHEROS

Jose Ramon GARITAGOITIA*, Isidro RAMOS**

*) Fac. Ciencias - Univ. del Pais Vasco
**) Fac. Ciencias - Univ. de Valencia

1. INTRODUCCION

El problema del interbloqueo de procesos concurrentes (Deadlock) se da en todo sistema en el que se comparten recursos no compatibles simultaneamente.

La compartición simultánea de un sistema de ficheros puede dar lugar a estados incoherentes desde el punto de vista sintáctico y semántico (15), lo que obliga a definir nuevas operaciones (abrir,cerrar,...) sobre el sistema, en orden a eliminar ciertas ejecuciones simultaneas.

De esta forma, se puede ver el sistema de ficheros como un conjunto de recursos no compatibles simultaneamente, al menos para un conjunto de operaciones, lo que permite que aparezca la situación de interbloqueo.

Hay tres formas de abordar el problema:

- Establecer una serie de reglas para que no se den las condiciones (7,8) en las que aparece el interbloqueo (Exclusión mutua, Scheduling no preventivo ocupación parcial, espera circular)."Prevention". (3,5,6,11,14,24,37,40).

- Esperar a que se dé el problema, detectarlo y establecer una salida de la situación. "Detection and Recovery".(1,2,4,28,30,33,34,37).

- Controlar el sistema para no llegar a la situación. Basado en el conocimiento a priori de las necesidades globales de recursos (Declarar). "Avoidance". (12,19,21,22,23,26).

El algoritmo presentado sigue la tercera linea. Al igual que otros autores (25,27,41), se busca un modelo del sistema sobre el que se caracterizan los estados sin peligro de interbloqueo ("safe situation") y aquellos en que puede darse ("unsafe situation"). Finalmente, se construye un algoritmo que hace evolucionar al sistema, siempre entre estados permitidos.

2. DEFINICION DEL MODELO

Para la construcción de un algoritmo de prevención, es necesario saber de antemano las necesidades de recursos en el sistema. Se va a obligar a cada proceso a "declarar"

todos los ficheros que va a utilizar, antes de empezar su ocupación. La operación declarar, es aceptada siempre por el sistema y define el triplete: proceso, fichero, operación.

El modelo del sistema es el "GRAFO DE FICHEROS DECLARADOS", que se va a ir construyendo con las operaciones "declarar fichero".

Sea $P=\{p_1,\ldots,p_n\}$ conjunto de procesos en ejecución, $F=\{f_1,\ldots,f_m\}$ conjunto de ficheros declarados y $O=\{o_1,\ldots,o_r\}$ conjunto de operaciones posibles sobre los elementos de F.

Se define el grafo $G=\{D,\Gamma\}$ llamado "Grafo de ficheros declarados" en el conjunto $\Phi = P*F*O$ de la siguiente forma:

- D conjunto de nodos: $D=D_1 \cup D_2 \cup \ldots \cup D_r$ y $D_i \in \Phi$.

  $D_k=\{(p_i,f_j,o_k) \mid p_i$ ha declarado f_j con una operación $o_k\}$

  $(p_i,f_j,o_w)\in D_w$, $(p_k,f_r,o_v)\in D_v$ son nodos opuestos si $f_j=f_r$ y o_w,o_v son operaciones incompatibles (no compatibles simultaneamente).

- Γ función de sucesión: $\Gamma=(\Gamma_h,\Gamma_v)$

  Sean $(p_i,f_j,o_w),(p_k,f_r,o_v)\in D$

  $\Gamma_h(p_i,f_j,o_w)=(p_k,f_r,o_v) \Leftrightarrow (p_i,f_j,o_w),(p_k,f_r,o_v)$ son nodos opuestos.

  $\Gamma_v(p_i,f_j,o_w)=(p_k,f_r,o_v) \Leftrightarrow p_i=p_k$

El grafo es no orientado.

El nodo (p_k,f_r,o_v) es siguiente del (p_i,f_j,o_w) si existe un camino Γ^* tal que $\Gamma^*(p_i,f_j,o_w)=(p_k,f_r,o_v)$. Siendo

$$\Gamma^* = \underbrace{\Gamma_x \cdot \Gamma_{x-1} \cdot \ldots \cdot \Gamma_v \cdot \Gamma_h}_{\longleftarrow k\text{-veces} \longrightarrow} \quad \left| \begin{array}{l} \Gamma_x=\Gamma_h \text{ k=n. impar} \\ \Gamma_x=\Gamma_v \text{ k=n. par} \end{array} \right.$$

El nodo $(p_i,f_j,o_v)\in D$ es un nodo activo si p_i ocupa f_j

El nodo $(p_i,f_j,o_v)\in D$ es un nodo pasivo si p_i no ha intentado la ocupacion de f_j.

El nodo $(p_i,f_j,o_v)\in D$ es un nodo en espera si p_i ha intentado la ocupacion de f_j y esta aun no ha podido realizarse.

Los nodos (p_i,f_j,o_v) y $(p_k,f_r,o_w)\in D$ son del mismo proceso si $p_i=p_k$.

Debido a la incompatibilidad de operaciones, solo uno de los nodos opuestos puede ser activo.

3. CARACTERIZACION DEL INTERBLOQUEO SOBRE EL MODELO

La situación de interbloqueo puede ser detectada sobre el grafo de ficheros declarados.

El teorema siguiente da las condiciones necesarias y suficientes para que aparezca el problema.

TEOREMA

Sea $C=\{D',\Gamma\}$ un circuito del grafo de ficheros declarados $G=\{D,\Gamma\}$, $D'\subset D$. Existe situación de interbloqueo si y solo si:

$\forall (p_i,f_j,o_v)(p_i,f_r,o_w) \in D'$ se cumple:

(p_i,f_j,o_v) es un nodo activo.

(p_i,f_r,o_w) es un nodo en espera.

DEMOSTRACION

$\Rightarrow$)

Supongamos $F=\{f_1,.....,f_m\}$ conjunto de ficheros declarados y $P'=\{p_{i1},.....p_{in}\}$ conjunto de procesos implicados en el circuito.

La aparición de un interbloqueo es equivalente a la situación (15,34):

p_{i1} espera a p_{i2} por f_{j1}
p_{i2} espera a p_{i3} por f_{j2}
.....
.....
p_{in} espera a p_{i1} por f_{jn}

Sea la situación dada, si p_{i1} espera a p_{i2} por f_{j1} $\Rightarrow$

$\Rightarrow$ $\begin{cases}(p_{i1},f_{j1},o_{x11}) \text{ es un nodo en espera.}\\(p_{i2},f_{j1},o_{x21}) \text{ es un nodo activo.}\end{cases}$

Ademas $\Gamma_h(p_{i1},f_{j1},o_{x11})=(p_{i2},f_{j1},o_{x21})$ por ser nodos opuestos, ya que las operaciones son incompatibles y la ocupacion simultanea no es posible.

Igualmente: si p_{in} espera a p_{i1} por f_{jn} $\Rightarrow$

$\Rightarrow$ $\begin{cases}(p_{in},f_{jn},o_{xnn}) \text{ es un nodo en espera.}\\(p_{i1},f_{jn},o_{x1n}) \text{ es un nodo activo.}\end{cases}$

y tambien $\Gamma_h(p_{in}, f_{jn}, o_{xnn}) = (p_{i1}, f_{jn}, o_{x1n})$.

Siempre se cumple: $\Gamma_v(p_{in}, f_{jn-1}, o_{xnn-1}) = (p_{in}, f_{jn}, o_{xnn})$

Luego existe Γ^* tal que:

$$\Gamma^*(p_{i1}, f_{j1}, o_{x11}) = (p_{i1}, f_{j1}, o_{x11})$$

El grafo de ficheros declarados, posee al menos un circuito cuyos nodos cumplen las condiciones dadas.

$\Leftarrow$) Sea $C = \{(p_{i1}, f_{j1}, o_{x11}), (p_{i2}, f_{j1}, o_{x21}), \ldots$
$\ldots, (p_{in}, f_{jn}, o_{xnn}), (p_{i1}, f_{jn}, o_{x1n})\}$

Cada par de elementos $(p_{ik}, f_{jk}, o_{xkk}), (p_{ik+1}, f_{jk}, o_{xk+1k})$ son nodos opuestos por pertenecer al circuito.

Las condiciones implican:

$(p_{ik}, f_{jk}, o_{xkk})$ es un nodo en espera.

$(p_{ik}, f_{jk-1}, o_{xkk-1})$ es un nodo activo.

Luego necesariamente

$(p_{ik+1}, f_{jk}, o_{xk+1k})$ es un nodo activo y tendremos que:

p_{ik} espera a p_{ik+1} por f_{jk}

Tambien $(p_{ik+1}, f_{jk+1}, o_{xk+1k+1})$ debe ser nodo en espera y asi sucesivamente. Llegando a la situación:

p_{i1} espera a p_{i2} por f_{j1}
.....
.....
p_{in} espera a p_{i1} por f_{jn}

Definida anteriormente como situación de interbloqueo

4. PREVENCION

4.1. ESTUDIO DE LAS CONDICIONES PARA EL INTERBLOQUEO

Segun el teorema visto, las condiciones que determinan la situación de interbloqueo de procesos son:

- Existencia de circuitos en el grafo de ficheros declarados.
- Los nodos de un mismo proceso implicados en el circuito de dicho grafo, deben cumplir: uno debe ser nodo activo y el otro, nodo en espera.

Es suficiente con que una de las condiciones dadas

no se cumpla, para que no se desarrolle la situación problema.

El grafo de ficheros declarados se construye con cada orden "declarar" dada desde los distintos procesos y desde ese momento, puede existir circuitos en dicho grafo. Es mas, este contiene todos los circuitos que pueden implicar situaciones de interbloqueo. Cuando se cumpla la segunda condición aparecerá el problema.

Se busca una solución que no rebaje las posibilidades de concurrencia de procesos y por ello, se deja que la ocupación de ficheros se produzca de forma aleatoria.

El algoritmo construido, vela para que la segunda condición no se cumpla, al menos para uno de los nodos implicados en el circuito.

El algoritmo se ejecuta con la "ocupación" del fichero y conserva al menos un proceso por circuito con sus dos nodos implicados en este, en estado no-activo (en espera o pasivo).

Aparece una situación nueva, en la cual van a existir procesos esperando para ocupar ficheros que no estan ocupados por proceso alguno, aunque estén declarados.

Sin embargo, debido a la forma en que se ha definido la función Γ_h, el algoritmo sigue admitiendo varias operaciones compatibles sobre un fichero dado, simultaneamente.

4.2. ALGORITMO

El algoritmo busca un circuito a partir del nodo que quiere pasar a activo, ensayando todos los caminos posibles, pero sin pasar dos veces por una misma columna (nodos del mismo proceso), ya que el camino que pudiera encontrar, se puede generar igualmente desde la columna repetida.

Un camino emprendido puede ser desechado al llegar a una cierta columna:

- Si en esta solo hay un nodo.
- Si los nodos de esa columna implicados en el circuito, estan todos en estado no-activo.

El algoritmo determina si la ocupación se puede realizar (stop=false) o no (stop=true) imponiendose una espera.

Supongamos $P = \{p_1, \ldots\ldots, p_n\}$ conjunto de procesos.
$F = \{f_1, \ldots\ldots, f_m\}$ conj. ficheros decla.

$O = \{o_1, \ldots\ldots\ldots, o_r\}$ conj. de operaciones sobre los ficheros.

```
PREVENCION (p_x,f_x);
Comienzo
Para i:=1 hasta n hacer
   Comienzo
   Buscar (p_i,f_x)#(p_x,f_x) tal que m_i=false y (p_i,f_x)(p_x,f_x)
       sean nodos opuestos;(* enlace horizontal *)
   Si p_i#p_0 entonces
      Comienzo
      m_i:= true; (* nivel ya chequeado *)
      Para todo j_i hacer
         Comienzo
         Buscar (p_i,f_j_i);
         Si (p_i,f_x) o (p_i,f_j_i) son nodos activos entonces
            Comienzo
            Prevencion (p_i,f_j_i);
            Si stop entonces return;
            Fin;
         Fin;
      Fin
      Si no stop:= true y return;
   Fin;
Fin; (Prevencion)
```

La rutina que inicia el proceso:

```
AVOIDANCE (p_0,f_0,stop);
Comienzo
Para i:=1 hasta n hacer m(i):= false;
Stop:= false;
Prevencion (p_0,f_0);
Fin; (Avoidance)
```

4.3. EVALUACION

El algoritmo de prevención presentado, marca los caminos ya recorridos, lo que evita su nuevo desarrollo.

De esta forma, el algoritmo recorre las ramas de una estructura arborescente (sin circuitos) buscando que el elemento terminal sea un nodo del mismo proceso que el de la raiz. El costo es el de un algoritmo que recorre un árbol.

Sea $P=\{p_1,.....,p_m\}$ conjunto de procesos de card= m.

$F=\{f_1,.....,f_n\}$ conjunto de ficheros de card= n.

Se define $F_i=\{f_{j_1},.....,f_{j_{n_i}}\}$ conjunto de ficheros declarados por el proceso p_i y tal que card $F_i = n_i$.

Evidentemente: $F_1 \cup F_2 \cup ..... \cup F_m = F$

$n_i \leq n$

$$\text{costo} = \sum_{i=1,m} n_i \leq m*n$$

En este caso se puede escribir con mas exactitud:

$$\text{costo} = 1 + \sum_{\substack{i=1,m \\ i\#j}} n_i \leq m*n$$

siendo j el número del proceso que intenta la ocupación (raiz)

Ademas, en todo caso la profundidad de llamada de la rutina recursiva construida, no es superior al número m de procesos en ejecución.

El costo del algoritmo es menor al conseguido por otros autores (12,18,22,23,26) y en todo caso aceptable, por lo que se piensa que su aplicación puede ser efectiva.

4.4. EJEMPLO DE COMPORTAMIENTO

Sea $F=\{A,B,C,D\}$ conjunto de ficheros declarados.

$P=\{1,2\}$ conjunto de procesos.

$O=\{\text{lectura,escritura}\}$ conjunto de operaciones.

Siendo escritura incompatible con si misma y con la operación de lectura.

Sean las declaraciones siguientes:

- El proceso 1 declara A,B,C,D, en escritura, lectura escritura y lectura respectivamente.
- El proceso 2 declara B,C,D en lectura, lectura y escritura.

La fig.1 muestra el grafo de ficheros declarados

correspondiente, en el que se puede observar la existencia de un circuito.

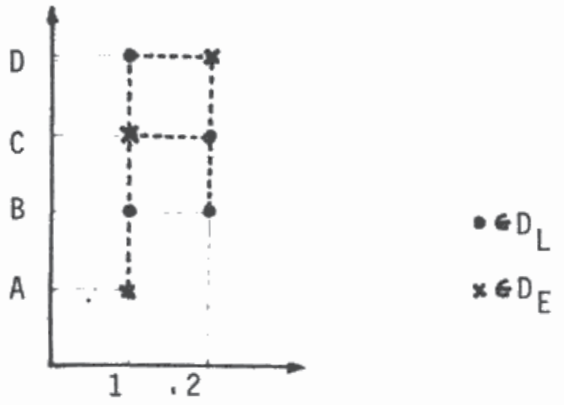
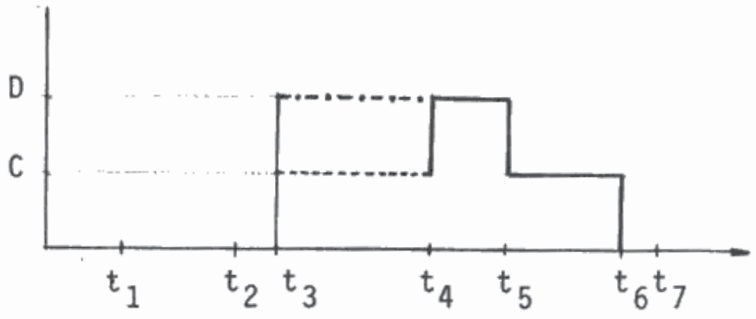


fig.1

Si la ocupación de ficheros se realiza de la forma siguiente:

El proceso 1 ocupa en t_1 el fichero A, en t_2 ocupa B, en t_3 ocupa C, en t_4 ocupa D, en t_5 libera D y en t_6 libera C.

La ocupación de ficheros del sistema vista por el proceso 2 viene representada graficamente en la fig.2.



Ocupación real y simulada ————

real - - - - - - -

simulada -.-.-.-.-.-

fig.2

BIBLIOGRAFIA

(1) V. Ahuja
'Algorithm to check network states for deadlock'
IBM J. Res. Develop. 23,1 (1979)

(2) C,W. Bachman
'The programmer as navigator'
Comm. ACM Vol.16,11 (1973)

(3) P.A. Bernstein, J.B. Rothnie, N. Goodman, C.A. Papadimitriou
'The concurrency control mechanism of SDD-1: A system for distributed data bases'
IEEE Trans. Software Engineering SE-4,3 (1978)

(4) P. Bittman and K. Unterauer
'Models and algorithms for deadlock detection'
Proc. Second International Symposium on Operating Systems, IRIA- Rocquencourt (octubre 1978)

(5) P. Brinch Hansen
'Operating systems principles'
Prentice-Hall (1973)

(6) D.D. Chamberlin, R.F. Boyce, I.L. Traiger
'Deadlock -free scheme for resource locking in a data-base environment'
Proc. IFIP 74 (1974)

(7) E.G. Coffman Jr., M.J. Elphick, A. Shoshani
'Systems deadlocks'
Computing Surveys 3,2 (1971)

(8) E.G. Coffman Jr., p.J. Denning
'Operating systems theory'
Prentice-Hall (1973)

(9) E.W. Dijkstra
'A class of allocations strategies inducing bounded delays only'
AFIPS Conf. Proc. Vol. 40 (1972)

(10) E.W. Dijkstra
'Co-operating sequential processes'
Programming Languages, Academic Press (1968)

(11) G.C. Everest
'Concurrent update control and database integrity'
Data Base Management (J.W. Klimbie, K.L. Koffeman eds.) (1974)

(12) R.O. Fontano
'A concurrent algorthm for avoiding deadlocks in multiprocess multiple resource systems'
Proc. Third Symp. on Op. Systems Principles, SIGOPS Op. Syst. Review 6,1,2 (junio 1972)

(13) B.M. Fossum
'Data base integrity as provided for by a particular data base management system'
Data Base Management (J.W. Klimbie, K.L. Koffeman, eds.) (1974)

(14) D.J. Frailey
'A practical approach to managing resources and avoiding deadlocks'
Comm. ACM 16,5 (1973)

(15) G. Gardarin
'Une solution au probleme du verrou mortel en partage simultane de fichers'
Tesis 3 ciclo. Paris (abril 1974)

(16) G. Gardarin, S. Spaccapietra
'Integrity of data bases: A general lockout algorithm with deadlock prevention'
Comm. Institut de Programmation Paris VI (octubre 1975)

(17) J.R. Garitagoitia
'Soluciones al problema del interbloqueo de procesos concurrentes'
Tesis doctoral, Univ. Pais Vasco (junio 1981)

(18) E.M. Gold
'Deadlock prediction: Easy and difficult cases'
SIAM J. Comput. 7,3 (1978)

(19) B. Goldman
'Deadlock detection in computer networks'
MIT/LCS/TR-185 (setiembre 1977)

(20) J.N. Gray, R.A. Lorie, G.R. Putzolu
'Granularity of locks in a shared data base'
Proc. International Conf. on Very Large Data Bases (setiembre 1975)

(21) A.N. Habermann
'A new approach to avoidance of system deadlocks'
Operating Systems (E. Gelenbe, C. Kaiser, eds.)
Lecture Notes in Computer Science, Vol. 16 (1974)

(22) A.N. Habermann
'Prevention of system deadlocks'
Comm. ACM 12,7 (1969)

(23) A.N. Habermann
'System deadlocks'
Cap. 7 de Current Trends in Programming Methodology, Vol III Prentice-Hall

(24) J.W. Havender
'Avoiding deadlock in multitasking systems'
IBM Systems Journal 7,2 (1968)

(25) P.G. Hebalkar
'A graph model for analysis of deadlock prevention in systems with parallel computations'
Proc. IFIP (agosto 1971)

(26) R.C. Holt
'Comments on Prevention of system deadlocks'
Comm. ACM 14,1 (1971)

(27) R.C. Holt
'On Deadlock in computer systems'
Ph.D. Thesis, Cornell University, Ithaca, N.Y. TR-71-91 (1971)

(28) R.C. Holt
'Some deadlocks properties of computer systems'
Computing Surveys 4,3 (1972)

(29) D.A. Hutchison, J.S. Riordan, S.A. Mahmoud
'A recursive algorithm for deadlock preemption in computer networks'
Information Processing 77 (Proc. IFIP Congress, Toronto, 1977)

(30) P.F. King, A.J. Collmeyer
'Database sharing-an efficient mechanism for supporting concurrent processes'
AFIPS Conf. Proc. Vol. 42 (1973)

(31) D.B. Lomet
'A practical deadlock avoidance algorithm for data bases systems'
Proc. International Conference on Management of data (ACM SIGMOD) (agosto 1977)

(32) P.P. Macri
'Deadlock detection and resolution in a CODASYL based data management systems'
Proc. ACM SIGMOD Conf. (1976)

(33) T.J. Miller
'Deadlock in distributed computer networks'
U. of Illinois Comp. Sci. Dept. (Dic. 1974) report UIUCDCS-R-74-619

(34) J.E. Murphy
'Resource allocation with interlock dtection in a multi-task systems'
AFIPS Conf. Proc. Vol. 33,2 (1968)

(35) G.J. Nutt
'Some applications of finite state automata theory to the deadlock problem'
U. of Colorado Comp. Sci. Dept. (abril 1973) report CU-CS-017-73

(36) D.L. Parnas, A.N. Habermann
'Comment on a deadlock prevention method'
Comm. ACM 15,9 (1972)

(37) G.C. Pirkola
'A file system for a general-purpose time-sharing environment'
Proc IEEE 63,3 (junio 1975)

(38) I. Ramos, J.R. Garitagoitia, J.M. Aramberri
'Algunas soluciones al problema del interbloqueo entre procesos'
IV Congreso de Informatica y Automatica. Madrid (oct. 1979)

(39) D.J. Rypka, A.p. Lucido
'Deadlock detection and avoidance for shared logical resources'
Trans. IEEE, SE-5,5 (set. 1979)

(40) L.C. Sekino
'Multiple concurrent updates'
Proc. International Conf. on Very Large Data Bases, Ed. por D. Kerr (set. 1975)

(41) A. Shoshani, A.J. Bernstein
'Synchronization in a parallel-accessed data base'
Comm. ACM 12,11 (1969)



PARTE III

UNIVERSITAT DE VALÈNCIA
UNIVERSIDAD DE CASTILLA-LA MANCHA
(1978-1987)

PRÓLOGO DE FRANCISCO JOSÉ QULES FLOR

UNIVERSITAT DE VALÈNCIA UNIVERSIDAD DE CASTILLA–LA MANCHA (1978-1987)

Describir el periodo que va desde el año 1978 al 87 de Isidro Ramos Salavert es, sin lugar a dudas, poder narrar una de las etapas más apasionantes de su trayectoria. Tuve la suerte de compartir con él parte de la aventura, primero como alumno suyo en la Facultad de Ciencias Físicas de Valencia y después intentando poner en marcha una nueva Universidad, la Universidad de Castilla-La Mancha.

Conozco a Isidro una mañana de octubre del año 85; yo estaba en quinto de carrera y él era el Catedrático que impartía inteligencia artificial. Sólo nos podía dar clase los viernes porque, como poco después me contaron, era Rector de una nueva Universidad que tenía su sede en Ciudad Real. Si queréis que os diga la verdad, en aquellos momentos para mí tierra desconocida y lejana, nunca pude pensar los años que iba a pasar viajando por sus carreteras.

Pero centrémonos en Isidro como profesor. Fue el primero que nos daba toda la información de la asignatura en otros idiomas, y además no eran libros, ¡sino artículos de investigación de revistas extranjeras! Particularmente de LISP en francés y de PROLOG en inglés. No os voy a engañar, no fue precisamente recibida con mucha alegría dicha novedad, más bien con cierta reticencia y protesta (hay que recordar que entonces los estudiantes universitarios hasta nos movilizábamos de vez en cuando). Pero al final el señor catedrático se salió con la suya y empezamos a traducir aquellos documentos. Pasados los años tengo claro que Isidro fue mi primer profesor con las ideas claras sobre por dónde se iba a manejar el inmediato futuro, y cuál debía ser el camino a seguir por unos casi licenciados con aspiraciones de investigador.

Poco después me lo volví a encontrar en aquellas tierras lejanas. En 1987 obtuve mi primera plaza de profesor ayudante en la Universidad de Castilla—La Mancha, Universidad

donde Isidro era su presidente de la Comisión Gestora, cargo que ocupó del 83 al 88. Años después, en reconocimiento de su labor, fue nombrado Rector Honorario de la Institución.

Para mí una de las actividades académicas más ilusionantes que uno puede llegar a hacer es, sin lugar a dudas, poner en marcha una nueva institución académica. Intentando que todo sea diferente en la nueva institución, que sea como tú siempre has creído que una Universidad debe ser. Y en ello estaba en esos años Isidro. Castilla—La Mancha era una comunidad autónoma de reciente creación, a diferencia de su añorada Valencia, con escasos cuando no nulos vínculos entre sus tierras, es decir, una comunidad unida por la nueva constitución, pero en absoluto por su historia.

Si a eso sumamos que en esos territorios la última Universidad que había existido era la Universidad Pontificia y Real de Santa Catalina en Toledo, extinguida por el Plan General de Estudios de 1845, que reducía las universidades en España a diez y convertía a Toledo en Instituto de Segunda Enseñanza. Es decir, estas tierras llevaban casi siglo y medio sin universidad. Realmente en Castilla-La Mancha sólo existían media docena de centros docentes adscritos a diferentes universidades (Universidad de Murcia, Universidad Politécnica de Madrid y Universidad Complutense) y algunos de ellos con ningún deseo de sumarse a la nueva institución, ya que estaban muy orgullosos de pertenecer a una «gran universidad».

Luego el objetivo de Isidro además de apasionante era particularmente complicado, tanto es así que el primer presidente de la Comisión Gestora, D. Javier de Cárdenas y Chávarri, su único predecesor, aguantó un año en el cargo.

Pero Isidro Ramos de nuevo demostró tener muy claras sus ideas y muy constante su empeño. Luchando contra aquellos que no creían en la nueva universidad, contra aquellos que no veían por qué había que viajar a Ciudad Real para ver al Rector, y con unas escuelas que en general no sabían ni cómo se declinaba el verbo investigar. Cuento como anécdota que en una de las primeras reuniones de profesores que asistí en mi centro solicité comprar la Transaction on Computer, a lo cual el director del centro me contestó «que eso estaba en Ingles y no se lo leía nadie». No hace falta decir cómo terminó el tema. Pues bien, en ese entorno Isidro consiguió poner los cimientos de lo que hoy es mi Universidad y por ello siempre le estaremos agradecidos los que amamos la Universidad de Castilla—La Mancha.

Quiero destacar que en todo este periodo compaginó la investigación, prueba de ello son los artículos que publicó, la docencia en el nuevo campus de Ciencias de Burjasot y ejercer como Rector de una nueva Universidad. Todo ello unido por un Seat 133 del que disfrutaba el Rector y de unas carreteras desastrosas que permitían una rápida conexión Valencia-Ciudad Real en unas seis horas. En un tiempo en que ni las autopistas de la información ni las reales existían en nuestro país.

Pero incluso en esos artículos de investigación se preocupaba de la nueva Universidad y demostraba su capacidad de visión sobre el futuro. Recomiendo la lectura de «The impact of new information technologies on university organization: the example of Castilla-La Mancha», donde habla de lo que van a ser las TICs para las universidades, desde el punto de vista de la organización administrativa, el apoyo a la docencia, la organización de las bibliotecas, la investigación… Todo ello es obvio hoy en día, pero bastante inimaginable a mediados de los ochenta.

Posiblemente yo no sea objetivo al evaluar la actividad académica de Isidro en estos años, es imposible serlo con un buen amigo que además fue mi profesor. Pero creo que cualquiera que analice esa década y vea la actividad desplegada, comprenderá el sacrificio personal que

supone hacer cientos de kilómetros semanalmente, para poner en marcha una Universidad en tierras remotas y mantener la actividad como profesor e investigador en Valencia. Por todo ello lo único que me queda por decir es: GRACIAS, ISIDRO.

Francisco José Quiles Flor

"SEMÁNTICA PARA SPL"

RAMOS, I.

En este trabajo se presenta un modelo de semántica calculatoria para el lenguaje SPL. También se presentan varias interpretaciones del lenguaje SPL, tanto en el caso en que se incluyan restricciones como en el caso en que no se incluyan. Finalmente se enumeran las consecuencias y las ventajas deducibles del SPL y de su modelización semántica que se ha descrito.

1. INTRODUCCION

Entre las distintas aproximaciones a la definición de la semántica de un lenguaje de programación: Interpretativa, calculatoria, denotacional, axiomática,... las hay que tienen en cuenta el aspecto dinámico o algorítmico de la ejecución de un programa. Es el caso de la interpretativa y calculatoria. Otras como la denotacional y la axiomática prescinden de esta característica lo que las hace inadecuadas para describir los aspectos "ejecución" de los programas: programas infinitos asociados con programas de resultado indefinido (erróneo),..

En este trabajo se formaliza la semántica de un lenguaje de programación: SPL (Structured Programming Language) /1/, /2/, /3/, /4/ de acuerdo con la aproximación calculatoria /8/ puesto que pensamos que el aspecto algorítmico de un programa es por el momento una realidad. Tendremos por lo tanto en cuenta el aspecto "ejecución" de un programa o cálculos.

En el trabajo usamos pues la notación y orden expositivo de los autores de la aproximación calculatoria a la semántica /10/.

2. CONCEPTO DE PROGRAMA EN SPL

Un programa es en SPL una forma eficaz de especificar el conjunto de ejecuciones (cálculos) que puede generar.

Una ejecución o cálculo es una cadena (o secuencia) de modificaciones (tratamientos) sobre la estructura de la información asociada con el programa.

Cada modificación (también llamada acción) viene descrita en el programa por una instrucción (tratamiento terminal).

Si una ejecución es una cadena de elementos del alfabeto de acciones (pertenecientes al repertorio de un procesador real o virtual), el conjunto de ejecuciones de un programa será un subconjunto de cadenas o sea un lenguaje sobre dicho alfabeto. Una forma cómoda y eficaz de especificar lenguajes es mediante gramáticas y en particular gramáticas algebraicas Tipo 2 de Chomsky (T2) dotadas de cierta función de etiquetaje en las reglas: S - gramáticas /4/.

El aspecto algorítmico viene pues descrito por la "generación" desde el axioma de las frases terminales (cálculos).

La interpretación se realiza a dos niveles:

- Tratamientos terminales.
- Predicados.

La estructura de información asociada con SPL depende en las implementaciones realizadas del lenguaje objeto elegido: ALGOL, COBOL, FORTRAN ...

Sintacticamente el aspecto algorítmico (ESTRUCTURA DE UN PROGRAMA) viene dado por el sis

- I. Ramos Salavert del Centre d'Informàtica de la Universitat de València. Dr. Moliner, s/n. Burjasot (Valencia)
- Article rebut el Gener de 1980.

Ramos, I. Semántica para SPL. Questió. Vol. 4. nº 1.1980.

tema de ecuaciones:

PROGRAMAS = REGLAS .
REGLAS = REGLA ; REGLAS ∪ REGLA
REGLA = IZDA ::= DERECHAS
DERECHAS = DERECHA | DERECHAS ∪ DERECHA
DERECHA = CADENA PREDICADO
CADENA = TRATAMIENTO ∪ TRATAMIENTO CADENA
TRATAMIENTO = TERMINAL ∪ NO TERMINAL
TERMINAL = NIL ∪ a ∪ b ∪ ... ∪ z
NO TERMINAL = ℙ ∪ A ∪ B ∪ ... ∪ Z
IZDA = NO TERMINAL
PREDICADO = DIGITO ∪ DIGITO PREDICADO
DIGITO = 0 ∪ 1 ∪ 2 ∪ ... ∪ 9

Los símbolos subrayados son los que aparecen en las cadenas terminales (símbolos terminales), los no subrayados son símbolos auxiliares usados para definir dichas cadenas (símbolos no terminales).

En las S - Gramáticas descritos por esta metagramática:

ℙ ∈ NO TERMINAL es el axioma

NIL ∈ TERMINAL es la instrucción vacía.

La interpretación de los terminales y predicados se da separadamente y en el lenguaje - de programación escogido.

3. CONCEPTO DE ESTADO DE MEMORIA Y DE MODIFICACIONES

Estructura de información asociada con SPL.

Un programa describe unos tratamientos sobre unos objetos.

El orden de ejecución de los tratamientos -- viene descrito por el aspecto algorítmico, - mientras que en la descripción de los objetos inicialmente se introdujo el concepto de vector de estado y en la actualidad se habla de datos abstractos.

Una de las ventajas de SPL es el que se puede prescindir de una interpretación detallada de los objetos y de los tratamientos para describir su semántica.

Necesitamos suponer una memoria (estructura de la información) /9/:

M = {O , L , M}

O = Conjunto de objetos
L = Conjunto de funciones de acceso
M = Conjunto de modificaciones

susceptible de estar en un conjunto de estados (valor en un momento dado de las funciones de acceso):

E = {eo, e, ..., en, Ω},

siendo Ω el estado indefinido que cumple

∀m∈M m(Ω) = Ω

suponemos además un conjunto de modificaciones

M = {mo, ..., mk, NIL}

tales que:

∀m∈M ; m:E → E

Las modificaciones se describen en un programa mediante instrucciones (TERMINALES). La - modificación identidad se representa mediante NIL ∈ TERMINALES y la hemos notado NIL -- cumpliendo:

e ∈ E igual (NIL(e),e)

Siendo el igual la igualdad entre estados de memoria que suponemos una función primitiva definida.

4. SEMANTICA DE SPL

No vamos a describir paso a paso la ejecución de un programa sino que globalmente especificaremos el conjunto de cálculos con él asociados, siendo éste conjunto el valor semántico del programa considerado (semántica calculatoria).

Un cálculo no lo definimos pues como una sucesión de estados de memoria (semántica interpretativa) sino como una sucesión de modificaciones elementales (representadas por -- instrucciones) de la estructura de información asociada al lenguaje (memoria).

4.1 NOCIÓN DE CÁLCULO

Intuitivamente un cálculo es la ejecución en secuencia de un conjunto de acciones elementales.

Formalmente un cálculo es una secuencia finita (o no) de elementos del conjunto (M) de modificaciones elementales de la estructura de información asociada al lenguaje.

Toda modificación $m \in M$ se describe mediante una instrucción a $\in$ TERMINAL.

Llamaremos M^* al monoide libre sobre M y M^ω al conjunto de cálculos infinitos. Por M^∞ notaremos

$$M^\infty = M^* \cup M^\omega$$

(es también un monoide al que se le extiende la operación de concatenación).

Todo cálculo finito

$$C = m_1 . m \ldots m_n$$

se interpreta como la modificación:

$$m_n . m_{n-1} \cdots m_1$$

de la estructura de la información, interpretando la concatenación como la composición según el orden de escritura de las modificaciones elementales:

$m_i \in M$, $\quad i=1,n.$

Usaremos pues indistintamente las dos notaciones.

4.2 COMPOSICIÓN SECUENCIAL DE CÁLCULOS

La concatenación en M permite expresar la ejecución secuencial de los cálculos.

4.3 COMPOSICIÓN CONDICIONAL E ITERATIVA

Con estas formas de composición se asocian un conjunto de cálculos. Por ejemplo a la regla:

A ::= a 1 | b 2

le asociamos el conjunto de cálculos:

{igual (Eval'(1),verdad).Eval(a)} $\cup$

$\cup$ {(Eval'(2),verdad).Eval(b)}

ya dijimos que igual(u,v) es la modificación elemental, restricción de la aplicación identidad al conjunto de estados de memoria para los que u es igual a v. Una definición más precisa sería:

$$\text{igual}(u,v)(e) = \begin{cases} e & \text{si } u=v \text{ se verifica en el estado } e \\ \Omega & \text{sino} \end{cases}$$

Por otra parte, las funciones Eval se definen como:

Eval: TERMINALES → M

Eval': PREDICADO → {verdad,falso}

Materializan la Interpretación que se realiza.

Un cálculo es pues un elemento de $P(M^\infty)$: conjunto de las partes de M^∞. Todos los conjuntos de cálculos se construyen partiendo de las modificaciones elementales por aplicación de dos leyes de composición internas en $P(M^\infty)$:

- Concatenación "." extensión natural de la de M^∞ y definida por:

$$C_1 . C_2 = \{\alpha.\beta \mid \alpha \in C_1,\ \beta \in C_2\} \qquad C_1 \in P(M^\infty),\ C_2 \in P(M^\infty)$$

- Reunión conjuntista "$\cup$"

Cálculos asociados con una regla

El conjunto de cálculos asociados con una regla depende de los cálculos asociados a las partes derechas que las componen. Podemos definirlo por las siguientes ecuaciones:

a) Condicional

A la regla

P ::= α 1 | β 2

siendo: V = TERMINAL ∪ NO TERMINAL
P ∈ NO TERMINAL
α,β ∈ V^+
1,2 ∈ PREDICADO

le corresponde el sistema

A(P):

νP = igual(Eval'(1),verdad).$\nu(\alpha)$ ∪

∪ igual(Eval'(2),verdad).$\nu(\beta)$

Donde A(P) es el sistema de ecuaciones -- asociadas con P∈N de acuerdo con las reglas de las que es parte izquierda y $\nu(\alpha)$ es el conjunto de modificaciones asocia-- das con la cadena de instrucciones α.

Definimos $\nu(\alpha)$, $\nu(\beta)$ por recurrencia sobre la longitud α y β (notada respectivamente como $\|\alpha\|$ y $\|\beta\|$) en la forma:

$\nu(\alpha)$ = Si $\|\alpha\| \leq 1$ entonces si α ∈ TERMINAL entonces Eval (α)

sino (* α ∈ NO TERMINAL *) A(α)

f si

sino (* α será: α = σ.α', con α, α' ∈ V*, σ ∈ V y $\|\alpha'\| \leq \|\alpha\|$ *)$\nu(\sigma).\nu(\alpha')$

f si

Las cadenas comprendidas entre (* y *) son comentarios y se incluyen en la ecuación solo a efectos aclarativos.

El sistema A(P) aparece formado por una - ecuación que liga la incógnita $\nu(P)$ aso-- ciada a P, a sus partes derechas: $\nu(\alpha)$ y $\nu(\beta)$.

En la definición de $\nu(\alpha)$ aceptamos que si $\|\alpha\| < 1$ entonces Eval(α)=NIL.

Como $\|\alpha\|$ es finito, obtenemos una sucesión monótona decreciente que asegura que el - proceso de sustitución termina.

Solo en el caso en que existe:

A ∈ NO TERMINAL tal que A ::= A

o de una descripción explícita de un cálculo infinito como:

A ::= a A

el proceso de sustitución no termina.

b) Iteración vendrá dada en SPL por:

P ::= σ P 1 | NIL 2

donde P ∈ NO TERMINAL, σ∈V^+
1,2 ∈ PREDICADO
NIL ∈ TERMINAL

le corresponde el sistema:

$\nu(P)$ = igual(Eval'(1),verdad).$\nu(\sigma).\nu(P)$ ∪

∪ igual(Eval(2).verdad).Eval(NIL)

que traduce la equivalencia (equivalencia entendida como igualdad de las cadenas de modificaciones asociadas y usando una notación de lenguaje de programación)

$\nu(P)$ ≡ si Eval'(1) entonces $\nu(\sigma).\nu(P)$

sino Eval(NIL)

A toda regla y en general a todo programa ℙ se le asocia un sistema de cálculo A(ℙ) sobre $P(M^{\infty})$ en la forma:

$$A(\mathbb{P}): \begin{cases} \nu(\mathbb{P}) = f(\nu(p_1),\ldots,\nu(p_k)) \\ A(p_i) \ ; \quad i=1,2,\ldots,1 \end{cases}$$

siendo las p_i las partes derechas de ℙ ($p_i \in V^+$, i=1,k).

La función f se compone de las "funciones de base"

- Concatenación "."
- Reunión "∪"

Para este sistema hay siempre una solu--

ción mínima /10/.

La componente relativa a ℙ en esta solución mínima es el conjunto de cálculos -- que llamamos valor semántico de ℙ.

5. RESTRICCION E INTERPRETACIONES DE SPL

Con la finalidad de hacer de SPL un lenguaje de programación se han realizado:

- Una restricción
- Varias interpretaciones

Restricción

Dada una regla, sean 1, 2, ... K los predicados de sus partes derechas. Se cumple que:

$$\{\bigcup_{i=1}^{k} \text{Eval'}(i) = \text{verdad};$$

siendo $\cup$ el "o" lógico, y

$$\forall i,j \begin{cases} 1 \leq i \leq k \\ 1 \leq j \leq k \end{cases}$$

$$\text{Eval'}(i) \text{ Y } \text{Eval'}(j) = \text{falso } \}$$

con lo cual obtenemos SPL - secuencial /1/, /2/, /3/, /4/, /7/.

La instrucción condicional

(si ... entonces ... sino ...)

se obtiene haciendo en:

P ::= α 1 | β 2 que

Eval'(2) = NO Eval'(1)

La instrucción iterativa se obtiene haciendo que

P ::= σ . P 1 | NIL 2 que

Eval'(2) = NO Eval'(1)...

Sin restricción

Si no incluimos la restricción, varias interpretaciones son posibles:

1) SPL con vuelta atrás

El hecho de que no se cumpla la restricción en el proceso de obtención por sustitución del cálculo partiendo del sistema de ecuaciones se debe a que en ese momento de la ejecución esta información no está disponible, lo que origina varios cálculos posibles. De ellos solo uno es correcto lo que se averigua al final del proceso de sustitución recursivamente: (Descenso recursivo desde el axioma) /6/.

2) SPL no determinista

No se cumple la restricción y varios cálculos pueden originarse en cada sustitución. Esta línea está siendo desarrollada con propósito de estudiar la equivalencia semántica entre las ejecuciones de programas no interpretados, con fines de optimización...

3) SPL concurrente

No se cumple la restricción y varios cálculos pueden originarse en una sustitución. Estos cálculos son procesos concurrentes en la realización de la tarea especificada en el programa /5/.

Interpretaciones realizadas

Las interpretaciones del SPL se han realizado especificando las funciones Eval y Eval' en distintos lenguajes de programación

- ALGOL 60 /6/
- FORTRAN /7/
- COBOL ANSI74 /7/
- PL/I /7/

Dando origen a los correspondientes sistemas de programación operativos sobre UNIVAC-1180.

Las interpretaciones a nivel objetos se realizan siempre en el lenguaje escogido.

La introducción de datos Abstractos en SPL - /5/ se está realizando al resolver un problema planteado por C. Pair (comunicación personal) a SPL:

¿Si queremos realizar la misma modificación m sobre objetos diferentes, tenemos que dar

un nombre distinto a cada instrucción que se evalúa a esa modificación o aceptamos parámetros en las instrucciones TERMINALES?.

La introducción de Datos abstractos resuelve el problema mediante el uso de parámetros -- que son objetos:

"Una modificación es la ejecución de un procedimiento: P sobre un objeto O del tipo abstracto T". Obviamente el procedimiento es -- uno de los del tipo abstracto. Por lo tanto la notación será:

.uso : P (O)

.declaración (en un T.dato abstracto)

procedure P(O:T);<cuerpo proc.>;

Un programa en SPL aparece entonces como:

```
Program   PRO     (input, output);
const...;
Abstract Data Type T1; T2;...Tn;
Objets O1 : T1  ;  O2 : T2  ; ......;
Procedure P1(O:T;O':T';...;O":T");
          . <cuerpo proc.>;
          .  (*  inter. objetos  *)
          .  Pn...,
          .
begin     S - Gramática   end.
```

En este caso la estructura de la información asociada con SPL vendrá dada por la reunión de los objetos definidos en cada programa y las funciones de acceso y modificaciones declaradas en los tipos abstractos a los que - pertenecen.

6. CONCLUSIONES

Como conclusiones a este trabajo y en general a SPL podemos enunciar las siguientes:

- La separación clara entre el aspecto esquema de control e interpretación de un programa permite dedicarse especificamente a cada uno de ellos aumentando la legibilidad de los programas y también su "manipulabilidad" formal.

- La proximidad a la Teoría de lenguajes del esquema de control de SPL permite la reducibilidad de algunos problemas a otros en este campo tan bien definido de la Informática (y con tantos resultados).

- Tal y como está pensado, SPL es un "mecano" desde el punto de vista de los lenguajes y la programación: programación con vuelta - atrás, no determinista, concurrente...

- La metodología de la programación que permite se encuentra entre los "niveles de -- abstracción" de Dijkstra y los "refinamientos sucesivos" de Wirth.

- Permite una programación bastante "natural" ya que pensamos que:
  . el aspecto estático y dinámico se mantienen lo más próximo posible,
  . la mente humana está acostumbrada al pensamiento secuencial (pensar en varias cosas a la vez es difícil) para lo que SPL está perfectamente adaptado.

7. BIBLIOGRAFIA

/1/ RAMOS, I., MOYA, R.: "Metaprogramación - estructurada". Congreso Nacional de Informática y Automática. Madrid 1975.

/2/ RAMOS, I., MOYA, R.: "SPL (Structured -- Programming Language): A Language for -- structured programming". Annual IFIP - III Conference. Lorient 1977.

/3/ MOYA, R., RAMOS, I.: "EPL: Un lenguaje - en español para la programación estructurada". Proceso de Datos, n.70, Julio 1977, pp. 7-12.

/4/ MOYA, R., RAMOS, I.: "SP-Grammars (Con-- text-Free Grammars for Structured Pro-- gramming)". ACM-Sigplan, v.14, n.1, 1979, pp. 69-76.

/5/ RAMOS, I.: "Un modelo de esquema de control para programas concurrentes". Conferencia Centro de Cálculo de la Univ. Politécnica. Barcelona 1979.

/6/ AGUIRRE, J.: "Sistema de programación estructurada con vuelta atrás". Tesina Universidad de Bilbao. 1978.

/7/ MOYA, R.: "Diseño e implementación de un sistema de Programación Estructurada". -

Universidad de Bilbao. 1977. Tesis Doctoral.

/8/ FINANCE, J.P.: "Contribution à la Formalisation d'un Langage de Programmation, Application à ALGOL 68". Thèse de Spécialité. Univ. de Nancy I, 1974.

/9/ PAIR, C.: "Les structures d'information et leur representation en memoire". Ecole d'été d'informatique de l'AFCET. Ales 1971.

/10/ LIVERCY, C.: "Theorie des programmes". Dunod. 1978.

"UN LENGUAJE DE PROGRAMACION CON TIPOS ABSTRACTOS INTERPRETADOS EN PROLOG: OTRA FORMA DE COMPILAR"

por: Isidro Ramos Salavert

Departamento de Electrónica e Informática.
Facultad de Ciencias Físicas.
Universidad de Valencia.

RESUMEN: Cobran interés cada vez más en Programación, lenguajes no imperativos que sean herramienta de expresión del problema - que queremos resolver y no de la forma como queremos resolverlo. Ideas ya clásicas, como el uso de cláusulas de Horn como lengua je de programación (Prolog)(1), han perdido su aire académico/ y condicionan ya, no sólo el software (entornos de programación y su especificación, B.D. relacionales) sino también el hardwa- re (la llamada 5ª generación de ordenadores).

Este trabajo es una síntesis, en forma de un lenguaje de progra mación "clásico" (con tipos abstractos como parte declarativa y con términos no interpretados del álgebra (que es el tipo abs-- tracto) como parte imperativa) y la programación me(nte cálcu lo de predicados de 1er. orden (Programación lógica: Prolog).

El lenguaje presentado es una maqueta, así como su traductor a- sociado. No obstante, la realización "industrial" de un prototi po se apoya en técnicas suficientemente difundidas como para / ser motivo de presentación en este trabajo.

1.- INTRODUCCION

Cobran interés cada vez más en Programación, lenguajes no imperativos que sean herramienta de expresión del problema que queremos resolver y no de la forma como queremos resolverlo. Ideas ya clásicas, como el uso de cláusulas de Horn como lenguaje de programación (Prolog) (1), han perdido su aire académico y condicionan ya, no sólo el software (entornos de programación y de especificación, B.D. relacionales) sino también el hardware (la llamada 5ª generación de ordenadores).

Este trabajo es una síntesis, en forma de un lenguaje de programación "clásico" (con tipos abstractos como parte declarativa y con términos no interpretados del álgebra (que es el tipo abstracto) como parte imperativa) y la programación mediante cálculo de predicados de 1er. orden (Programación lógica: Prolog).

El lenguaje presentado es una maqueta así como su traductor asociado. No obstante, la realización "industrial" de un prototipo se apoya en técnicas suficientemente difundidas como para ser motivo de presentación en este trabajo.

2.- EL LENGUAJE

El lenguaje LALI (Lenguaje Algebraico Lógico Interpretado) está basado en su parte declarativa de objetos en la Teoría de Tipos Abstractos Algebraicos (2) con axiomas ecuacionales. La parte imperativa viene dada por un término del álgebra que es el Tipo Abstracto.

Dicho término es evaluado en el álgebra de términos (sin interpretar) de acuerdo con los axiomas ecuacionales, dando lugar a un término en forma normal (3). Veámoslo mediante un ejemplo:

Supongamos que queremos resolver

Ramos, I. Un lenguaje de Programación con tipos Abstractos interpretado en PROLOG. Otra forma de compilar. Revista de Informática y Automática. 1984.

problemas en el universo de las pilas. Definamos dicho universo mediante el tipo abstracto genérico PILA en la forma normal:

```
Tipo PILA (V);
Var v: V; p: PILA
Perfil
        pilavacia: ---> PILA
        push: PILA x V ---> PILA;
        pop: PILA ---> PILA;
        top: PILA ---> V;
        vacia: PILA ---> BOOL
Axiomas
        pop (pilavacia)= error;
        pop (push (p,v) )= p;
        top (pilavacia)= error;
        top (push (p,v) )= v;
        vacia (pilavacia)= true;
        vacia (push (p,v) )= false;
```

Un programa en LALI sería:

```
        Programa ejemplo;
        Tipo PILA (V); /* Definición del
                          tipo */
        Var P: PILA (V); A, B: V;
        begin push (pop (push(P,A)),B)end
```

con resultado trivial: push (P,B)

3.- EL TRADUCTOR

Tomamos como lenguaje objeto Prolog. La traducción se realiza de acuerdo con el siguiente esquema:

Objetos LALI

constantes: pilavacia, error, ...
variables: p, v
Operaciones: push, pop, ...
términos: pop (pilavacia)
Axiomas: pop (pilavacia)= error

Objetos Prolog

átomos: pilavacia, error, ...
variables: P,V (sintaxis Prolog)
functores: push, pop, ... declarados como operadores
términos: pop (pilavacia)
hechos: su (pop, pilavacia, error).

/* su: símbolo unario */

Si permitimos axiomas condicionales del estilo:

```
top(push(p,v))= si talla(p)< max entonces v
                              sino top (p);
```

Su traducción en Prolog sería obvia:

```
su(top, push (P,V),V): -talla (P,T), T<Max.
su(top, push (P,V), top (P): -talla (P,T),T>=Max.
```

De acuerdo con este esquema la traducción del tipo PILA (V) sería:

Tipo Abstracto

```
a) Tipo PILA (V)
   Var v: V; P: PILA (V);
   Perfil
        Pilavacia: ---> PILA;
      b) push: PILA x V ---> PILA;
      c) pop: PILA ---> PILA;
      d) top: PILA ---> V;
      e) vacia: PILA ---> BOOL;
```

Versión Prolog

```
a') ?- consult ( 'PILA(V)').
b') ?- op (65,fy y, push).
c') ?- op (65,fy,pop).
d') ?- op (65,fy,top).
e') ?- op (65,fy,vacia).
```

Axiomas

```
f) pop (pilavacia) = error
g) pop (push (p,v)) = p
h) top (pilavacia) = error
i) top (push (p,v)) = v
j) vacía(pilavacia) = true
k) vacía(pilavacia) = false.
```

```
f') su (pop,pilavacia, error).
g') su (pop,push(P,V),P).
h') su (top, pilavacia,error).
i') su (top,push(P,V),V).
j') su (vacia,pilavacia,true).
k') su (vacia,push(P,V),false).
```

En su caso los simétricos sufrirían un tratamiento análogo.

OBSERVACIONES

- El nombre del tipo: PILA (V) será el de la Base de Datos Prolog en el que se almacenan las cláusulas resultado de la traducción.

- La declaración de variables nos lleva a la construcción de la tabla de símbolos del traductor lo que nos permitirá, durante la traducción, realizar una comprobación de tipos estática.

- Las constantes (operadores 0-arios) se almacenan en la tabla de constantes con la finalidad de ser detectadas en lo sucesivo por el traductor.

- Los operadores declarados en el perfil (unarios, binarios,..) nos proporcionan la declaración Prolog usando el predicado predefinido "op" y dando su: prioridad, asociatividad y nombre en la forma usual Prolog.

Asignamos una prioridad "65" en el ejemplo, ya que de acuerdo con las prioridades de los operadores predefinidos Prolog usuales es la que corresponde.

- Los axiomas ecuacionales nos originan una colección de hechos Prolog. Si el operador es unario, el functor lo llamamos su, si binario sb, ... de acuerdo con los visto en el ejemplo (cualquier otro nombre es posible).

Una consideración adicional: si / los axiomas introducen constantes, y funciones, éstas deben de ser conocidas por el intérprete Prolog o definidas por el usuario así: el axioma.

pop(push(p,v)) = si talla(p) max entonces p
sino pop (p);

Originaria en Prolog:

su(pop,push(P,V),P): -talla(P,N),N < Max.

su(pop,push(P,V),pop(P)): -talla(P,N),not(N<Max)

Y además la definición de:

Talla(pilavacia,o).

Talla(push(P,V),M):-talla (P,N), M is N +1.

Talla (pop(P,V),M): -talla(P,N),
N is N - 1.

La traducción del término podría realizarse en un dominio de interpretación. En el caso del ejemplo podrían ser las listas, lo que obligaría a introducir el predicado general de evaluación: eval definido en la forma:

eval (pilavacia,( )).

eval (pop (p),T): - eval (p,(H|T)).

eval (push (A|T),B), (B,A | T)).

eval (push ((P,B),R):-eval(P,M),append((B),M,R).

lo que nos llevará, en el ejemplo a:

?-eval(push(pop(push(P,A)),B),X).

P = -27, A = -28, B = -29

X = (-29)

Suponiendo que -27, -28, ... son los identificadores internos asignados por Prolog a las variables correspondientes. Pero la respuesta que nosotros queremos es el término reescrito y en forma normal, de acuerdo con los axiomas y sin pasar por ninguna interpretación:

?-eval(push(pop(push(P,A)),B),X).

P = - 27, A = - 28, B = - 29

X = push (- 27, - 29)

Por lo tanto la función de evaluación deberá ser (teniendo en cuenta además la aridad: N arbitraria de los operadores del tipo):

```
eval(E,E): -atomic(E),!.

eval(E,F): -
        functor (E,-,N),
        evalx (E,N,F).

evalx(E,N,P):-N=1,E=..(op,Ar),eval(Ar,x),su(Op,X,F)
evalx(E,N,F):-N=2,E=..(op,Ai,Ad),eval(Ai,X),
              eval(Ad,Y),sb(Op,X,Y,F).

    /* para cada aridad presente */
```

este predicado eval será incorporado automáticamente por el traductor en el fichero "PILA(V)" en el que almacenamos el programa objeto.

4.- FUNCIONAMIENTO DEL SISTEMA

Dispondríamos de un sistema conversacional típico cuyo funcionamiento podemos esquematizar mediante la expresión regular:

[Pedir la definición de los tipos [Pedir un término a evaluar. Dar el resultado]+]+

El sistema dispondrá del conjunto de tipos predefinidos usuales: Bool, NAT, ... más los tipos propios del usuario en forma de Base de Datos Prolog y de acuerdo con lo expuesto en el punto 3. El traductor incorporará a la Base de Datos del usuario ("user") aquéllas definiciones de tipos que se correspondan con identificadores que aparezcan en el perfil (los perfiles) y que no estén definidos como variables, tipo de interés,...

Ahorramos los aspectos de "cosmética" en cuanto a los problemas de E/S del sistema por ser estrictamente técnicos y facilmente solubles en Prolog.

Obviamente el sistema incorpora el predicado intérprete

Lali: -read(término), eval (término,R),write(R).

y su funcionamiento

?- Lali.

5.-CONCLUSIONES

El trabajo aporta:

- Un lenguaje fuente clásico (pero moderno: Tipos Abstractos, reescritura ...).

- Un lenguaje objeto de actualidad (pero clásico: Prolog).

- Una forma de compilar sencilla y potente.

El programa objeto goza de todos los inconvenientes de Prolog: ineficacia de los intérpretes Prolog (en vías de estudio) (4). Pero globalmente el sistema supone otra forma de compilar y abre perspectivas cómodas en la implementación de lenguajes.

6.- REFERENCIAS

(1) CLOCKSIN W.F., MELLISH C.S.,"Programming in Prolog"
Springer Verlag, 1981.

(2) ADJ, "Abstract data Types as initial álgebras and correctness of / data representations".
Proc. Conf. on Comp. Graphics, Pattern Recognition and Data Structure, 1975.

(3) HUET G., "Confluent reductions: / abstract properties and applications to term rewriting systems".
IRIA, 1982.

(4) SATO, T., TAMAKI, H., "Enumeration of success pattern in logic programs".
10 th. ICALP. Barcelona, 1983.

(5) DERANSART, P.,"Derivation de programmes PROLOG a partir de specifications Algebriques".
Note interne INRIA, 1982.

(6) VAN EMDEN, M.H., MAIBAUM, T.S.E., "Equations compared with clauses for specification of abstract data types"
Advances in data base Theory(vol.1)
Eds. Gallaire & Minker, 1980.

ESPECIFICACION DE PANTALLAS

Por: Isidro Ramos Salavert

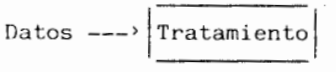
Centro de Informática.
Universidad de Valencia.

Existen realizaciones (1) de sistemas para definir el objeto gráfico complejo que es una pantalla. Asimismo se han producido formalizaciones (2). En este trabajo se relaciona la E/S mediante pantalla interactiva con los Tipos Abstractos Algebraicos y se describe un sistema para su utilización.

1.- INTRODUCCION

Podemos ver un programa como un conjunto de objetos de ciertos tipos sobre los que se realizan tratamientos.

Los módulos de tratamiento permiten ocultar (encapsular) el cómo se realizan en favor del qué hacen. Aparecen como una caja negra de la función que realizan.



que en notación informática:

```
Function Tratamiento (Datos:Td): Tr;
/* Declaración */  :
Resultados:= Tratamiento (Datos)
/* Uso */
```

Siendo Td el tipo de datos y Tr el de los resultados.

La notación informática obliga a introducir los tipos de los objetos.

En matemáticas esta convención no suele ser necesaria pues las Algebras en que se trabaja son homogéneas y de sort conocido en todo momento.

El tratamiento (visión macroscópica) es un operador:

Tratamiento: Td ---> Tr

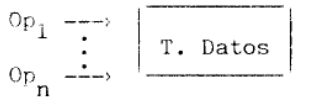
del tipo de los resultados y la resolución del problema se realiza mediante el término elemental:

Tratamiento (Datos)

Es normal descomponer el tratamiento (visión microscópica) y obtener un término en el que participan operadores de tipos menos completos que el tipo del problema. O en forma más precisa damos el término y el método de cálculo expresados en un lenguaje de programación: programa.

Ramos, I. Especificación de Pantallas. Revista de Informática y Automática, 1984.

Un tipo de datos oculta (encapsula) objetos: nos despreocupamos de cómo se representan dichos objetos en favor de qué podemos hacer con ellos usando el conjunto de operadores del tipo:



La dualidad Tratamiento / T. Datos es - clara.

Pero los objetos además de las manipulaciones que hemos llamado tratamientos sufren otras operaciones que por su especificidad pueden considerarse independientemente: son las operaciones de E/S.

Ya lenguajes evolucionados tratan tipos específicos a la E/S.

La forma de E/S más evolucionada es la interactiva, realizada mediante "pantallazos", pudiendo considerar las otras como restricciones de ésta.

Si ha existido una preocupación por los tipos de datos orientados a tratamientos: pilas, colas, tablas,.. no ha ocurrido así con la E/S que ha sido sistemáticamente tratada pobremente.

Nuestro propósito es introducir tipos de datos orientados E/S, e incluso introducir un tipo único genérico (la limitación y especificidad de los tratamientos lo justifica) que será el de cualquier objeto que sufre un tratamiento de E/S y que llamaremos: pantalla.

2.- EL TIPO PANTALLA COMO TIPO GENERICO UNICO DE E/S.

La visualización de datos o resultados mediante el tipo pantalla implica una manipulación de los objetos de acuerdo con criterios diferentes a los que se siguieron para calcularlos (tipos diferentes). Además las acciones a realizar (operadores del tipo) tienen una clara especificidad.

Consideraremos la descripción del tipo pantalla a tres niveles:

a) Nivel abstracto: Definiremos lo que es el tipo genérico "pantalla" con sus sorts, operadores, ... adecuados, independientemente de toda representación o ligadura por parte del hardware usado.

b) Nivel dispositivo: Escojeremos representaciones concretas mediante tipos predefinidos o del problema para cada uno de los tipos constituyentes de pantalla. Dando (explicita ó implicitamente) su bidimensionalidad...

c) Nivel dispositivo: Por cada tipo elemental representado daremos las particularidades que aporta el hardware: brillo ó no, blinking.

Una vez especificado el tipo pantalla, una instanciación (puesto que es genérico) será un cierto formato de pantalla y una pantalla será una variable del tipo pantalla instanciado.

La asignación de valores a los campos nos dará constantes del tipo instanciado.

La correspondencia entre las variables de los tipos orientados a tratamientos y las variables del tipo pantalla instanciado se realizará mediante una interface que servirá para dar detalles de la forma de transferencia de la información, acciones asociadas ...

Tenemos dos formas de asignar esta información:

i)- Asignación: obliga a introducir un conjunto de asignaciones solo a estos efectos en el programa / del usuario.

ii)- Correspondencia parámetro actual-formal al realizar la visualización mediante la interface.

Veamos con un poco más de detalle la definición del tipo pantalla.

Nivel abstracto.

El tipo pantalla es un producto de tipos (análogo al record PASCAL) Al definirlo debemos tener en cuenta este hecho proporcionando constructores y los operadores adecuados:

Tipo pantalla (T_1,...,T_n); var T_i:T; p: pantalla;

Operadores

Pantalla vacía: ---› Pantalla;
añadir: T_i x pantalla ---› Pantalla;

Campos: pantalla ---› integer,Nat
Longitud: pantalla ---› integer,Nat
Retirar: T_i x pantalla ---› pantalla;

:

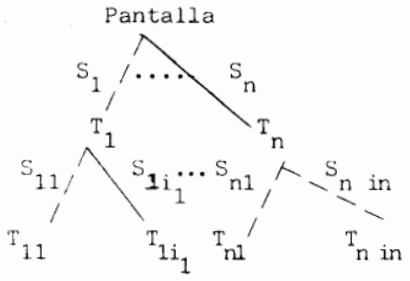
Axiomas

Campos (pantalla vacía) = 0;
Longitud (pantalla vacía) = 0;
Campos (añadir(T,P)) = # Campos (P)+1;
Longitud (añadir (T,P)) = longitud (P)+long(T);
:

No necesariamente la realización de estos operadores se hará mediante un lenguaje más o menos enriquecido sintácticamente: el uso de las facilidades del material: desplazamiento del cursor: ↑ ↓ → ←, teclas de función P F_i, ... permiten una definición de una vez por todas para un sistema concreto del tipo pantalla a nivel abstracto.

La idea básica es: una pantalla es un producto cartesiano de tipos (record) : $(T_1 x T_2 x \ldots x T_n)$
que tendrán sus selectores

$(S_1:T_1; S_2:T_2; \ldots ; S_n:T_n)$,
esto le dá a una pantalla una estructura arborescente:



ó

01 Pantalla
02 $S_1 : T_1$;
03 $S_{11} : T_{11}$;
⋮
03 $S_{1i_1} : T_{1i_1}$;
⋮
02 $S_n : T_n$;
03 $S_{n1} : T_{n1}$;
⋮
03 $S_{n\ in} : T_{n\ in}$

De forma análoga al "Screen" del COBOL

Nivel Representación

En él daremos la representación de los tipos componentes T_i , i= 1,n mediante tipos predefinidos, así como la geometría: posición, longitud, situación en el campo, ...

S_1: T_1(literal) integer, (X_1,Y_1), I6, left justified , required ...

S_2: T_2(literal) real, (X_2,Y_2), F7.2, right justified , required

S_n: T_n Complex ...

A las representaciones de los tipos T_i las llamaremos Campos: C_i

Nivel Dispositivo

Agruparemos en este nivel las características específicas del dispositivo que queremos utilizar: Mayúsculas/Minúsculas, juego de caracteres 1 ... n, Brillo/opacidad, Blinking/Fijo, color ...

Existiendo unas características standard por defecto.

Esta información se dará normalmente junto a la del nivel anterior en la especificación de un campo: $S_i:C_i:D_i$

3.- De los tipos de tratamiento al tipo pantalla: La Interface.

Una vez definido el tipo pantalla y sus variables, tenemos que realizar la correspondencia con las variables de los tipos de los tratamientos:

var V_1: T_1 --- S_1: C_1: D_1
⋮
V_n: T_n --- S_n: C_n: D_n

Esta correspondencia (análoga a la de parámetro formal-parámetro actual) tiene ciertas características propias: la variable es de E/S ... y además debemos poder expresar acciones y condiciones a cumplir al introducir ciertos campos.

De este conjunto de detalles se va a encargar la interface.

En una situación real el usuario dispondría de una biblioteca de pantallas predefinidas, además podrían definirse las específicas a su problema y usarlas mediante las interfaces que realizaría.

Veamos un esquema:

```
Programa      P ( ---);
Type  pantalla_i (T_1, ..., T_n)
        ⋮
End;

Procedure interface j (var p:pantalla_i);
Begin P.S_n = P.S_1 + 3;
      if P.S_n > 27 then P.S_j = P.S_k + P.S_1+X
      ⋮
End ;
```

```
Var  W: pantalla 1 (TA1,...,TAn);
     Y: pantalla 2 (TA1',...,TAm');
     :
     :
Begin .... Display interface 1 (W)....
      .... Display interface 2 (Y)....
           :
           :
End.
```

El no cumplimiento de alguna de las condiciones de la interface provocará un error y la reinicialización del proceso.

Las pantallas que definimos son pantallas lógicas que no tienen porqué coincidir con la pantalla física de que se disponga en una aplicación determinada. El "scrolling" aparece así como una necesidad de desplazar la pantalla física sobre la pantalla lógica usando operadores típicos de desplazamiento: adelante/atrás, izquierda/derecha, columnas ... realizadas con teclas de función según los casos.

También varias pantallas lógicas pueden aparecer sobre una pantalla física. Lo que será la situación corriente.

4.- SISTEMA ASOCIADO

En la realización de las ideas presentadas en este trabajo necesitaremos concretar:

- Lenguaje de definición de pantallas de acuerdo con las ideas presentadas.

- Forma de uso de las pantallas: desplazamiento sistemático del cursor de campo a campo, posibilidad de forzar su ubicación ...

- Sistema que nos permita gestionar una biblioteca de pantallas:

* Crear pantalla: dar control al lenguaje de creación.

* Probar pantalla: Desencadenar un test interactivo.

* Modificar pantalla: Dar control al lenguaje de creación/modificación.

* Almacenar pantalla (P): guardar la pantalla corriente con un nombre en la biblioteca de pantallas.

* Recuperar pantalla (P): convertir P en la pantalla corriente.

5.- CONCLUSIONES

- El "pantallazo" no es una representación de un tipo de tratamiento en el tipo pantalla puesto que no nos interesa la representación de los operadores del tipo tratamiento en operadores del tipo pantalla.

- Un pantallazo es la visualización de un objeto complejo tal que no nos interesamos en la forma de obtenerlo (término del tratamiento) sino una vez obtenido realizar una codificación visualizadora.

- La codificación de las constantes resultado de los tratamientos sigue ciertas reglas sintácticas que son las que nos interesan a efectos de su visualización ó adquisición.

- El sistema propuesto, aportando soluciones técnicas, situa a su vez el problema en el marco más amplio de los tipos abstractos algebraicos. Como producto software es equivalente en potencia a otros sistemas vistos, intentando a la vez situar el problema en el marco general presentado.

6.- BIBLIOGRAFIA

(1) ROWE L., A., SHOENS K., A., "Programming Language Constructs for Screen Definition"
IEEE Transactions on software engineering. Vol. SE-9, NO-1, Enero 1983

(2) GUTTAG J., HORNING J.J., "Formal Specification As a Design Tool"
1980 ACM, 7e POPL.

METODOLOGIA PARA LA ADQUISICION DE MATERIAL INFORMATICO

Isidro Ramos Salavert (Centro Informática U. Valencia)

José Marín Navarro (Diputacion Valencia y CIUV)

Centro Informática Universidad de Valencia, Dr. Moliner s/n Burjasot .

tlf. (96) 363.90.27

Marín, J, Ramos, I. Metodología para la adquisición de material informático. Actas de las Jornadas de Informática en la Administración Local. 1984.

I.- INTRODUCCION

La contratación de un sistema informático(*), ya sea por primera vez o como ampliación de alguno ya disponible es una tarea frecuente en las distintas Corporaciones.

La "voracidad" competitiva de las casas comerciales, la enorme variedad de constructores, equipos, modelos, programas,etc... , y la complejidad del propio mercado informático dificultan la elección del sistema más adecuado y hacen absolutamente necesario disponer de herramientas que ayuden a la selección, eliminando aspectos tan subjetivos, y en muchos casos habituales, como: el nombre de la marca comercial más conocida, el mismo equipo de la Corporación más próxima, el equipo más barato,.. .

En este artículo se describe una metodología, de fácil aplicación, que supone una ayuda importante en la selección de un sistema infórmatico. Esta metodología, basada en un método de análisis de decisión multi-objetivo, pretende aportar unas guías mínimas que eviten los aspectos subjetivos y den al proceso de selección un carácter más metódico y riguroso.

La organización del artículo es la siguiente: en el punto II, se hace una descripción detallada de la metodología, a continuación se analiza su aplicación al contexto de una Corporación, se estudia un caso práctico, terminando con las conclusiones y la bibliografía.

(*) Utilizamos el término sistema infórmatico en un sentido amplio para referirnos a elementos hardware, elementos software o a ambos simultáneamente.

II:EL PROCESO DE SELECCIÓN DE UN SISTEMA INFORMÁTICO: UNA METODOLOGÍA.

II.1- Introducción:

Un proceso de selección es un problema de decisión entre varias alternativas. En este caso lo podemos enunciar del siguiente modo: "Elegir el sistema informático más eficiente,donde eficiencia significa alcanzar los objetivos del sistema al menor coste posible" (**)

Este tipo de problemas son objeto de estudio en una parte de la Informática que es la "Ingenieria del Logicial " (Software Engineering), existiendo una serie de técnicas de medida de eficiencia (modelización, simulación, análisis estadístico, ...) que podrían utilizarse (1, 2, 3, 6) .

La metodología que desarrollamos, se basa en una técnica de análisis de la decisión multi-objetivo conocida con el nombre de "factor de mérito" (figure of merit).Esta técnica combina algunos criterios de eficacia en un criterio de decisión único.A tal criterio, se le da el nombre de factor de mérito y es una magnitud sin dimensiones que se asocia a una opción o alternativa.

En la bibliografía (1, 2, 3, 5) se pueden consultar métodos con una filosofía análoga al que describimos a continuación.

II.2-Una metodología de evaluación:

El método que proponemos comporta las siguientes fases:

1)Definición de un conjunto jerarquizado de criterios de selección.

Los criterios de selección a definir,función de los componentes y caracteristicas técnicas del sistema informático a evaluar,se organizan en una estructura arborescente:

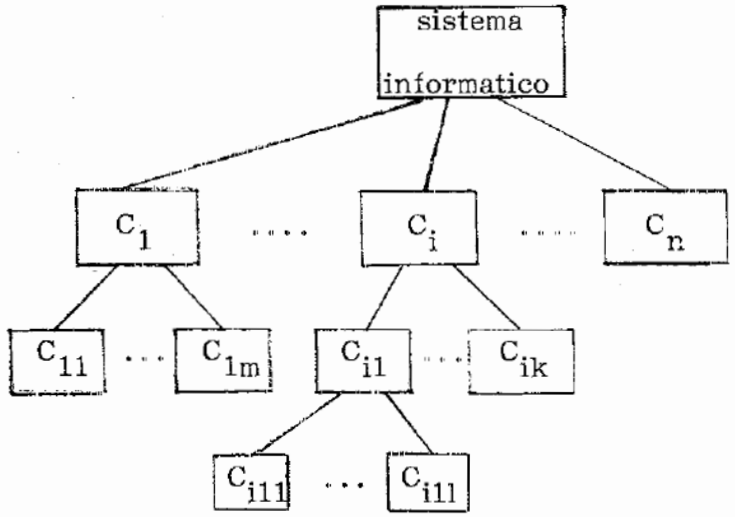


Fig. 1

El proceso de selección es función de los criterios C_i que definen el sistema informático.A su vez, y progresivamente, cada criterio puede descomponerse en una serie de subcriterios para obtener una evaluación más completa del mismo.Si un criterio admite descomposición, diremos que es compuesto, en caso contrario diremos que es simple.

La idea es obtener una descomposición del sistema en un conjunto de criterios donde, siempre que sea posible, cada criterio simple esté asociado con un dato o magnitud cuantificable.

A cada criterio de selección C_i se le asocia un factor de mérito F_i dependiente de :

- Una valoración V_i, que suponemos es un valor perteneciente al intervalo (o, MUL10), donde MUL10 es un valor numérico múltiplo de 10.
- Un peso P_i, que refleja la importancia relativa del criterio C_i en el proceso global de evaluación.Estos pesos P_i se definen en el intervalo (0,1) y verifican la propiedad de que la suma de los pesos asociados a los criterios resultantes de la descomposición de un criterio compuesto es 1.Es decir:

$$\sum_{i=1}^{n} P_i = 1 \ , \ \sum_{i=1}^{m} P_{i1} = 1 \ , \ \ldots .$$

2) Proceso de evaluación:

a) Definimos el factor de calidad F, asociado al criterio C, del modo siguiente:

caso 1 : El criterio C es simple,

F = V * P donde V es la valoración y P es el peso asignado al criterio C

caso 2 : El criterio C es compuesto,lo componen n subcriterios C_i

$$F = \sum_{i=1}^{n} F_i = \sum_{i=1}^{n} V_i * P_i$$

es decir,el factor de mérito F se define como la suma ponderada de las valoraciones de los subcriterios constituyentes del criterio original C .

Si suponemos que la definición del sistema informático comporta un conjunto de n criterios simples C_i.

$$F_{si} = \sum_{i=1}^{n} V_i * P_i$$

siendo F_{si} un factor de mérito asociado al sistema informático. Este factor nos indica la medida de lo que llamaremos la "calidad" o la "eficiencia" del sistema.

b) Cálculo de las valoraciones V_i de cada criterio C_i,

Se presentan dos casos:

caso 1 : Criterio asociado con un dato técnico cuantificable.

1) Buscar el máximo valor de este dato entre las alternativas en curso de evaluación, sea MAXDT

2) Calcular V_i,

$$V_i = (VDT * MUL10) / MAXDT$$

VDT es el valor del dato asociado al criterio C_i.

caso 2 : Criterio no asociado a un dato técnico cuantificable.

En función de las informaciones disponibles se da una valoración MUL10 a la alternativa que mejor responde a este criterio. Las restantes alternativas se definen en función de ésta, con un valor V_i menor o igual que MUL10.

En este cálculo de valoraciones hemos tomado como referencia la mejor de las alternativas en curso de evaluación. En ciertos casos o para ciertos criterios podría ser interesante tomar como referencia la alternativa o criterio ideal.

3) Proceso de selección

El proceso de evaluación anterior ha reducido el conjunto de criterios a un criterio simple llamado "calidad" o "eficiencia" y que es un valor comprendido entre 0 y MUL10. Aplicando el método a todas las alternativas se obtiene para cada una un factor calidad que permite establecer un orden entre ellas.

II.3 Aspectos económicos.

El factor económico es, sin duda, muy importante y debe intervenir en el proceso de selección. Ante la dificultad de dar criterios claros sobre la relación calidad/precio o costo/eficacia haremos algunas consideraciones sobre la misma.

El enunciado (**) plantea la búsqueda de la mejor relación costo/eficacia C(E). En general, los modelos coste/eficacia tienen la estructura siguiente:

- Un modelo de coste C=C(F), indica cuanto cuesta adquirir un conjunto de facilidades F.
- Un modelo de eficacia E=E(F), indica la eficacia que se obtiene con las facilidades F.
- Una relación costo/eficacia C(E) mediante las facilidades F

En estos términos debe plantearse el problema de la selección de un sistema informático.

El método anterior nos proporciona un modelo del tipo E=E(F), donde las facilidades F se introducen con los criterios de selección. Una primera solución para obtener la relación E(C) sería el considerar el precio como un criterio más y darle el peso correspondiente. No obstante, desaconsejamos esta solución, pues conceptualmente es importante disponer del factor "calidad" y el factor "precio" de cada alternativa.

Sin entrar en discusión de modelos económicos (funciones de producción, economía de escala, ..) utilizables en la selección de sistemas informáticos (ver (2) para más detalles), indicar que en el cálculo de la función costo debe incluirse, no solo el coste neto del sistema, sino también los costes marginales(implantación,valor añadido,...) .

II.4 Observaciones.

Sobre el método expuesto podemos hacer las siguientes observaciones:

- Es conceptualmente simple,de fácil aplicación, sistemático y automatizable.
- El enfoque de suma ponderada de criterios es una estructura clara y objetiva que permitirá,además de evaluar el sistema,analizar los pesos y valoraciones más influyentes y discutirlos.En este caso, el disponer de un programa que implemente el método es fundamental.
- Las conclusiones que se obtienen al aplicarlo son muy sensibles a los pesos y valoraciones asignados.
- Su aplicación es general a cualquier tipo de sistema.Habría que elaborar una descripción de los tipos de sistemas informáticos existentes (ordenadores grandes,minis,micros,personales,SGBD, paquetes de software,...) .

III.- APLICACION DE LA METODOLOGIA AL CONTEXTO DE UNA CORPORACIÓN:

La implantación de cualquier metodología en Informática requiere de una infraestructura organizativa que supervise su correcta aplicación.En este apartado,atendiendo a las caracteristicas específicas de las Corporaciones,describiremos una posible organización para la aplicación de la metodología desarrollada en el punto II.

Antes de detallar la secuencia de fases, que según nuestra experiencia es conveniente realizar,insistir en un hecho que,si bien es conocido por todos,no siempre es practicado.Previamente a cualquier tipo de contratación de un sistema informático,es básico y fundamental hacer el estudio previo de las necesidades y objetivos a cubrir.Este estudio previo puede tener la forma de un proyecto independiente o enmarcarse en el contexto más general de un plan informático de la Corporación.

La secuencia de fases puede ser la siguiente :

1) Constituir una comisión técnica de evaluación.

Esta comisión debe tener una composición mixta : política,administrativa y técnica.

Resaltar el rol a realizar por el representante político como garantía del buen funcionamiento de la comisión y de la aplicación correcta del método.

Un aspecto importante para el buen funcionamiento de esta comisión es el darle una estructura formal adecuada.

2) Definición de los criterios de selección.

La mayor parte del trabajo en esta fase la realizaran los informáticos, como conocedores del mercado informático y de las caracteristicas funcionales del sistema a evaluar.Los informáticos expondrán y razonaran a los restantes miembros de la comisión los criterios elegidos.De este modo, estos criterios serán filtrados y modificados por la visión "usuario".

3) Asignación de pesos a los criterios.

De acuerdo con los criterios de selección,en esta fase se recopilaran los datos técnicos necesarios para proceder a la evaluación posterior.Los soportes de trabajo son, básicamente, los manuales del constructor, no obstante, se recomienda consultar revistas especializadas(p. e., AUERBACH, COMPUTER REVIEW, DATAPRO, COMPUTER/WORLD, 01 INFORMATIQUE, ...) dónde se encontraran estudios puntuales y comparativos.

Esta fase,bastante costosa,puede simplificarse si se exige acompañar las ofertas de un cuestionario, cuyo contenido son los criterios de selección a utilizar.Igualmente,si la comisión lo estima oportuno,se harían entrevistas aisladas con cada uno de los ofertantes implicados.

5) Evaluación de las ofertas presentadas.

Como resultado de la fase anterior dispondremos de los datos técnicos de cada oferta.La evaluación se facilita con la construcción de un cuadro comparativo en forma de una tabla de doble entrada.

Aplicando el proceso de evaluación descrito en II.2 ,calculamos para cada oferta su factor "calidad técnica" y establecemos el orden técnico de las ofertas.

6) Analisis económico-técnico de las ofertas.

Sin duda la fase más dificil por la cantidad de factores impredecibles que pueden intervenir.Es importante evaluar, además del valor real de la contratación, los costes marginales que la implantación conlleva y durante un periodo de tiempo que puede ser, perfectamente, el ciclo de vida estimado para el sistema informático.

Los resultados que se obtienen de esta fase son:un orden estrictamente económico de las ofertas y, si es posible establecerlo, un orden técnico-económico.

7) Elaboración de un documento final.

Como conclusión del trabajo realizado por la comisión se elabora un documento técnico destinado al poder político para la toma de la decisión final.

Este documento podría constar de :

1) Las actas de las reuniones realizadas.
2) Informe técnico detallando el método usado,criterios empleados, pesos asignados y datos técnicos recopilados.
3) Un resumen con las conclusiones del análisis compuesto por : clasificacion técnica,clasificacion económica o técnico-económica y consideraciones.

IV.ESTUDIO DE UN CASO PRACTICO.

Se pretende contratar un ordenador de tipo medio y el logicial correspondiente, para consolidar el proceso de informatización en curso en una Corporación que dispone de un miniordenador(Máquina 1) y de un microordenador(Máquina 2).

Dado que, los criterios de selección dependen de la estructura del sistema informático a evaluar,en este caso serán función de las caracteristicas de un ordenador de tipo medio.La tabla I contiene la estructura jerárquica de los criterios elegidos.Esta tabla no pretende ser exhaustiva sino simplemente orientativa,evidentemente que algunos criterios podrían ampliarse,pero incluso en casos reales hay que llegar a un cierto compromiso sobre el nivel de detalle de la información a manipular.

La tabla I contiene también los pesos asignados a cada criterio.Esta asignación, igualmente discutible, puede modificarse en función de la importancia relativa que demos a los cinco criterios básicos:Material, Logicial, Mantenimiento, Compatibilidad y Ampliabilidad.

Con la información que suministren las casas ofertantes, construimos los cuadros de datos técnicos de cada oferta,los englobamos en un cuadro comparativo en forma de tabla de doble entrada criterios-ofertas y comenzamos el proceso de valoración.Este proceso lo haríamos según el algoritmo detallado en el punto II.2 b) dando puntuaciones en el intervalo 0 a 10.Se puede presentar en la forma de una tabla de doble entrada análoga a la de datos técnicos. Al final del proceso tendremos la "calidad" de cada oferta.

Al hacer el estudio económico de las ofertas tener en cuenta factores como: precio del material, alquiler del software, costo de instalación,coste de conversión de programas, precio del mantenimiento, consumo,.. .
La suma de estos factores la calcularíamos en un intervalo de tiempo de 5 años (ciclo de vida estimado para un ordenador), y obtendríamos el "precio" de cada oferta.

En este tipo de sistema informático,la relación calidad/precio puede hacerse con la ley de Grosch (3).Se expresa formalmente como

$$C = K * \sqrt{E}$$

donde, C es el coste,

E es la eficacia o calidad técnica,

K es una constante de proporcionalidad.

No obstante, conviene ser precavidos al aplicar esta ley, ya que si hace unos años era cualitativamente válida,actualmente, dada la variabilidad de las políticas comerciales y el desarrollo tecnológico,podría conducir a resultados incoherentes .

Criterios de seleccion de un ordenador

0.30 I. Material.

0.35 1. Unidad Central de proceso (CPU).

0.40 1. Procesador.

0.30 1. Biprocesador.

0.50 2. Velocidad/Potencia(en MIPS).

0.10 3. Conjunto de instrucciones.

0.10 4. Arquitectura.

0.30 2. Memoria Central.

0.40 1. Capacidad ofertada(bytes)

0.35 2. Tiempo de ciclo.

0.25 3. Memoria cache.

0.30 3. Unidades de entrada/salida.

0.60 1. Velocidad de transferencia.

0.40 2. Procesador de entrada/salida.

0.35 2. Periferia.

0.55 1. Discos.

0.30 1. Velocidad de transferencia.

0.40 2. Capacidad.

0.30 3. Tiempo de acceso.

0.20 2. Cintas.

0.50 1. Numero de unidades.

0.50 2. Velocidad de transferencia.

0.20 3. Impresoras.

0.50 1. Número de unidades.

0.50 2. Número de lineas minuto.

0.05 4. Otros dispositivos.

Tabla I

0.30 3. Comunicaciones

0.40 1. Procesador especializado.

0.30 2. Número de lineas ofertadas.

0.30 3. Número de terminales.

0.30 II. Logicial.

0.30 1. Sistema operativo.

0.10 2. Sistemas de gestión de ficheros.

0.20 3. Sistemas de gestion de base de datos.

0.10 4. Procesadores de lenguajes de programación.

0.15 5. Programas de aplicaciçon.

0.10 6. Burótica.

0.05 7. Logicial de red.

0.15 III. Mantenimiento

0.50 1. Material.

0.50 2. Logicial.

0.05 IV. Compatibilidad.

0.90 1. Maquina 1.

0.10 2. Máquina 2.

0.15 V. Ampliabilidad

0.35 1. Procesador Central.

0.25 2. Procesador de comunicaciones.

0.25 3. Memoria central.

0.10 4. Periferia.

0.05 5. Sistema gráfico.

0.05 VI. Otros

0.60 1. Formacion.

0.25 2. Red de usuarios.

0.15 3. Calidad de documentación.

Tabla I (cont.)

V. CONCLUSIONES.

En este artículo, hemos estudiado aspectos técnicos de la contratación de un sistema informático (ver en (5) los aspectos jurídicos), desarrollado una metodología general de selección y analizado su implantación organizativa en una Corporación.

La metodología desarrollada es un primer paso para objetivizar el problema de selección de un sistema informático y creemos que aporta un conjunto de principios mínimos que pueden ser de utilidad al decidir la contratación de material informático.

****

****

******

BIBLIOGRAFIA:

(1) BAUER F.,editor,1977,Software Engineering :An advanced course,Berlin,Springer Verlag

(2) BOEHM W.B:,1982,Software Engineering Economics,Prentice Hall.

(3) GOTLIEB C.C.,1977,Perfomance Measurement, en referencia (2).

(4) ORERO GIMENEZ A. M.,1982,Proceso de selección del sistema informático idóneo en una mediana empresa,Revista Proceso de Datos no. 119,pag 17-25.

(5) ROMERO HERNANDEZ F.,1982,Régimen jurídico de la contratación de equipos y sistemas informáticos en la Corporacion Local, JIAL 82,Valencia.

(6) RUSHINEK S. F.,RUSHINEK S.,1983,EDP auditior's Role in Evaluating Computerized Accounting Informations Systems by Queuing, Simulation and Statistical Models, en SICSE ACM,vol 15,no. 1,pag 207-210.

THE IMPACT OF NEW INFORMATION TECHNOLOGIES ON UNIVERSITY ORGANISATION: THE EXAMPLE OF CASTILLA-LA MANCHA

Isidro Ramos Salavert, Rector
University of Castilla-La Mancha

'The new technologies must not change the organisation of universities, since the organisational criteria need to be chosen with regard to the specific aspects of the function of universities; but the new technologies make it possible to overcome the limitations which exist in present-day university organisation, and which are a sign of obsolete technology and outdated practices'

GENERAL REMARKS

Computers and new information technologies in general have made the process of educational production much easier and they have also facilitated the dissemination of education to increasingly large and diversified segments of society. At the same time, they enable the universities to provide specialized services to specific sectors of the education market.

The computer has brought a new interactive dimension to the production and management tools of education, without, however, replacing them; thanks to the computer, these tools merely offer new possibilities in the educational environment.

The following are the fields of application of the new technologies: education in general, vocational training, research, the storage and retrieval of information, as well as another function which is not mentioned as often but which is no less important, namely the possibility of *sharing* the information between large groups of people (using networks)

The use of new technologies is therefore not only an economic fact but it is above all a new way of teaching and a powerful vehicle for increasingly wide-ranging aspects of education.

The greatest challenge facing the universities at the moment is that of producing high-quality software at a reasonable cost which would encourage

Ramos, I. The impact of new tecnologies on University organization: the example of Castilla-La Mancha. C.R.E. INFORMATION. Volumen: nº 70. 1985.

the many complex educational activities. There are several reasons for this:

- the increase of costs at universities (especially staff costs);
- cuts in university budgets;
- lower prices of both hardware and software;
- a more in-depth knowledge of software packages as well as their design and utilisation.

However, there are not only advantages in using the new technologies in the educational context: rejection phenomena manifest themselves in a more subtle way and they are usually related to the individual's loss of control.

Moreover, a package is never tailored exactly to the needs of the users, therefore the software either has to be modified (not always possible) or else it must be developed at the university itself, in which case a considerable number of staff and financial resources need to be involved. Once such software has been produced it very often turns out to suffer from a characteristic lack of flexibility; this is a definite disadvantage in an environment so prone to change as the university.

Perhaps there is a more exact explanation as to the low degree of computerisation seen in universities: the skills required for the production and utilisation of software are quite different from those normally expected of staff working in universities, and acceptance and retraining are usually fraught with problems.

In such an environment one needs to be both careful and modest when introducing the new technologies at the university; only a step-by-step approach will guarantee good results.

The use of new technologies brings with it major changes both as regards staffing and work stations, as well as in the management of administrative data, which brings additional difficulties.

However, there must be a global approach to computerisation as a tool for everyone. In the process of introducing computers there should not be any marginal non-computerised sectors since these would lead to pockets of resistance which could only be overcome with great difficulty.

At the same time, the university has to join regional and national networks. These will allow it to have access to information and computing resources available in other parts of the country or abroad.

Now for the concrete aspects of this paper: namely *the organisation of the university*. Here, the impact of the new technologies is great indeed.

New information technologies (NITs) will help lecturers to plan their coursework, manage the administrative data of the class, and they will also

be used at the highest level of the university where they will provide the information necessary for decision-making.

NITs allow the services to be channelled to those who need them, when and where they need them, thereby undermining the view that universities require a centralised structure.

'The fact that the university services are available in one single location to which all users must go is not so much due to real needs having to be satisfied but rather it is one of the characteristics of the traditional system and its limitations'

The new technologies must therefore be approached in a global perspective with each one contributing its different characteristics, and complementing those of the other technologies.

When we look at the increase in universities' administrative costs (usually exceeding the inflation rate) and at the falling cost of the new technologies we should realise that the latter are becoming less expensive every day. This means that, without taking into account their effectiveness, in many cases they will be the only real choice for the future.

For a long time the possibility of using computers in university management has been known. But it is only since the recent massive invasion of microcomputers that this application has been put into practice.

A tentative list of computer applications might appear as follows:

- *Administrative applications in the university*. NITs change the whole accounting system in a manner already well known. Other, more recent applications can be used in planning, cost simulation, and in drawing up the budget. NITs can also be used for cataloguing, checking on suppliers, products, and orders, as well as for any other type of stock management (automatic and centralised ordering leading to lower costs).
- *Assistance to students* (information and service): enrolment, diplomas, files.
- *Monitoring and drawing up of timetables*: different planning and simulation systems, including:
  * *Automatic listing of information on classrooms, lecturers, students, timetables*. The way in which curriculum changes affect the use of resources: classrooms, laboratories, lecturers, etc. The way in which budget cuts affect the staff hiring policies and the educational supply. The effects of salary increases on the budget.
- *Academic applications*
  * Databases with information on the different types of resources: lecturers, books, articles (stock control and ordering).
  * Use of the computer to help in the production of material for lecturers: examinations, course abstracts.

- *Applications which help in-service training and promotion of university staff*: training programmes for staff using computers for teaching, programming graphics, etc.
- *Information and documentation centre library*: financial checks, lending within the university and from one university to another (this helps avoid duplication and optimises the utilisation of resources), compiling statistics, research, abstracts, access to commercial databases: ERIC, etc.
- *Research support*
  * A computing device for physics, chemistry, etc.
  * Data analysis.
  * Use of packages for statistics and economics.
- *Vocational guidance for students*: preparing and evaluating test series, vocational guidance using personal characteristics and students' wishes, etc.
- *Classrooms and laboratories equipped with computers*:
  * Possibility of eliminating the constraints of space and time.
  * Real time simulation and presentation of models.
  * Electronic classroom, laboratory, and blackboard.
- *Educational management*: Automatic establishment of course groups, student groups, etc.
- *CAI, CAL*: once again this idea becomes attractive due to the falling costs and better overall knowledge.

In most cases computers have only been used in administration. The other aspects were not really taken into consideration.

COMPUTERISING THE SPANISH UNIVERSITY

Prehistory

Before 1967 small computers were distributed in a random fashion by a few universities.

Antiquity

In 1967 a mixed group of professionals established the Computer Centre of the Complutense University in Madrid. This was in fact Spain's first university computer centre (the small computers which were used in some universities filled the needs of scientists and technicians, but the aim was not the global computerisation of the universities concerned); the basic objective of this centre was that of encouraging Spanish universities at the national level to use computers. The 'spectacular' application of NITs in a creative

manner (art, architecture, education, linguistics) greatly helped in reaching this objective

Medieval period

During the first half of the '70s the universities were unable to purchase the necessary hardware, both through lack of funds and because general attitudes did not favour investment of resources in new information technologies However, the Spanish Education Ministry established a star-shaped network of remote job entry (RJE) terminals in the Spanish universities and it also gave the technical support which ensured, together with some autonomous minicomputers, the availability of information in the universities: there was a development of general-interest applications, including enrolment, accounts, library, and stocks, in addition to the computing and teaching service (courses designed to teach users the necessary computer languages).

One may say that NITs thus found their place in the university system.

Modern times

During the second half of the '70s the universities purchased medium-size computers (mega-minis); these contributed towards an independently centralised computerisation of universities

At the end of the '70s micro-computing proper appeared on the scene and this led to an abandoning of computing centres in the global context of the universities

Contemporary period

The first half of the '80s has been characterised by a telecommunications invasion and a proliferation of hardware. Micros, minis, mainframes, local and national networks; at the same time, universities are no longer insisting on centralised purchasing and training policies.

An important new factor is the increasing weight of commercial firms in the computing sector

The future

This is aptly described as follows:

'The university in a world of computers'

In the specific case of Castilla-La Mancha University we are at present introducing a homogeneous and distributed computer network which will provide services such as electronic mail, word processing, sharing of information, computing, research support, and assistance to the university administrative services

It is likely that the concept of 'classroom' as a location in time and space will be completely changed in the following manner:

- Increasing quantity and better quality of information transmitted; use of the computer as a vehicle for the information to be provided for the student: CAI, practical work, simulation exercises in real-time, coupled with traditional teaching methods using the blackboard, projectors, etc
- Disappearance of the 'four walls', as a physical learning environment, and disappearance of the 'timetable'; use of the television, telephone, videocassettes, etc; this will make it possible to provide teaching at a distance of several thousand kilometres (interaction between teacher and student is made possible through the telephone terminal).

The classroom, however, is not the only type of learning environment.

The possibilities of the library are also greatly enhanced thanks to the computer:

- Interactive access to bibliographic data.
- Monitoring of books lent and purchased.
- Administrative management.
- Abstracts and selective dissemination of information.

The book as a means of acquiring knowledge is complemented by other media which are faster, as well as interactive and flexible, such as terminals, videodiscs, etc.

The *laboratories*, and the whole research environment in general are the traditional places in which computers are used: as tools for calculation, monitoring and simulation of experiments, in addition to the routine tasks of storing and retrieving information related to research in general. Computers have been traditionally used for research in disciplines such as science and technology, but new horizons are now appearing in subjects such as history, literature, philology, etc which are also beginning to use computers.

Administrative and planning tasks in universities (and, generally speaking, all institutions of higher learning) are influenced by the use of computers: admission procedures, enrolment of students, files, etc. All these are being automated completely, and the data as a whole then serves as a basis for decision-making at the different planning and organisational levels.

The classroom, the library, the laboratory and the administration have already undergone changes owing to the presence of NITs. But groups of them in the shape of departments or faculties are still undergoing structural changes as far as information is concerned, and this is due to the availability of computer networks which are integrated into the total network of the university. One may also add other computerised environments such as stu-

dent halls of residence, all kinds of associations, consultation services, and services providing assistance to students.

The 'notice-board' is also becoming more versatile. Hospitals and general care of university staff have certain characteristics which cannot usually be taken into account with the traditional methods.

All these interlinked systems permit the following:

— electronic mail can be sent intra and inter campus;
— texts can be written and edited;
— an electronic timetable can be kept.

The student must of course purchase a personal computer which he can then link up to the university network at his place of study; this is a condition *sine qua non* for entering the world of information in which the student finds himself, and to which he, in turn, contributes information. If necessary, the student can have access to this information service via public terminals owned by the university.

Nowadays, many lecturers and research workers need to supply their computer code of access as additional information on their visiting cards. In this way, those wishing to contact them can reach them through the computer network from any point of the university's network, or even from other universities.

THE EVOLUTION OF NEW TECHNOLOGIES

'From the office calculator to the university computer'

The process of truly incorporating technology into the work at universities has not yet been completed on a global level. This introduction is happening only slowly, as it is accompanied by a strong feeling of guilt related to an imaginary 'original sin' in the shape of a turning away from the traditional university world (lectures, all-important role of the teacher, etc.). Despite all this, several factors have led to the incorporation of new technologies in the universities during the last twenty years. These factors include; the dictatorship role of the major firms in the field, the increasingly high cost of traditional resources, as well as the desire not to appear old-fashioned. One remaining question could be: what is the use of it all?

In this mechanism of assimilating technology the universities are following the innovations which have been created with various commercial aims in mind (including applications in the field of defense), and therefore one needs to analyse the trends expected from these new technologies, which can be summed up in the following way:

— New products appearing all the time, while existing ones are improved through the incorporation of micro-electronics.

- New discoveries in the field of technology as well as products resulting from them will be distributed, as hitherto, by existing companies; in some cases these companies tend to specialise in this sort of market.
- Users in the sector of education are not numerous enough to warrant the development of products specifically aimed at their needs; therefore, products developed with a more general application in mind will have to be adapted
- The new products are increasingly user friendly and usually incorporate a 'training' stage

In the world of new technologies communications pay a major role. This is due to several reasons:

* Falling costs
* Availability of different systems specifically designed for each type of user
* Ease of use and flexibility of communication networks.
* Possibility of transmitting bulk data
* Possibility of link-up to a world network, thereby permitting a universal sharing of knowledge

Amongst the new technologies, especially from the point of view of communication, we also need to consider bi-directional cable television, the use of satellites, digital telephones, local networks, local television channels, etc Each of these types of technology has its own specific characteristics: for example, combining cable and satellite transmission can be useful for certain groups; the digital telephone allows links to be established between audio networks and computers, and this will enable not only the transmission of data, but also the storage and elaboration thereof (package switching networks, etc.)

As for computers, an even greater reduction in cost is expected, for the same computing power. It will become easier to use heterogeneous networks, since protocols will be increasingly standardised; software development will also progress due to the large number of potential buyers. Microcomputers will continue to expand, and will become the standard interface to be used within the education system, for students, lecturers, etc Desk-top computers will also be used as terminals (with telephone links) at any remote location

This whole technological evolution will, of course, play an important part in the future organisation of the universities The evolution of interaction between user and computer is reaching a new stage now: not only in the cybernetics of machine behaviour coming ever closer to human behaviour (speech recognition and synthesis, natural language), but also through the development of 'friendlier' programming environments (communication by

graphics, menus, windows, backed by high resolution screens, interaction by means of the mouse, etc)

ASPECTS FOR CONSIDERATION

Under the first three headings of this paper we did not consider the implications of the ideas expressed for university organisation.

Far be it for us to provide ready-made conclusions and ideas to be applied

The new University of Castilla-La Mancha is divided amongst several campuses; it saw the light of day under less than favourable economic circumstances; and we have very highly specialised experts among our staff (the technological level of the region is rather low). Because of these problems we are introducing a homogeneous distributed network of mega-mini-micro computers, and are now in the introductory phase, God willing !

However, before we go on to consider the practical problems at Castilla-La Mancha, let us take a closer look at the university sectors which will be affected by the generalised introduction of computers.

Academic sector

Here it is the administrative aspects with regard to the instruction given in the different centres, and the general services which will be concerned.

Staffing Sector

Different aspects need to be considered under this heading: data relating to staff, salaries and benefits, etc.

Financial management sector

This concerns all the accounting jobs relating to the budget, auditing and management of financial resources in general.

In particular, computer support can help in drawing up the budget and in constructing financial models.

Library, periodicals, documentation sector

This concerne management of the lending library, purchases, access, retrieval and selective dissemination of information

The University of Castilla-La Mancha is the result of a grouping together of ten existing Centres in the region (80,000 km², 1,600,000 inhabitants) The population distribution is as follows :

The distances between the various provincial capitals where the centres are located are as follows:

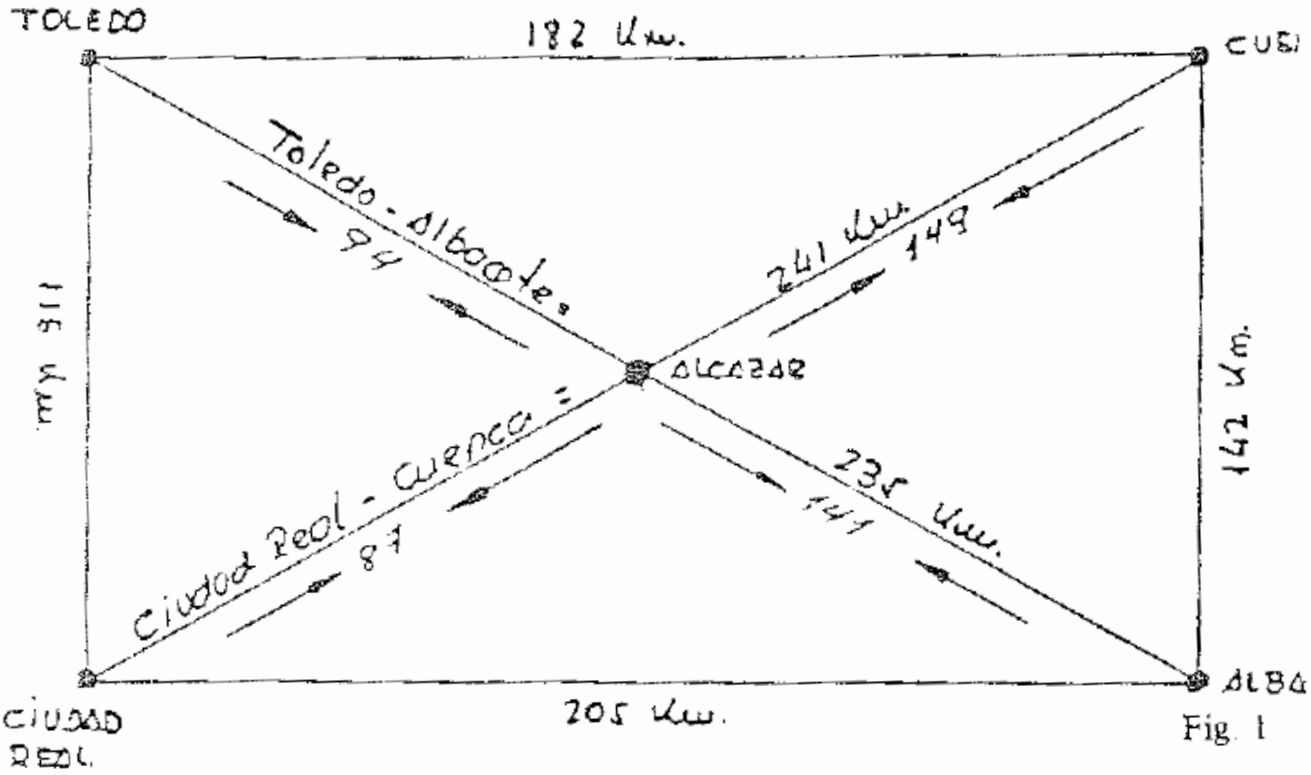


Fig. 1



Fig 2

Map of regional centres of equilibrium

500

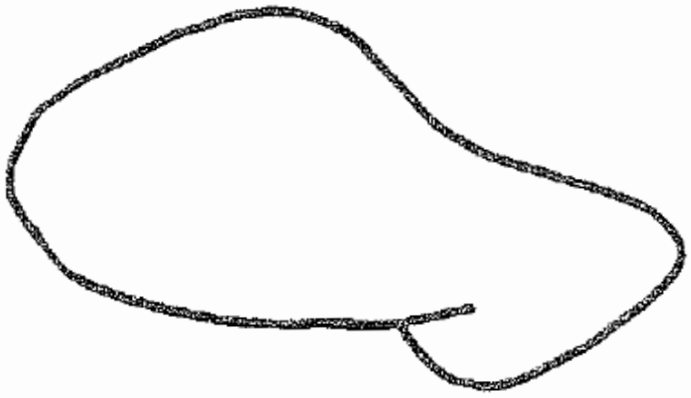
The already existing and the newly founded centres of the region are distributed amongst the following cities: Toledo, Ciudad Real, Cuenca, and Albacete.

It is the regional government which will propose the setting up of new centres, but the central government will have to give its final approval.

Once the problem had been defined one had to take into consideration the fact that the regions in Spain are in the process of becoming autonomous (with a transfer of certain powers to regional governments in the medium-term). This, and the political decision to found the university shows clearly that a traditional type of university would not have been a possible solution.

The university was, from its birth, geographically divided in order to play the role of a higher edication service for the whole region.

It is obvious that the organisation problems of Castilla-La Mancha University are not easy to solve, and challenge the traditional solutions. We tend to think that solutions must be found as a result of the problems, and not the other way round In our case, for virtually every aspect of organisation it was necessary to provide a homogeneous and yet scattered computer network for support.



A technological question which was a source of worry was the choice between two options: *centralisation* or *decentralisation* of computer resources, and, if the latter course is chosen, does the appropriate choice involve integration into networks? We think that a mixed solution is best: we need to have clusters of micro-computers linked by local networks, but, in addition, there must also be links to powerful computing centres This configuration has an advantage over the traditional system of time-sharing since the micros do not lose any of their power or speed when there are a lot of users (unlike the situation in time-share set-ups).

The general communication network must also be able to transmit other information: video, speech, digital signals, etc. and will thus allow each individual member of the university community to link up

This, clearly, is the future direction in which to go. The University of Castilla-La Mancha intends to follow in the footsteps of other universities, especially those in the U.S.A. Brown University has already linked up 130 buildings and, in 1989 will have 10,000 workstations included in its network.

Major investments are nevertheless necessary:

* MIT: 50 million dollars in three years.
* University of Valencia: 100 million pesetas
* University of Castilla-La Mancha: 35 million pesetas

One of the main problems in all these purchases is the technological *obsolescence*, which leads us to question all these investments. To overcome this, a global policy is needed which includes the following elements:

- Agreements with firms which might grant discounts on large-scale purchases. Joint development of software. Encouragement of firms to make gifts since the universities are about to train future 'brand-dependent' users.
- Increasing university fees for students, or else making it a requirement for all students to bring their computer with them to university (they will naturally take it away with them at the end of their course of study).
- University budget..

In practical terms, the network will ensure an equal sharing in all information for the different groups in the university. It will also provide storage for all those documents which do not necessarily exist in hard copy. The network will also provide the following traditional services:

- Word processing
- Electronic mail
- Timetabling
- Databanks, etc..

The network will moreover allow access to computer resources without regard to the point of access (virtual terminal, distributed process, etc.), as well as scientific computing facilities and management tools

There are major implications for university organisation (at the level of structures and processes), for the following reason: 'the point of access loses its importance', since it is no longer necessary to ensure continuity in time and proximity in space in order to share information. This holds equally true for the generation, storage, and dissemination of information.

For example, the library will consist of separately held data stores, but it

will be centrally managed through the network, and access to it will be from any point along the network.

ASPECTS TO BE CONSIDERED BY THE RECTORS

When introducing technology into the universities there is always the risk of producing a *technocracy*, i e. a danger that technicians will have a major say in matters they are not really competent to judge. The groups concerned must be the ones taking the decisions, and, in order to do this, training must be given to those groups: lecturers, students, administrative staff, etc At the same time, the university must facilitate the use and/or the purchase of computers. This aim can be achieved through an agreement with computer firms granting reasonable prices for large quantity orders, financing through the university budget, sharing of royalties with teachers who write 'course-ware', etc

Another aspect which needs some thought is the *compatibility* of the various computer systems. Homogeneity brings with it dependence upon a single firm, with all its attendant problems; this is especially true in a university environment where the presence of the different hardware firms is a desirable situation (different languages, and ideas) Heterogeneity, on the other hand, always includes the danger of incompatibility.

The only solution to be envisaged is that of controlled heterogeneity. In other words, the university offers a support, teaching and counselling service for 2 or 3 manufactured brands (ones which are considered most suitable at any given moment).

Whatever the choice, the cost of introducing computerisation remains high The universities must therefore decide either to let this process pass them by, or, if they decide to join in, to bear all the consequences deriving from it.

Another choice which needs to be made is between *micros and mainframes*. We think that mainframes will still be used in the future, but they will mainly be involved in large-scale scientific programmes which need their particular possibilities; most university users, however, will be happy to use micros, and the purchase of a mainframe will depend mainly on whether circumstances warrant this

The *national or international character of the communication networks* is mainly dependent on the disciplines involved, for example, the computing networks of CERN are oriented towards high-energy physics, etc The different networks will be established as a result of characteristics and criteria which are not of a technological nature (multinational research teams, European research policy, etc.).

It should be noted the universities' lack of funds makes it less than certain that these institutions will be completely equipped with the new technologies, since these are costly to buy and especially to maintain.

There is also the phenomenon of *rejection*, as a result of the computerisation of universities, a kind of elitist view that *the quality of thought is more important than the quantity of information*', as well as the worry over possible job losses (many traditional information channels are disappearing and with them the respective jobs) All this is hindering the integration process, whose leadership (owing to research) has long since left the universities and been taken up by the large computer companies

Therefore, it is necessary to establish *priorities* allowing one to 'manage poverty'; every time this happens the administrative services are top of the list, with everything else having to take second place

We do not envisage the possibility of *limiting investments* in the new technologies; personal and individual access points, computerised classrooms, etc. require a maximum achievement level.

In any case, computerisation has to begin within the groups which want to use it; it must not be initiated by the computer specialists In other words, the real problem does not lie in the science of computing as such, but in its integration into other disciplines.

The whole process will therefore be slow, and its success uncertain

It is absolutely necessary to maintain good relationships with the computer manufacturers. It is essential to make full use of their desire to open new market sectors for their products and to find new users, when one is considering the purchase of new technologies. Projects must be approached jointly, with the university providing the fundamental knowledge of the problem, and the firm its funds.



ISIDRO RAMOS

PUBLICACIONES 1968-2013

VOLUMEN 2:
MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS

VALENCIA, SEPTIEMBRE DE 2014

ÍNDICE GENERAL

VOLUMEN 1: MADRID, BILBAO, VALENCIA

Prólogo i

Días de hemeroteca v

… y noches de ordenador ix

Estructura del libro xi

El saber y el saber hacer en informática: una perspectiva histórica xiii

Índice general xxi

PARTE I: MADRID (1968-1974)

Prólogo de Roberto Moya Quiles 3

Isidro Ramos y el Centro de Cálculo de la Universidad de Madrid, por Ernesto García Camarero ex-director del CCUM 5

Seminarios del Centro De Cálculo De la Universidad Complutense de Madrid. 1968. 11

Gil Gayarre, M., Ramos, I., et al., Posibilidades de la Automatización en la corrección de exámenes. Revista de Automática. Año III, nº 7. 1970. 109

Fernández Flores, I. Ramos I., Traductor 70/13. Boletín CCUM. Volumen 10. 1970. 123

Moya R., Ramos, I. Aplicación de Hash-Coding al tratamiento automático de Gramáticas". Boletín CCUM. nº 17. 1971 137

Fernández Flórez, I., García Camarero, E., Ramos, I. A Basic Language Oriented to secondary School. Information Processing 71. 1972. 147

García Camarero, E., Ramos, I. Ensayo sobre la definición automática de tratamientos clínicos. Boletín CCUM. 1971. 151

Ramos, I. Lenguajes para la escritura de traductores. Revista de Automática. Enero-Marzo 1973. 163

Alfonseca, M., Ramos I. Un analizador sintáctico para un subconjunto de PL/I. Actas del Congreso Nacional de Automática. 1972. 169

Ramos, I. Un ejemplo de recursividad: HANOI. Boletín CCUM. nº 24. 1974. 205

Ramos, I., Lozano, C. Muñoz, A. Sistema de Creación y Mantenimiento de Bancos de Información. Revista de Automática, 1974 (Parte I), 1975 (Partes II y III). 211

PARTE II: BILBAO (1975-1983)

Prólogo de José Ramón Gariagoitia 255

Ramos, I. Système pour la création et gestion des banques de données. Seminaires Institut de Programmation (Université de Paris VI). 1974. 257

Moya R., Ramos I. Metaprogramación Estructurada. 3er Congreso Nacional de Informática y Automática. 1975. 263

Ramos, I., Moya, R., Martín, P. Evaluador Sintáctico del Lenguaje de Control de Comandos del Sistema Operativo EXEC-8. 3er Congreso Nacional de Informática y Automática. 1975. 289

Ramos, I. Aramberri, J. Cross-Macroassembler for Micro MCS-80 in a HP 2116-B Minicomputer. Euromicro. 1976. 301

Ramos, I. Informátique de Base: These, Antithèse, Synthèse. Convention Informatique. 1976. 309

Garitagoitia, J.R., Ramos, I. Relocating Direct Linking Loader for MCS-8 Microcomputer Series. Proc. of COMPCON 77. 1977. 313

Ramos, I. Sistema Editor / Perforador de textos. Revista de Automática e Informática. Volumen: nº 31. 1977. 319

Moya, R. Ramos, I. LPE: Un lenguaje en español para la programación estructurada. Proceso de Datos, no. 70, 1977. 325

Moya, R., Ramos I. SPL (Structured Programming Language). A language for structured programming. Proc. Of the 5th Annual III Conference. 1977. 331

Ramos, I. Aproximación prospectiva al software de base. Actas de la Convención Informática Latina. 1977. 339

Ramos, I., Aramberri, J. y Mellado, M. Modelo de realización de primitivas de sincronización para sistemas multimicroprocesador. Actas del 75 Aniversario Real Sociedad Española de Física y Química. 1978. 345

Aramberri, J., Ramos, I., Garitagoitia, J. R. Diseño y desarrollo de un sistema microprocesador. Actas I Jornadas sobre microprocesadores. 1978. 347

Moya, R., Ramos, I. SP Gramars (Context Free Gramars for Structured Programming). ACM SIGPLAN. Vol.4 Nº1. 1979. 357

Aramberri, J, Ramos, I., Mellado, M., Garitagoitia, J. R. Modelo de Sistema Microprocesador. Actas del 4º Congreso de Informática y Automática. 1979. 365

Sopeña, L., Ramos, I. Sistema de control en tiempo real de una red viaria. Actas del 4º Congreso de Informática y Automática. 1979. 375

Aguirre, J., Ramos, I. Sistema de programación estructurada con vuelta atrás. Actas del 4º Congreso de Informática y Automática, Madrid, 1979. 385

Ramos, I. Garitagoitia, J.R., Aramberri, J. Algunas soluciones al problema de interbloqueo entre procesos. 4º Congreso de Informática y Automática. 1979. 391

Guaresti, J.M., Garitagoitia, J.R. y Ramos I.A software chronological state controller. Actas del 4º Congreso de Informática y Automática. 1979. 403

Ramos, I. Calculadora de mesa. Un proyecto de clase sobre teoría de Lenguajes y Autómatas. Revista de Informática y Automática. 1979. 415

Zuloaga, J.L., Santamaria, R., Emaldi, R., Ramos, I. Sistema de Control lógico programable sobre microordenador. Revista de Informática y Automática. 1979. 421

Garitagoitia, J.R., Martinez,C., Ramos, I. MIKBUS: un bus para placas de microcomputadores cooperantes. Actas del II Simposio Español de Informática Repartida. 1982. 429

Garitagoitia, J.R., Ramos, I. Algoritmo de prevención para un sistema de gestión de ficheros. Actas de la Convención Informática Latina. 1983. 439

PARTE III: UNIVERSITAT DE VALÈNCIA - UNIVERSIDAD DE CASTILLA-LA MANCHA (1978-1987)

Prólogo de Francisco José Quiles Flor 455

Ramos, I. Semántica para SPL. Questió. Vol. 4. nº 1.1980. 459

Ramos, I. Un lenguaje de Programación con tipos Abstractos interpretado en PROLOG. Otra forma de compilar. Revista de Informática y Automática. 1984. 467

Ramos, I. Especificación de Pantallas. Revista de Informática y Automática, 1984. 471

Marín, J, Ramos, I. Metodología para la adquisición de material informático. Actas de las Jornadas de Informática en la Administración Local. 1984. 475

Ramos, I. The impact of new tecnologies on University organization: the example of Castilla-La Mancha. C.R.E. INFORMATION. Volumen: nº 70. 1985. 491

VOLUMEN 2: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

PARTE IV: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

Prólogo de María Alpuente 527

Celma, M., Moreno, L., Pastor, O., Ramos, I. Implementación de un sistema de ges-

tión de bases de datos deductivas. Revista de Informática y Automática, Vol. XX, Nº 3. 1987. 529

Alpuente, M., Botti, V., Galiano, I., Sanchis, E., Vila, F., Ramos, I. Entorno de programación LALI implementado en PROLOG. Revista de Informática y Automática . Vol. XX, Nº 4. 1987. 537

González del Río, A., Alpuente, M., Casamayor, J.C., Pastor, M.A., Ramírez, M.J., Ramos, I. Protolog : A conceptual Schema Facility for Automated Prototype Generation. 7th. IASTED International Symposium. 1989. 543

Ramos I., Canós, J.H., Forradellas, R., Oliver, J. A conceptual schema specification system for rapid prototyping. 8th IASTED International Symposium. Austria. Feb. 1990. 547

Ramos, I. Logics and OO-Data Bases: a declarative approach. Proc. International Conference on Database and Expert Systems Applications (DEXA 90), 1990. 551

Ramos, I.; Pastor, O.; Casado, V. OO and Active Formal Information System Specification. Proc. International Conference on Database and Expert System Applications (DEXA 91) Vol. II. 1991. 557

Pastor, O. Ramos, I. Canós, J.H. Object Oriented and Relational Specification of Information Systems. Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS. 1991. 563

Ramos, I., Cuevas, J., Devesa, J., Pastor, O., Canós, J.H. On the use of Algebras as Semantic domains of Object Societies. Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS. 1992. 579

Alpuente, M., Ramírez, M. J. An Equational Constraint Logic approach to Conceptual Modelling. In Ramos, I. and Tjoa, A. (eds), Proc. of the International Conference on Database and Expert Systems Applications (DEXA 92), 1992. 589

Ramos, I., Pastor, O., Cuevas J., Devesa , J. Objects as Observable Processes. Proc. of the 4th International Workshop of the Deductive Approach for DB and IS.1993. 595

Pastor, O., Ramos, I., Canós, J.H. From Analysis to Design in an OO and Deductive Environment. Proc of the International Conference on Database and Expert System Applications (DEXA´93). 1993. 611

Toval, A., Ramos, I. , Pastor, O. Prototyping Object Oriented Specifications in an Algebraic Environment. Proc. Of the International Conference on Database and Expert System Applications (DEXA'94), 1994. 617

Devesa, J., Cuevas, J., Ramos, I. Implementation of a Term Rewriting System for solving process queries in an object-oriented environment". Proc. Of the GULP-PRODE 94. 1994. 627

Canós, J. H., Penadés, M.C., Ramos, I., Pastor, O. A Knowledge-Base Architecture for Object Societies. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 643

Pastor, O., Ramos, I., Canós J.H. OASIS v2: A Class Definition Language. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 649

Ramos, I., Toval, A., García-Molina, J., Flores, M., González, M., Fernández, J. L., Nicolás, J., Pastor, O., Canós, J. H., Pelechano, V., Penadés, M.C., Bonet, B. Entorno General de Produccion automática de prototipos de software orientado a objetos: OOAP (Object-Oriented Assistant Prototyper). Actas de las I Jornadas de Informática. Asociación Española de Informática y Automática (AEIA). 1995. 659

Pastor O., Pelechano V., Bonet B., Ramos I. An object Oriented Approach for making Automated Prototyping Feasible. Proc. of the International Conference on Database and Expert System Applications (DEXA 96). 1996. 673

Carsí, J.A., Ramos, I., Penadés, M.C., Pelechano, V., Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic. Actas de las I Jornadas de Trabajo en Ingeniería del Software, 1996. 683

Penadés, M.C., Ramos, I., Canós, J.H., Tolsá, J., Ferrer, J. Análisis, diseño e implementación de un Entorno de Prototipación Automática. Actas de las I Jornadas de Trabajo en Ingeniería del Software,1996. 695

Sánchez, P. Letelier, P., Ramos, I. Representación de Objetos OASIS como Teorias y Traducción a un Modelo de Concurrencia. Computación y Sistemas, Vol. I, Nº. 2. 1997. 707

Lozano, M.D., Ramos, I. Evolución y perspectivas en el desarrollo de software: nuevas tendencias orientadas a objetos. Ensayos: revista de la Escuela Universitaria de Magisterio de Albacete, Nº. 12. 1997. 719

Sánchez, P., Letelier, P., Ramos, I. Constructs for Prototyping Information Systems using Object Petri Nets. Proc. of the IEEE International Conference on System, Man and Cybernetics.1997. 733

Lozano, M.D., Ramos, I., Cuenca, P. Application of an Object-Oriented Methodology for Automatic Software Production : OO-Method. Proc. Of the IEEE PACRIM 97. 1997. 739

Letelier, P., Sánchez, P., Ramos, I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitios. Proc. of the Argentine Symposium on Object Orientation. 1997. 743

Letelier, P. Sánchez, P., Ramos. I. Animation of Systems Specifications Using Concurrent Logic Programming. Proc. of the Symposium in Logical Approaches to Agent and Modelling and Design (ESSLLI´97). 1997. 751

Carsí, J.A., Canós, J.H., Ramos, I. Un intérprete de Transaction Frame Logic. Actas de las II Jornadas de Trabajo en Ingeniería del Software. 1997. 759

Tolsá, J.M., Ferrer, J., Penadés, M.C., Canós, J.H., Ramos, I. Meta-KAOS. Un metaentorno de prototipación automática. Actas de las II Jornadas de Trabajo en

Ingeniería del Software. 1997. 773

Ramos, I.,Canós, J.H., Pastor,O.+ investigadores proyecto MENHIR. MENHIR: Modelos, Entornos y Nuevas Herramientas para la Ingeniería de Requisitos. Actas de las III Jornadas sobre Tecnología de Objetos. 1997. 787

Canós, J.H., Lozano, M.D., Ramos, I. KAOS: An Object-Oriented Software Tool for the Objects Definition, Updating, Querying and Programming in an Object-Oriented Environment. Proc. of the IEEE Canadian Conference on Electrical & Computer Engineering.1998. 799

Letelier, P., Sánchez, P., Ramos, I. Especificaciones de proceso para objetos y su representación en Lógica Dinámica. Actas de las III Jornadas de Ingeniería de Software. 1998. 803

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto. Actas de las III Jornadas de Ingeniería del Software. 1998. 813

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a las pautas de representación de OASIS en un formalismo gráfico. Actas de las III Jornadas de Ingeniería del Software. 1998. 827

Carsí, J. A., Camilleri, S., Canós, J.H., Ramos, I. Propuesta de Interfaz Gráfica de Usuario homogénea para el diseño y la explotación de Sistemas de Información. Actas de las III Jornadas de Ingeniería del Software. 1998. 839

Letelier, P., Sánchez, P., Ramos, I. Un ambiente para especificación incremental y validación de modelos conceptuales. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99).1999. 853

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99). 1999. 867

Letelier, P., Sánchez, P., Ramos, I. Prototyping a requirements specification through an automatically generated concurrent logic program. Proc. of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99). 1999. 879

Pastor,O., Canós,J.H., Ramos,I. From CASE to CARE (Computer-Aided Requirements Engineering). Proc. of the International Conference on Conceptual Modeling (ER'99). 1999. 895

Lozano, M., Ramos, I. Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. Workshop on Integrating Human Factors into Use Cases and OO Methods at ECOOP'99. 1999. 911

Letelier, P., Sánchez, P., Ramos, I. Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales". Actas del VI Congreso Internacional de Investigación en Ciencias Computacionales (CIICC'99). 1999. 917

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. Modelado conceptual con un lenguaje

formal y orientado a objeto. Actas de la XXV Conferencia Latinoamericana de Informática (CLEI'99). 1999. 929

Letelier, P. Sánchez, P., Ramos, I. Animation of Conceptual Models using two Concurrent Environments: An overview. Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers. 1999. 943

Parets, J., Carsí, J., Canós, J.H., Anaya, A., Hurtado, M.V., Penadés, M.C., Paderewski, P., Ramos, I., Rodriguez, M.J., La evolución de modelos en el desarrollo de software: los enfoques MEDES y OASIS. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos. 1999. 951

Sánchez, P., Letelier, P., Ramos, I. Una Interpretación de la Herencia de OASIS en Lógica Dinámica. Computación y Sistemas, vol. 4, núm.4. 2000. 963

Sánchez, P., Letelier, P., Ramos, I. Animating formal specifications with Inheritance in a DL-based framework. Requirements Engineering Journal, vol.4, no.4. 2000. 973

Anaya, R., Ramos, I. Un lenguaje formal para especificación y refinamiento de Objetos de Negocio. Ingeniería del software y bases de datos: tendencias actuales. Universidad de Castilla-La Mancha, 2000. 995

Anaya, R., Ramos, I. AR2CA: Una Herramienta para la Construcción de componentes reutilizables a través de niveles de refinamiento. Actas de 3as Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software (IDEAS2000). 2000. 1011

Carsí, J. A., Ramos, I., Canós, J.H. Reflection in the OASIS object model to solve Schema Evolution Problems. Proc. of the International Symposium on the Principles of Software Evolution (ISPSE2000). 2000. 1025

Sánchez, P., Letelier, P., Ramos, I. Validation of Conceptual Models by Animation in an Scenario based approach. Workshop 'Scenario-based round-trip engineering' at OOPSLA 2000. 2000. 1031

Giret, A., Ramos, I.Full software agents as OASIS 3.0 objects. Proc. of the 2nd Constraint) Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2001. 2001. 1037

Letelier, P., Ramos, I. Trazabilidad de Requisitos y Calidad del Software: Una Aproximación basada en UML. En Piattini y García (Eds.), Calidad en el Desarrollo y Mantenimiento del Software. Ra-MA, 2002. 1053

Pérez, J., Carsí, J. A., Ramos, I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR) at ICSM'02. 2002. 1075

VOLUMEN 3: ARQUITECTURAS SOFTWARE (2002-2010)

PARTE V: ARQUITECTURAS SOFTWARE (2002-2010)

Prólogo de Patricio Letelier 1109

Ferrer, J., Lorenzo, M. A., Ramos, I.,Carsí, J. A., Pérez, J. Modeling Dynamic Aspects in Architectures and Multiagent Systems. Proc. of the Constraint Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2002. 2002. 1111

Pérez, J., Ramos, I., Lorenzo, M. A., Letelier, P., Jaén, J. PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos. Actas de las VII Jornadas de Ingeniería del Software y Bases de Datos. 2002. 1125

Ferrer, J., Lorenzo, M. A., Ramos, I., Carsí, J. A. Dinamismo en arquitecturas y sistemas multiagente. Actas de las V Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2002). 2002. 1137

Jaén, J., Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures. Proc. of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03). 2003. 1147

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. Proc. of the 3rd IEEE International Conference on Quality Software (QSIC 2003). 2003. 1153

Pérez, J., Ali, N., Ramos, I., Carsí, J. A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes. Workshop Desarrollo de Software Orientado a Aspectos (DSOA). Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1161

Pérez, J., Ali, N., Ramos, I., Pastor, J. A., Sánchez, P., Álvarez, B. Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1167

Lorenzo, M. A., Peñarrubia, A., Carsí, J.A., Ramos, I. Modelando con UML Arquitecturas Software Reflexivas desde la Aproximación Orientada a Aspectos: Un Caso de Estudio. Actas de las VI Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2003). 2003. 1177

Navarro, E., Ramos, I. Requirements and Architecture: a marriage for Quality Assurance. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1189

Silva, J., Ali, N., Carsí, J. A.,Ramos, I. El aspecto de distribución de PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1199

Ali, N., Silva, J., Jaén, J., Ramos, I., Carsí, J. A., Pérez, J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proc. of the 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2003). 2003. 1209

Pérez J., Ali, N. Carsí, J. A., Ramos, I. Navarro, E. Designing Software Architectures with an Aspect-Oriented Language. ADVICE: Journal of Aspect Orientation, Vol. 1, No. 1, 2004. 1217

Navarro, E., Letelier, P., Ramos, I. Goals and Quality Characteristics: Separating

Concerns. Proc. of Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004. 2004. 1239

Navarro, E., Letelier, P., Ramos, I. UML Visualization for an aspect and goal oriented approach. Proc. of the 5th Aspect-Oriented Modeling Workshop (AOM'04) at UML 2004 Conference. 2004. 1247

Navarro, E., Ramos, I., Pérez, J. Goals Model Driving Software Architecture. 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004). 2004. 1255

Ali, N., Pérez, J., Ramos, I. High Level Specification of Distributed and Mobile Information Systems. Proc. of the Second International Symposium on Innovation in Information & Communication Technology (ISSICT2004). 2004. 1263

Ali, N., Carsí, J. A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2004. 1277

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004). 2004. 1285

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Orientación a aspectos y Orientación a objetivos: una propuesta para su integración. Taller de Desarrollo de Software Orientado a Aspectos, junto a JISBD'2004. 2004. 1293

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Dynamic Evolution in Aspect-Oriented Architectural Models. European Workshop on Software Architecture. 2005. 1299

Pérez, J., Ali, N., Costa, C., Carsí, J. A., Ramos, I. Executing Aspect-Oriented Component-Based Software Architectures on . NET Technology. Proc. of the 3rd International Conference on .NET Technologies. 2005. 1317

Ali, N., Ramos, I., Carsí, J. A. A Conceptual Model for Distributed Aspect-Oriented Software Architectures. Proc. of the International Conference on Information Technology Coding and Computing, ITCC 2005. 2005. 1329

Limón, R., Torres, J., Ramos, I. Separación de "Crosscutting Concerns" en el Desarrollo Orientado a Aspectos a Nivel de Diseño Arquitectónico. Actas del 6º Simposium Iberoamericano de Computación e Informática (SICI'05). 2005. 1335

Navarro, E., Letelier, P., Ramos, I. Integrating Expressiveness of Modern Requirements Modeling Approaches. Proc. of the 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005). 2005. 1345

Costa, C., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1355

Cabedo, R., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Implementación Orientada a Aspectos en C# de un sistema robótico tele-operado. III Taller de Desarrollo de Software Orientado a Aspectos de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2005. 1363

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1371

Navarro, E., Letelier, P., Mocholí, J. A.,Ramos, I. A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems, XLVI(5), Special Issue on Systems Analysis and Design. 2006. 1379

Cabello, M. E., Costa, C., Ramos, I.,Carsí, J. A. Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System. Journal of the WSEAS Transactions on Information Science and Applications, Issue 10, Volume 3. 2006. 1391

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. Proc. of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006). 2006. 1399

Pérez, J., Navarro, E., Letelier, P., Ramos, I. A Modelling Proposal for Aspect-Oriented Software Architectures. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). 2006. 1415

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. A Goal-Oriented Approach for Safety Requirements Specification. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06). 2006. 1425

Limón, R., Ramos, I., Torres, J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development. Proc. of the International Conference the Computational Science and Its Applications (ICCSA 2006) . 2006. 1433

Limón, R., Ramos, I. Using Styles to Improve the Architectural Views Design. Proc. Of the International Conference on Software Engineering Advances (ICSEA 2006). 2006. 1443

Cabello, M. E., Costa, C., Ramos, I., Carsí, J. A. Generic Architecture of an Expert System for Diagnosis. Proc. of the 10th WSEAS International Conference on Computers. 2006. 1449

Cabello, M. E., Ramos, I., Costa, C. Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico. Actas del XIX Congreso Nacional y V Congreso Internacional de Informática y Computación del ANIEI-CIICC´07. 2006. 1455

Ali, N., Millan, C., Ramos, C. Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model. Proc. of the 8th International Sym-

posium on Distributed Objects and Applications (DOA). 2006. 1463

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes. IEEE Latin America Transactions, Special Edition JISBD'06, vol. 5, issue 4. 2007. 1481

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Mobile Ambients in Aspect-Oriented Software Architectures. Proc. of the IFIP Working Conference on Software Engineering Techniques - SET 2006. 2006. 1489

Navarro, E., Letelier, P., Reolid, D., Ramos, I. Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques. Proc. of the 15th International Conference on Information System Development. 2006. 1501

Navarro, E., Letelier, P., Ramos, I. Requirements and Scenarios: running Aspect Oriented Software Architectures. Proc. of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). 2007. 1515

Limón, R., Ramos, I. Analyzing Styles of the Modular Software Architecture View. Proc. of the First European Conference on Software Architecture (ECSA 2007). 2007. 1521

Limón, R., Ramos, I. Relating Software Architecture Views by Using MDA. Proc. of the ICCSA 2007 International Conference on Computational Science and its Applications. 2007. 1525

Limón, R., Ramos, I., Morales, M. Zárate, J. Analyzing and Designing Software Architecture Views driven by their Relationships. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 1537

Pérez, J., Costa, C., Carsí, J. A.,Ramos, I. Verificación de Modelos Arquitectónicos Orientados a Aspectos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos. 2007. 1547

Costa, C., Ali, N., Pérez, J., Carsí, J. A., Ramos, I. Dynamic Reconfiguration of Software Architectures Through Aspects. Proc. of the First European Conference on Software Architecture (ECSA'07). 2007. 1557

Limón, R., Cabello, M. E., Ramos, I. Establishing Relations among Software Architecture Views through MDA for SPL. Proc. of the 14th International Congress on Computer Science Research (CIICC'07). 2007. 1563

Pérez, J., Ali, N., Carsí, J. A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J. A. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Journal of Information and Software Technology 50 (2008) . 2008. 1575

Ali, N., Solís, C., Ramos, I. Comparing architecture description languages for mobile software systems. Proc. of the First International Workshop on Software Architectures and Mobility at ICSE'08. 2008. 1597

Navarro, E., Gómez, A., Letelier, P., Ramos, I. MORPHEUS: a supporting tool for MDD. Proc. of the18th International Conference on Information Systems

Development (ISD2009). 2009. 1603

Ali, N., Ramos, I., Solís, C. Ambient-PRISMA: Ambients in Mobile Aspect-Oriented Software Architecture. Journal of Systems and Software 83. 2010. 1615

Pérez, J., Ramos, I., Carsí, J.A., Costa, C.: Model-Driven Development of Aspect-Oriented Software Architectures. Journal of Universal Computer Science, vol. 19, no. 10. 2013. 1637

VOLUMEN 4: INTERFACES DE USUARIO (2000-2002) - GESTIÓN DE MODELOS (2002-2006)

PARTE VI: INTERFACES DE USUARIO - IDEAS (2000-2002)

Prólogo de María Dolores Lozano 1701

Lozano, M.D., Ramos, I. An Object Oriented Approach for the Specification of User Interfaces. Proceedings of PACRIM'99 Conference. 2000. 1703

Lozano, M. D., Ramos, I., González, P. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development". Proc. of the IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000). 2000. 1707

Lozano, M. D, González, P., Ramos, I. Models and Ontologies for the Specification of User Interfaces. Proc. of the Workshop on Model Engineering at ECOOP'00, 2000. 1717

Lozano, M. D., Ramos, I. Aproximación Basada en Modelos en la especificación de Interfaces de Usuario. Actas de las I Jornadas de Interacción Persona-Ordenador (INTERACCIÓN-20), 2000. 1725

Lozano, M. D., González, P., Ramos, I. User Interface Generation. Current Trends. INFORMATIK /INFORMATIQUE. Journal of the Swiss informatics societies. Vol. 2. 2001. 1733

Lozano, M. D., González, P., Ramos, I. IDEAS: An Object-Oriented Environment for developing User Interfaces". Proc. of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01). 2001. 1739

Lozano, M. D., González, P., Ramos, I., Montero, F., Molina, J. P. Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.16.2002. 1749

Lozano, M., González, P., Ramos, I. A Graphical User Interface Development Tool. 16th British Computer Society Conference on Human-Computer Interaction (BHCI'02). 1759

Lozano, M., González, P., Montero, F. Molina, J. P., Ramos, I. Integrating Usability within the User Interface Development Process of Web Applications. Proc. of the 2nd International Workshop on Web Oriented Software Technology at ECOOP'02. 2002. 1763

Montero, F. Lozano, M. D., González, P., Ramos, I. Designing Web Sites by Using Design Patterns. Proc. of SugarLoafPLoP 2002. 2002. 1779

PARTE VII: GESTIÓN DE MODELOS

Prólogo de José Ángel Carsí 1805

Carsí J.A, Ramos I., Silva J., Pérez J., Anaya V. Un Generador Automático de Planes de Migración de Datos. I+D Computación, Vol. 1, No. 1. 2002. 1809

Silva J.F., Carsí J.A., Ramos I. Comparación automática de esquemas conceptuales orientados a objeto, Revista Ingeniería Informática - Revista electrónica del Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción - Chile. 2002. 1821

Pérez J., Ramos I., Anaya V., Cubel J.M, Domínguez F., Boronat A., Carsí J.A. Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. Proc. of the Workshop on Software Evolution Through Transformations (SET'02) at ICGT2002. 2002. 1833

Silva, J., Carsí, J. A., Ramos, I. An algorithm to compare OO-Conceptual Schemas. Proc. of the International Conference on Software Maintenance (ICSM'2002). 2002. 1847

Pérez J., Carsí J.A, Ramos I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR'2002) at the ICSM'02. 2002. 1857

Pérez, J., Carsí, J. A., Ramos, I. ADML: A Language for Automatic Generation of Migration Plans. Proc. of the First Eurasian Conference on Advances in Information and Communication Technology. 2002. 1869

Boronat, A., Carsí, J. A., Ramos, I. Una plataforma semántica para la gestión de modelos. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos. 2003. 1879

Boronat, A., Pérez, J., Carsí, J. A.,Ramos, I. Two experiencies in software dynamics. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. 2004. 1889

Boronat, A., Carsí, J. A., Ramos, I. An Algebraic Baseline for Automatic Transformations in MDA. Proc. of the Workshop on Software Evolution Through Transformations at ICGT2004. 2004. 1919

Boronat, A., Pedrós, J., Carsí, J. A., Ramos, I. Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004). 2004. 1935

Boronat, A., Carsí, J. A., Ramos, I., Pedrós, J. An Approach for Cross-Model Semantic Transformation on the . NET Framework. Proc. of the .NET Technologies'2005 Conference. 2005. 1947

Boronat, A., Boronat, A., Carsí, J. A.,Ramos, I. Una Arquitectura para la Definición

de Metáforas Gráficas para Metamodelos. Revista IEEE América Latina, Special Edition - JISBD'2004. 2005. 1957

Silva, J., Carsí, J. A., Ramos, I. Análisis teórico experimental de criterios de comparación de esquemas conceptuales orientados a objeto. Computación y Sistemas, Vol. IX no.1. 2005. 1967

Boronat, A., Carsí, J. A., Ramos, I. Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. Proc. of the 9th European Conference on Software Maintenance and Reengineering. 2005. 1983

Boronat, A., Carsí, J. A., Ramos, I. Automatic Support for Traceability in a Generic Model Management Framework. Proc. of the European Conference on Model Driven Architecture - Foundations and Applications. 2005. 1995

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). 2005. 2011

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos - JISBD'05. 2005. 2019

Queralt, P., Hoyos, L., Boronat, A., Carsí, J. A., Ramos, I. Un motor de transformación de modelos con soporte para el lenguaje QVT Relations. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). 2006. 2025

Boronat, A., Carsí, J. A., Ramos, I. Exogenous Model Merging by means of Model Management Operators. Proc. of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change at ICGT 2006. 2006. 2035

Boronat, A., Carsí, J. A., Ramos, I.,Letelier, P. Formal Model Merging Applied to Class Diagram Integration. Proc. of the International ERCIM Workshop on Software Evolution (2006). 2006. 2055

Boronat, A., Carsí, J. A., Ramos, I. Algebraic Specification of a Model Transformation Engine. Proc. of the Fundamental Approaches to Software Engineering Workshop (FASE'06) at ETAPS'06. 2006. 2061

Boronat, A., Ramos, I., Carsí, J. A. Definition of OCL 2. 0 Operational Semantics by means of a Parameterized Algebraic Specification. Proc. of the 1st International Workshop on Algebraic Foundations for OCL and Applications. 2006. 2077

Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J. A. An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework. Proc. of the Second European Conference on Model Driven Architecture. 2006. 2093

Gómez, A., Boronat, A., Hoyos, L., Carsí, J. A., Ramos, I. Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos.

Actas de las XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006). 2006. 2109

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I., Täubner, C., Eckstein, S. Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2119

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2129

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2135

Lloret, E., Escobar, S., Palomar, M., Ramos, I. Incremental and adaptive Software Systems Development of Natural Language Applications. Proc. of the 22nd International Conference on Information Systems Development (ISD 2013). 2013. 2143

VOLUMEN 5: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

PARTE VIII: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

Prólogo de Emilio Insfrán y Silvia Abrahão 2177

Carsí, J. A., Insfrán, E., Abrahao, S., Genero, M., Piattini, M., Ramos, I. ¿Cómo utilizar experimentos controlados en la selección de transformaciones QVT para obtener diagramas de clases UML más fáciles de entender? Proc. of the 8th Argentine Symposium on Software Engineering (ASSE 2007). 2007. 2179

Limón, R. Ramos, I. Aragón, R. Transformación de Vistas Arquitectónicas Orientada por Modelos. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 2189

Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Marco de Trabajo basado en MDA para la Medición Genérica del Software. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2199

Mora, B., García, F., Francisco Ruíz, Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Software Measurement by Using QVT Transformations in an MDA Context (ICEIS 2008). Proc. of the Tenth International Conference on Enterprise Information Systems, Volume DISI. 2008. 2209

Abrahao, S., Genero, M., Insfrán, E., Carsí, J. A., Ramos, I., Piattini, M. Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In Rech, J. and Bunse, C. (Eds.), Model-Driven Software Development: Integrating Quality Assurance. 2008. 2217

Insfrán, E., Carsí, J. A., Abrahao, S., Genero, M., Piattini, M., Ramos, I. Towards Quality-Driven Model Transformations: A Replication Study. Proc. of the International Workshop on Empirical Studies of Model-Driven Engineering (ESMDE 2008) at MODELS 2008. 2008. 2247

Cabello, M. E., Ramos, I. A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines. Proc. of the International Conference of Health Informatics- HEALTHINF´08. 2008. 2257

Cabello, M. E., Ramos, I. Variability Management in Product Lines for Decision Support Systems. Proc. of the 10th International Conference on Enterprise Information Systems (ICEIS´08). 2008. 2267

Cabello, M. E., Ramos, I. The Baseline: the milestone of Software Product Lines for Expert Systems automatic development. Proc. of the 9th Mexican International Conference on Computer Science. 2008. 2277

Gómez, M., Gómez, A., Cabello, M. E.,Ramos, I. BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. Actas del VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). 2009. 2287

Cabello, M. E., Ramos, I., Gómez, A., Limón, R. Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for Expert Systems development. Proc. of the 1st Asia Conference on Intelligent Information and Database Systems (ACIIDS´09). 2009. 2297

Cabello, M. E., Ramos, I. Expert Systems development through Software Product Lines techniques. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2303

Cabello, M. E., Ramos, I. Model-Driven Development of Decision Support Systems: tackling the variability problem. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2315

Gómez, A., Ramos, I. Cardinality-based feature models and Model-Driven Engineering: fitting them together. Proc. of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. 2010. 2327

Gómez, A., Cabello, M. E., Ramos, I. BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. Proc. of the 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010) at SPLC 2010. 2010. 2335

Gómez, A., Ramos, I. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. Proc. of the 19th International Conference on Information Systems Development (ISD'10) 2010. 2343

Abrahao, S., Javier González, Insfrán, E.,Ramos, I. Software Evolution in Model-Driven Product Line Engineering. ERCIM News 88, Special Theme: Evolving Software. 2012. 2355

Cabello, M. E., Ramos, I., Gutiérrez, J. R., Gómez, A., Limón, R. SPL Variability Management, Cardinality and Types: an MDA Approach. International Journal of Intelligent Information and Database Systems (IJIIDS), Vol. 6, No. 2. 2012. 2357

Insfrán, E., Abrahao, S., Javier González, McGregor, J. D., Ramos, I. A Multimodeling Approach for Quality-Driven Architecture Derivation. Proc. of the 21st International Conference on Information Systems Development (ISD2012). 2012. 2383

PARTE IX: LOS AMIGOS

Los amigos 2397

Apéndice 2441

PARTE IV

MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

PRÓLOGO DE MARÍA ALPUENTE



MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

PARTE IV

Mi relación científica y personal con Isidro empieza a principios de los 80, como profesor de la Facultad de Ciencias Físicas de la Universitat de València (UV) donde estudié. Mi primer recuerdo con él es un «viaje de prácticas» para conocer el Macintosh 128K. Así escomo empezó el interés por la informática de nuestra promoción según mi recuerdo.

Después de unos años como presidente de la Comisión Gestora de la Universidad de Castilla—La-Mancha, en 1986 Isidro regresa a Valencia y crea en la Universidad Politécnica (UPV) el núcleo de lo que más tarde se consolidaría como Grupo de investigación en Programación Lógica e Ingeniería del Software GPLIS. En aquel grupo nos integramos principalmente profesores de la Escuela Universitaria de Informática de la UPV que habíamos comenzado a cursar estudios de doctorado en la Escuela Técnica Superior de Ingenieros Industriales de la UPV aunque procedíamos de la otra universidad, mayoritariamente de Físicas.

La actividad del grupo GPLIS se inició con cuatro líneas de trabajo relacionadas, organizadas alrededor de un interés común en los métodos formales con una base lógica o declarativa: 1) Prototipado con especificaciones lógico-algebraicas, 2) Bases de datos deductivas, 3) Procesamiento del lenguaje natural y 4) Modelización conceptual de sistemas de información. Durante los comienzos del grupo no fue fácil entender para unos físicos si un modelo conceptual sería un modelo matemático, un diagrama o qué tipo de cachivache. Vamos, que Isidro lo tenía difícil con nosotros. A pesar de ello consiguió sumergirnos en la modelización conceptual, los métodos formales y la programación lógica y algebraica, y a esa primera época corresponden

los primeros cuatro trabajos incluidos en esta sección.

Aunque en un primer momento trabajamos conjuntamente en todas las líneas, con la incorporación de nuevos miembros y la evolución personal de los componentes iniciales del grupo las áreas de trabajo pronto se especializaron, conduciendo a una estructuración del equipo inicial en grupos de trabajo autónomos, unidos por el interés común en la tecnología declarativa.

Desde el primer momento en que Isidro mencionó la idea de un mecanismo revolucionario para ejecutar programas funcionales conocido como *narrowing* (o estrechamiento), personalmente la idea me cautivó. No en balde, gracias a su refinada cultura e inteligencia, Isidro siempre ha sido un magnífico encantador de serpientes además de un excelente comunicador y profesor. A mí me fascinaron tanto el propio *narrowing* como sus numerosas aplicaciones, en particular su utilidad para formalizar la semántica de la integración de los principales paradigmas de programación declarativa. Y todavía hoy, treinta años después, las nuevas aplicaciones encontradas para el *narrowing* me siguen fascinando, y es asombroso observar cómo mis intereses y experiencia han resultado «marcados» por aquellos primeros trabajos.

Entre 1986 y 1991 profundizamos juntos en el área de las semánticas, optimizaciones y aplicaciones de los lenguajes de programación lógico-funcionales integrados. Fueron necesarios aproximadamente dos años largos para crear una pequeña infraestructura y la bibliografía necesarias para hacer posibles los primeros trabajos de investigación en este campo. Es el periodo en que realicé mi tesis doctoral bajo la supervisión de Isidro y que completé en la Universidad de Pisa en 1991. El trabajo que cierra esta etapa es «An Equational Constraint Logic Approach to Database Design».

Todos estos trabajos históricos se enmarcaron en los proyectos de investigación CICYT SIN-TESIS (TIC88-0572) y CICYT PRO-TESIS (TIC91-0425), financiados por la Comisión Interministerial de Ciencia y Tecnología, cuyo objetivo era el diseño y desarrollo de entornos de generación automática de software asociadas usando técnicas formales basadas en distintos tipos de lógicas (clausal, ecuacional, temporal, dinámica, etc.), sus fusiones y extensiones. En los artículos que siguen se aprecia la evolución de una familia de lenguajes de especificación formal basados en las lógicas mencionadas que, a partir de un cierto punto, se fusionan con los conceptos del modelo orientado a objetos y otros formalismos como las Redes de Petri. Con la perspectiva de los años se aprecia bien cómo aquellos proyectos definieron nuestra trayectoria de investigación.

Isidro es un *sensei* que ha inspirado en Valencia a varias generaciones de científicos, mostrándose como un hombre del renacimiento, un verdadero polímata a cuyo lado aprendimos desde historia a navegación pasando por arte, decoración, cocina o pesca de erizos (incluso se prestó a oficiar la boda de Pacho y Marta en alta mar).

Isidro, tenemos contigo muchas deudas difíciles de saldar. A través de estas líneas recibe mi más afectuoso e infinito agradecimiento.

María Alpuente

IMPLEMENTACION DE UN SISTEMA DE GESTION DE BASES DE DATOS DEDUCTIVAS

por: Matilde CELMA*
Lidia MORENO*
Oscar PASTOR*
Isidro RAMOS**

* Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia.
** Universidad de Castilla-La Mancha.

RESUMEN

En el presente trabajo se ha desarrollado un Sistema de Gestión de Bases de Datos Deductivas en PROLOG de forma rápida y eficiente. El sistema permite crear cualquier Base de Datos relacional y efectuar las operaciones convencionales tanto conjuntistas como específicamente relacionales: crear relación, insertar tuplas, seleccionar, proyección, "join", unión, intersección... El trabajo realizado es una maqueta pero las ideas utilizadas permiten desarrollar prototipos con calidad industrial.

I. INTRODUCCION.

Desde la perspectiva de la Lógica una Base de Datos (B.D.) puede ser considerada o bien como una interpretación (aspecto semántico) o como una teoría de primer orden (aspecto sintáctico) que formalizan los conceptos de B.D. convencionales y B. D. deductivas (B.D.D.) respectivamente.

La relación entre lógica y B.D. es inmediata: desde la perspectiva semántica se ve una B.D. como un conjunto de relaciones r (A_1, ..., A_n) definidas sobre un dominio D (unión de los dominios de los atributos) asociando un símbolo de predicado por cada relación y un conjunto de constantes: D (el lenguaje de primer orden definido no tiene funciones). La interpretación de las fórmulas del lenguaje es la siguiente: las variables toman sus valores de D y r(e_1,...,e_n) es cierto ssi la tupla (e_a,...,e_n) pertenece a la relación r.

Desde el punto de vista sintáctico la teoría asociada T tiene a BD como único modelo, con lo que toda fórmula del lenguaje se deduce de la teoría ssi es cierta en BD. Una definición completa se da en [REITER 84].

Llamaremos Bases de Datos Deductivas (BDD) aquellas en las que nuevos hechos se deducen de los existentes y dentro de ellas distinguiremos las BDD definidas (BDDD) en las que una aserción o un hecho se expresan como una cláusula de Horn.

Desde el punto de vista operacional una BDDD consiste en lo siguiente:

i) Un conjunto de axiomas formados por los hechos elementales y las leyes deductivas (cláusulas de Horn).
ii) Un conjunto de restricciones de integridad: RI.
iii) Una metarregla: la negación como fracaso con final (en inglés: "negation as finite failure").

JULIO-SEPTIEMBRE 1987 - VOL. XX - Nº 3
Celma, M., Moreno, L., Pastor, O., Ramos, I. Implementación de un sistema de gestión de bases de datos deductivas. Revista de Informática y Automática, Vol. XX, Nº 3. 1987.

Esta metarregla establece que el fracaso al intentar probar P nos permite inferir NOT P.

Las cláusulas de Horn mas la negación como fracaso conducen al lenguaje PROLOG. Un programa PROLOG es por lo tanto similar a una BDDD y con estas premisas y utilizando un intérprete de PROLOG que funciona bajo MS-DOS en un IBM PC-AT, se ha implementado y es operativo un Sistema General de Bases de Datos relacional y deductivo con las características usuales y en el que se tienen además en cuenta las restricciones de integridad.

II.- SISTEMA GESTOR DE BASE DE DATOS CONSTRUIDO EN PROLOG.

La estructura de datos del modelo relacional es la relación o tabla, consistiendo una Base de Datos Relacional en un conjunto de las mismas. Para representar una relación en PROLOG, se introduce un predicado que coincidirá con el nombre de la misma. Estos predicados tienen un único argumento que es una lista de tantos elementos como atributos tenga la relación. Cada tupla de la relación quedará almacenada como un hecho de dicho predicado.

Se ha desarrollado el núcleo fundamental de un S.G.B.D. que permite:

* La definición de una B.D.
* La realización de requerimientos, tanto de consulta como de actualización.
* La especificación de restricciones de integridad.

Posteriormente, se detallarán estos tres puntos fundamentales del sistema.

Para facilitar al usuario la manipulación del S.G.B.D. se ha creado un menú de opciones a través del cual se muestran los procesos básicos del sistema. Dicho menú es el siguiente:

OPCIONES:

C: Carga Base de Datos
N: Nueva Base de Datos
A: Crear relación
S: Salvar Base de Datos
R: Requerimientos
X: Salida a Prolog

II.1.- DEFINICION DE LA BASE DE DATOS

La definición de la base de Datos consiste en facilitar el esquema de cada relación. Para ello se especifican los atributos, sus dominios y aquellos atributos considerados como clave primaria de la relación.

El esquema de la relación se almacena introduciendo un predicado denominado relación, con cuatro argumentos. El primero es el nombre de la relación, el segundo consiste en una lista con los nombres de los atributos, el siguiente argumento es una lista de dominios asociada a la anterior y por último tenemos la lista de atributos que constituyen la clave primaria.

Por ejemplo, la relación:

```
r(atr1:entero,atr2:caract,atr3:decimal).
```

donde atr1 es la clave primaria, se representaría de la siguiente forma:

```
relacion(r,[atr1,atr2,atr3],[entero,caract,
,decimal],[atr1]).
```

Los dominios que se han definido son: entero, decimal, caract y lógico.

Con el fin de asegurar la integridad de la B.D., con respecto a la duplicidad de nombres de relación, se genera para cada B.D. un predicado bd con un solo argumento que es una lista cuyos elementos son los nombres de todas las relaciones que constituyen la B.D. en cuestión. Así pues, tendremos un predicado en la B.D. de esta forma:

```
bd([r1,r2,...rn]).
```

Este esquema de relación almacenado nos permitirá verificar las restricciones de integridad que se expondrán en el apartado II.3.

II.2.- REQUERIMIENTOS

II.2.1.- SINTAXIS

Tanto los requerimientos de consulta como los de actualización se realizan con un lenguaje del tipo SQL (orientado a "Mapping"), con la siguiente sintaxis:

REQUERIMIENTOS DE ACTUALIZACION:

```
INSERTAR nombre-rel (lista de valores).

                              >
BORRAR DE r CON nombre_atr  = valor.
                              <
```

REQUERIMIENTOS DE CONSULTA:

```
SELECCIONAR DE r nombre_atr [nombre_atr]...

... CON nombre_atr  >
                    = valor.
                    <

PROYECTAR r SOBRE nombre_atr[nombre_atr]...
JOIN r1 r2 SOBRE nombre_atr.
UNIR r1 r2.
INTERSECTAR r1 r2.
```

II.2.2.-DESARROLLO.

En la ejecución de estos requerimientos, en general, se distinguen las siguien

tes fases:

a) Lectura y análisis sintáctico del requerimiento.

En primer lugar se procede a la lectura, caracter a caracter, del texto del requerimiento, filtrando los caracteres que actúan como separadores. Paralelamente se va construyendo una lista con las palabras que han sido analizadas. A continuación se realiza el análisis de la lista construida para identificar el requerimiento a satisfacer y los parámetros significativos del mismo: nombre de la relación o relaciones que intervienen, nombres de los atributos solicitados, condiciones, etc.

b) Construcción de cláusulas PROLOG.

En todos los requerimientos ha sido necesaria la construcción de cláusulas PROLOG a partir del nombre de la relación.

Estas cláusulas tendrán como predicado el nombre de la relación, y como argumentos una lista de valores, si se trata de una inserción, o bien de variables desinstanciadas en el resto de los requerimientos.

Para la construcción de una cláusula PROLOG se utiliza el predicado predefinido =.. .

En el caso de la inserción, después de la fase de lectura y análisis del requerimiento, se obtiene el nombre de la relación sobre la cual se desea realizar la inserción, así como los valores de los atributos que constituyen la nueva tupla.

Por ejemplo, sea R una variable instanciada al nombre de la relación y [Ca|Cola] una lista con los valores de los atributos. La inserción se realizaría de la forma siguiente:

X=..[R,[Ca|Cola]]

con lo cual se construye la cláusula:

X=R([Ca|Cola])

que se almacenará en la B.D. al ejecutarse:

asserta(X)

por el contrario en un requerimiento como PROYECTAR, la fase de lectura y análisis proporciona el nombre de la relación y los atributos que se desean obtener.

Por ejemplo, sea R una variable instanciada al nombre de la relación y [Ca|Cola] una lista con los nombres de los atributos solicitados. Para realizar la proyección construiremos una cláusula PROLOG con predicado R y con una lista de variables desinstanciadas, [A1|A2], como argumento, con tantos elementos como atributos tuviere la relación. Esta última información se obtendrá consultando el esquema de la relación. Así pues, con la ejecución de

X=..[R,[A1|A2]]

se construiría la cláusula

R([A1|A2])

que nos permitirá recuperar todas las tuplas de la relación.

c) Recuperación de tuplas.

Todos los requerimientos, a excepción de la inserción, contemplan una última fase de recuperación de tuplas.

Como se ha visto en la fase anterior se tiene construida una cláusula PROLOG del tipo R([A1|A2]) cuyo argumento es una lista de variables desinstanciadas. Ahora es necesario realizar el "matching" de dicha cláusula con todos los hechos de la Base de Datos que tienen el mismo predicado (tuplas de la relación). Para ello se utilizará el predicado predefinido "clause". Así:

```
.
.
X=..[R|[A1|A2]],
clause (X,Y),
.
.
```

Cada vez que se satisface el objetivo de clause (X,Y), la lista [A1|A2] quedará instanciada a los valores de una tupla de la relación, y a partir de este momento podemos visualizarlos todos o parte de ellos en función de lo solicitado.

II.2.3.- INSTRUMENTACION.

Se va a presentar la instrumentación de un requerimiento de actualización -INSERTAR-, y un requerimiento de consulta -PROYECTAR-

a) Lectura y análisis del requerimiento

Como se indicó en el apartado anterior, la fase de lectura es común a todos los requerimientos.

El requerimiento se introduce con la opción 'R' del menú principal, a la que le corresponde el siguiente código PROLOG:

```
opcion('R'):-write('Introduzca requerimien
to'), nl, read_in([A|B]),req(A,B),!.
```

donde:

read in ([A|B]) es una regla que permite la lectura carácter a carácter del requerimiento, devolviendo una lista [A|B] con las distintas palabras detectadas. La ca-

beza de esta lista será el identificador del requerimiento (SELECCIONAR, INSERTAR, ...)

A continuación se presenta el código de la regla read_in:

```
read_in([W|Ws]):-
        get0(C),put(C),palabra(C,W,W1),
,resto_pal(W,C1,Ws).
resto_pal(W,_,[]):- ultpal(W),!.
resto_pal(W,C,[W1|Ws]):-
      palabra(C,W1,C1),resto_pal(W1,C1,Ws).
palabra (C,W,C1):-
        simple_car(C),!,name(W,[C]),get0
        (C1),put(C1).
palabra (C,W,C2):-
       in_pal(C,NewC),!,get0(C1),resto_
      _pal2(C1,Cs,C2),name(W,[NewC|Cs]).
palabra(C,W,C2):- get0(C1),palabra(C1,W,
,C2).
resto_pal2(C,[NewC|Cs],C2):- in_pal(C,NewC),
,!,get0(C1),resto_pal2(C1,Cs,C2).
resto_pal2(C,[],C).
simple_car(46).
in_pal(C,C):-C>96,C<123.
in_pal(C,L):-C>64,C<91,L is C+32.
in_pal(C,C):-C>47,C<58.
in-pal(C,C):-C=60;C=61;C=62.
ultpal('.').
```

req(A,B) es una regla que según el valor del argumento A ejecuta el correspondiente requerimiento, como veremos a continuación pra los requerimientos, ejemplo que vamos a desarrollar.

b) Requerimiento INSERTAR r (lista de valores).

Este requerimiento se satisface con la regla

```
req('insertar',Q)
```

donde Q es una lista cuya cabeza será el nombre de la relación sobre la que se va a insertar, y su cola la lista de valores que componen la nueva tupla.

El código PROLOG de esta regla es el siguiente:

```
req('insertar',Q):-
     append([X|Y],[.],Q),
     com_r(X),!,
     restric([X|Y],VF),
R=..[X|VF],asserta(R).
```

donde:

comp-r(X) comprueba que la relación existe.

restric([X|Y],VF) comprueba que el número de valores v introducidos en el requerimiento coincide con el número de atributos especificados en el esquema de la relación, que dichos valores pertenecen al dominio correspondiente, y la unicidad de tuplas.

R=..[X|VF] crea una cláusula PROLOG de la forma X(Y), que representa una tupla de la relación.

asserta(R) almacena la cláusula en la Base de Datos.

c) Requerimiento PROYECTAR R SOBRE nombre atr [nombre atr]

Este requerimiento se satisface con la regla

```
req('proyectar',Q).
```

El código PROLOG correspondiente es:

```
req('proyectar',[X|Y]):-
    append([A|B],['.'],Y),Aux=[],
    A='sobre',
    relacion(X,AT,D,C),
    crear_fila(AT,Aux,Atributos),
    Clausula=..[X!Atributos],
    clause(Clausula,Z),escribir(Atributos,
    ,B,AT),nl,fail.
```

donde:

crear fila (AT,AUX,Atributos) es una regla que construye una lista ATRIBUTOS de variables desinstanciadas, tantas como atributos haya en el esquema de la relación, de la siguiente forma:

```
crear_fila([X|Y],Aux,Fila):-
     append(Aux,[D],L),
     crear_fila(Y,L,Fila).
crear_fila([],Aux,Fila):-Fila=Aux.
```

cláusula=..[X|Atributos] crea una cláusula PROLOG de la forma X(Atributos).

clause(clausula,Z) permitirá recuperar todas las tuplas de la relación.

escribir(Atributos,B,AT) visualiza de cada tupla recuperada los atributos solicitados, como se indica a continuacion:

```
escribir([A1|A2],B,[AT1|AT2]):-
     member (AT1,B),!,write(A1),write(' '),
     escribir(A2,B,AT2).
escribir([A1|A2],B,[AT1|AT2]):- escribir
                              (A2,B,AT2).
```

```
escribir([],B,[]):- !.
```

II.3.- RESTRICCIONES DE INTEGRIDAD.

Se han especificado un conjunto de reglas en PROLOG con el fin de asegurar la integridad de la B.D.. Estas se centran en dos puntos esenciales del proceso de generación del S.G.B.D.: construcción del esquema de una nueva relación de tuplas.

En la construcción del esquema de una nueva relación se verifica que dicha relación no exista en la B.D., que no hayan atributos duplicados y que los dominios asociados a estos sean correctos, o sea, que pertenezcan al conjunto de dominios previstos, los cuales se enumeran en el punto II.1.

Las reglas PROLOG que realizan estas tareas son test_rel, test_atr, test_dom respectivamente, cuyos códigos se muestra a continuación:

test_rel comprueba que la relación no existe.

```
test_rel(A,W):- not member(W,A).

test_rel(A,W):-write('**Error:Ya existe
relación**'),!,fail.
```

test_atr comprueba la no duplicidad de atributos.

```
test_atr(Wy,[]).

test_atr(Wy,H):-
[A|B]=H,append(H1,[A|H2],Wy),
append(H1,H2,H3),
not(member(A,H3)),
test_atr(Wy,B).

test_atr(Wy,H):-
   nl,write('**Error:Duplicidad atributos:'),
   [A|B]=H,write(A),!,fail.
```

test_dom comprueba la corrección de dominios.

```
test_dom([]).

test_dom(D):-
   [A|B]=D,member(A,[caract,entero,decimal,logico]),
   test_dom(B).

test_dom(D):-
   nl,write('**Error:dominio incorrecto:'),
   [A|B]=D,write(A),!,fail.
```

Por otra parte, en el momento de insertar una nueva tupla también existe una serie de comprobaciones obligatorias como la de verificar que para la relación en la que se quiere insertar la tupla se haya definido el esquema correspondiente, que el número de valores especificados en el requerimiento sea igual al número de atributos y pertenezcan al dominio de estos, para lo cual consultaremos con dicho esquema de la relación. Además de las restricciones apuntadas existe en el S.G.B.D. una restricción básica referente a la clave primaria, ya que es esencial la unicidad de claves para que una tupla de la relación pueda ser identificada a través del contenido de dicha clave.

Las reglas PROLOG encargadas de estas comprobaciones con:

comp-r comprueba que existe la relación que se le pasa como argumento, verificando que es una relación constituyente de alguna de las Bases de Datos introducidas.

```
com_r(A):-bd(X),member(A,X).

comp_r(A):-nl,write('**Error:No existe la relación'),!,fail.
```

restric verifica que el número de valores introducidos es el correcto, que los dominios asociados son válidos, y que la unicidad en la clave primaria se preserva.

Su código PROLOG es el sigueinte:

```
restric([R|V],VF):-
     relacion(R,A,D,C),
     num_va(V,A,L1,Ls),
     com_val(V,D,VF,R1),!,
     clave_prim([R|R1],A,C,).
```

donde:

num-va comprobará que si el número de valores es igual al número de atributos

```
num_va(V,A,L1,L2):-
     N is 0,longitud(V,N,L1),
     N2 is 0,longitud(A,N2,L2),
     !,mens_lg(L1,L2).
```

siendo longitud (A,N,L) una regla - PROLOG que devolverá en L el núemro de elementos de la lista A, siendo N una variable auxiliar que se irá incrementando conforme se recorra la lista A, y mens_lg(L1,L2) otra regla PROLOG que comprobará si L1=L2, avisando del error en caso contrario.

comp-val comprobará si los valores introducidos pertenecen al dominio respectivo.

```
comp-val([],[],VF,R):-R=VF

comp_val(B,D,VF,R):-
[X1|X2]=B,[Y1|Y2]=D,com(Y1,X1,H),
```

```
,!,append(VF,[H],V2),comp_val(X2,Y2,V2,R).
```

siendo dom(Y,X,H) una nueva regla en la que se analizará si el valor X pertenece al dominio Y, devolviendo en H dicho valor una vez verificada su corrección.

clave-prim se encargará de la unicidad de la clave primaria.

```
clave_prim([R|V],A,C):
    L1=[],crear_lista(A,C,V,L1,L2),
    X=..[R|L2],comprobar (X)
```

creando una cláusula PROLOG para la relación analizada, con todas las variables desinstanciadas excepto la correspondiente a la clave primaria. En comprobar se detectará si ya existe alguna tupla con ese valor de clave primaria estudiando la posible satisfacción de la cláusula construida.

III. CONCLUSIONES.

Se ha intentado durante todo el trabajo aplicar resultados obtenidos desde una perspectiva lógica-matemática al entorno de las Bases de Datos, a través del concepto de Bases de Datos Deductivas y utilizando un lenguaje de programación lógica, que ha sido el PROLOG. Con él se le ha dado existencia real a un Sistema de Gestión de Bases de Datos elemental, susceptible de mejoras posteriores por extensión de sus prestaciones, pero que justifica de modo suficiente, desde un punto de vista práctico, todas las ideas expuestas en este trabajo.

Pensamos que la Lógica proporciona una firme base teórica desde la que abordar una teoría general de las Bases de Datos. Y no hay que pasar por alto que el problema de "Lógica y Bases de Datos" constituye el corazón del trabajo actual realizado en Japón en su 'Proyecto de Quinta Generación', lo que hace que esta línea de trabajo constituya uno de los temas de estudio de mayor interés entre la comunidad científica actual.

IV. BIBLIOGRAFIA.

(1) COLMERAUER, A., KANOUI, H. y VAN CANEGHEM, M., "Prolog, Bases Théoriques et dévoluppements actuels". TSI Vol. 2, nº 4, 1983. pp.271-311.

(2) GALLAIRE, H., MINKER, J. y NICOLAS, J.M., "Logic and Databases: a deductive approach". Computing Surveys, vol. 16, nº 2, Junio 1984. pp. 153-185.

(3) RAMOS SALAVERT, I. "Colección de apuntes para Curso de Doctorado". E. T.S.I.I.-U.P.V. Valencia,Curso 85-86.

(4) GREEN, C., "Theorem proving by resolution as a Basis for question-answering Systems". 1969. Machine Intelligence 4, B pp. 183-205.

(5) MINKER, J., "On deductive relational Databases". Proceedings of fhe 5th. International Conference on Collecti va Phenomena. Ed. Annlas of the New York Academy of Sciences, Vol. 10. pp. 181-200.

(6) CLOCKSIN, W.F. y MELLISH, C.S., "Programming in PROLOG". Springer-Verlag, 1984.

(7) COLMERAUER, A., "PROLOG, Lenguaje de la Inteligencia Artificial". Mundo Científico, nº 41, pp. 1.072-1.082. Nov. 1984.

V. APENDICE.

menú.

```
opciones:
        C: Carga base de datos:
        N: Nueva base de datos:
        A: Crear relación:
        S: Salvar base de datos:
        R: Requerimientos:
        X: Salida a Prolog:

Selecciona opción: n

** PREPARADA NUEVA B.D. **

opciones:
        C: Carga base de datos:
        N: Nueva base de datos:
        A: Crear relación:
        S: Salvar base de datos:
        R: Requerimientos:
        X: Salida a Prolog:

Selecciona opcion: a

**TERMINAR ATRIBUTOS Y CLAVES CON "FIN"**

Nombre de la Relación:
piezas.
Atributo-1:codpieza.
Dominio-1:caract.
Atributo-2:descripción.
Dominio-2:caract.
Atributo-3:exist.
Dominio-3:entero
Atributo-4:fin.
Clave primaria:codpieza.
Clave primaria:fin.

opciones:
        C: Carga base de datos:
        N: Nueva base de datos:
        A: Crear relación:
        S: Salvar base de datos:
        R: Requerimientos:
        X: Salida a Prolog:

Selecciona opcion:a
```

```
**TERMINAR ATRIBUTOS Y CLAVES CON "FIN"**

Nombre de la Relación:
pedidos.
Atributo-1:codprov.
Dominio-1:caract.
Atributo-2:codpieza.
Dominio-2:caract.
Atributo-3:cantidad.
Dominio-3:entero.
Atributo-4:fin.
Clave primaria:codprov.
Clave primaria:codpieza.
Clave primaria:fin.

opciones:
        C: Carga base de datos:
        N: Nueva base de datos:
        A: Crear relación:
        S: Salvar base de datos:
        R: Requerimientos:
        X: Salida a Prolog:

Selecciona opción:r

Introduzca requerimiento

insertar pedidos (123 32 100).

Introduzca requerimiento

insertar piezas(32 tuerca 500).

Introduzca requerimiento

insertar pedidos (123 22 50).

Introduzca requerimiento

seleccionar de pedidos codprov codpieza
cantidad con cantidad  25.

123 32 100
123 22 50

Introduzca requerimiento

proyectar piezas sobre codpieza exist.

32 500
22 100

Introduzca requerimiento

join piezas pedidos sobre codpieza.

tuerca 500 123 32 100
tornillo 100 123 22 50

opciones:
        C: Carga base de datos:
        N: Nueva base de datos:
        A: Crear relación:
        S: Salvar base de datos:
        R: Requerimientos:
        X: Salida a Prolog:

Selecciona opción:a

**TERMINAR ATRIBUTOS Y CLAVES CON "FIN"**

Nombre de la Relación:
piezas2.
Atributo-1:codpieza.
Dominio-1:caract.
Atributo-2:descripción.
Dominio-2:caract.
Atributo-3:exist.
Dominio-3:entero.
Atributo-4:fin.
Clave primaria:codpieza.
Calve primaria:fin.

opciones:
        C: Carga base de datos:
        N: Nueva base de datos:
        A: Crear relación:
        S: Salvar base de datos:
        R: Requerimientos:
        X: Salida a Prolog:

Selecciona opción:r
Introduzca requerimiento

insertar piezas2(45 mesa 50).

Introduzca requerimiento

insertar piezas2(22 tornillo 100).

Introduzca requerimiento

unir piezas piezas2.

32 tuerca 500
22 tornillo 100
45 mesa 50

Introduzca requerimiento

intersectar piezas piezas2.

22 tornillo 100

opciones:
        C: Carga base de datos:
        N: Nueva base de datos:
        A: Crear relación:
        S: Salvar base de datos:
        R: Requerimientos:
        X: Salida a Prolog:

Selecciona opción:x
```

ENTORNO DE PROGRAMACION LALI IMPLEMENTADO EN PROLOG

Por: María Alpuente Frasnedo*
Vicente J. Botti Navarro*
Isabel Galiano Ronda*
Emilio Sanchís Arnal*
Francisco Vila Aparicio*
Isidro Ramos Salavert**

* Departamento de Sistemas Informáticos y Computación.
Universidad Politécnica de Valencia
** Universidad de Castilla-La Mancha

RESUMEN: Los entornos de especificación permiten describir universos de forma declarativa dando lugar a sistemas que permiten deducir de ellos respuestas a preguntas que se les formulan. Un Universo estará formado por un conjunto de objetos, que modulizan la realidad en la que se plantea el problema, formalizados mediante tipos abstractos algebraicos (TAD) con axiomas ecuacionales, cuya semántica vendrá dada por el Algebra inicial de términos. El trabajo consiste en la implementación de un Entorno de Especificación en el que los programas se expresan en forma declarativa (jerarquía de TAD) y las preguntas se realizan mediante términos no interpretados del Algebra. Los términos evaluados en el Algebra de términos (sin interpretar) usando las ecuaciones orientadas como reglas de reescritura, hasta dar como resultado un término ya no reescribible. Los objetos pueden jerarquizarse (vía importación/exportación de otros), los errores son tratados mediante el concepto de sub-sort, se aborda el problema de no finitud introducido por la conmutatividad y se permiten ecuaciones condicionales. El sistema ha sido implementado en Prolog y es operativo sobre un IBM-PC modelo AT.

INTRODUCCION.

La principal y última meta de un proceso de desarrollo Software es la realización de un sistema Software escrito en un lenguaje de Programación.

Muchos lenguajes de Programación son capaces de soportar la Implementación de un sistema Software, pero cobran interés, cada vez más, en Informática, nuevas ideas acerca de la Especificación, Diseño, Validación e Implementación de sistemas capaces de deducir de ellos las respuestas a preguntas que se les formulan.

Prolog abandona el principio de la Programación Imperativa en favor de la Declarativa, que consiste en presentar en un formalismo adecuado conocimientos sobre un determinado tema, de donde se puede deducir respuestas a las preguntas formuladas.

En el trabajo que se presenta, las fuentes de conocimiento que manipulará Prolog son descripciones de Universos formadas por el conjunto de objetos que participan de la modelización de la realidad en que se plantea el problema, formalizados mediante Tipos Abstractos Algebraicos con axiomas ecuacionales, cuya semántica vendrá dada por el Algebra Inicial de Términos. Las respuestas de Prolog serán Términos del Algebra Libre que es el Tipo Abstracto.

El trabajo desarrolla un Entorno de Especificación y Validación (declarativo,no imperativo), en el que los programas se expresan en forma declarativa (Tipos Abstractos Algebraicos), y como parte imperativa se introducen Términos no interpretados del Algebra.

El término será evaluado en el Algebra de Términos (sin interpretar), usando las ecuaciones orientadas como reglas de reescritura, y la respuesta será el término final ya no reescribible, aunque no se contempla que el sistema de reescritura asociado con las ecuaciones sea confluente (conduzca a un resultado único)ni neotheriano (proporcione el resultado en un nú

Alpuente, M., Botti, V., Galiano, I., Sanchis, E., Vila, F., Ramos, I. Entorno de programación LALI implementado en PROLOG. Revista de Informática y Automática . Vol. XX, Nº 4. 1987.

mero finito de etapas).

La utilización de una técnica descendente, con tipos abstractos de datos, y el entorno de programación LALI, nos permitirá disponer de un medio semiautomático de validación, ya que el sistema probará la validez de las especificaciones, y cuando especifiquemos un universo en términos de otros universos, únicamente se deberán probar los axiomas de la especificación de este universo, sin necesidad de probar, en ese instante, la validez de los universos utilizados en su especificación, que serán probados por separado.

1.- EL LENGUAJE.

El lenguaje que nos servirá para formalizar las descripciones de Universos - tratados por Prolog es el Lenguaje Algebraico Lógicamente Interpretado: LALI.(1)

Ejemplo: Supongamos que se pretende resolver problemas en el Universo de los Directorios. Definimos dicho Universo mediante el Tipo Abstracto genérico "Directorio" en la forma usual:

```
objeto directorio;
sorts directorio, fichero/bool;
ok-ops
crear_d    :                  > directorio;
vacio      :directorio        > bool;
crear_f    :fichero,directorio>directorio;
borrar_f   :fichero,directorio>directorio;
ultimo_creado:directorio      > fichero;
buscar_f   :fichero,directorio>bool.
error-ops
fichero_nulo                  > fichero;
underflow                     > directorio.

vars
d: directorio;
f: fichero.

ok-eqs
vacio (crear_d)=true;
vacio (crear_f(f,d))=false;
borrar_f (f,crear_f(f,d))=d;
ultimo_creado  (crear_f(f,d))=f;
buscar_f(f,crear_d)=false;
buscar_f(f,crear_f(f,d))=true;
buscar_f(f,borrar_f(f,d))=false.

error-eqs
borrar_f(f,crear_d)=underflow;
ultimo_creado(crear_d)=fichero_nulo.
otejbo
```

Un programa LALI sería:

```
buscar_f(f,borrar_f(f,crear_f(f,crear_d))).
```

con resultado trivial:

```
false
```

que se obtendrá aplicando las ecuaciones orientadas desde el término más interno.

Enriquecemos el lenguaje LALI introduciendo en su sintaxis el concepto de SUBSORTS (2). Este concepto permitirá la implementación de "jerarquías múltiples" de "tipos", y al mismo tiempo preveerá un más sencillo y eficiente tratamiento de errores en la evaluación de los términos, detectando expresiones carentes de significado, sin necesidad de recurrir a las error-ops/eqs.

La sintaxis de declaración de subsorts en LALI será la siguiente:

```
subsorts sort-list   '  ' sort-list ...
```

A nivel semántico, se requerirá la detección de posibles circuitos en el grafo "ser subsort de", es decir, no se permiten declaraciones del tipo:

```
subsorts NAT < INT < NAT
```

Volviendo al ejemplo anterior (Universo de los Directorios), y utilizando el concepto de subsorts se puede especificar el objeto "Directorio" de la siguiente manera:

```
objeto directorio;
sorts directorio, fichero/bool;
subsorts vacio-dir<no-vacio-dir<directorio;
ok-ops
crear_d       :                       > vacio-dir;
vacio         : directorio            > bool;
crear_f       : fichero,directorio    > no-vacio-dir;
borrar_f      : fichero,no-vacio-dir  > directorio;
ultimo creado : no-vacio-dir          > fichero;
buscar_f      : fichero,no-vacio-dir  > bool.
vars
d: no-vacio-dir;
f: fichero.
ok-egs
vacio(crear_d)=true;
vacio(crear_f(f,d))=false;
borrar_f(f,crear_f(f,d))=d;
```

```
ultimo_creado(crear_f(f,d))=f;
buscar_f(f,crear_d)=false;
buscar_f(f,crear_f(f,d))=true;
buscar_f(f,borrar_f(f,d))=false.
```

otejbo

2.- EL TRADUCTOR

Una vez formalizado el Universo mediante el Lenguaje de Programación LALI (que sirve como entrada a nuestro sistema), se realiza la traducción, que proporcionará como objeto un programa Prolog. La traducción se realizará de acuerdo con el siguiente esquema:

OBJETOS LALI		OBJETOS PROLOG
CTES	crear_d	...ATOMOS crear_d...
VARS	f,d	...VARS _132,_124...
OPS	borrar_f,crear_f	...FUNCTORS borrar_f,crear_f..
TERMINOS	vacio(crear_d)	...TERMINOS vacio(crear_d)...
AXIOMAS	vacio(crear_d)=true	...HECHOS su(vacio, ,crear_d,true)...

Según dicho esquema, la traducción daría como resultado el siguiente programa objeto:

```
/*UNIVERSO DIRECTORIO TRADUCCION*/

nueva([direct,fichero]).
nueva([bool]).

importada([bool]).
importada([]).

sub([vaciodir,novaciodir,direct]).

v([d]).
v([a,b,f]).

tds([a,b,bool]).
tds([d,novaciodir]).
tds([f,fichero]).

ops([true,bool]).
ops([false,bool]).
ops([&,bool,bool,bool]).
ops([or,bool,bool,bool]).
ops([~,bool,bool]).
ops([creard,vaciodir]).
ops([vacio,direct,bool]).
ops([crearf,fichero,direct,novaciodir]).
ops([borrarf,fichero,novaciodir,novaciodir]).
ops([ultimocreado,novaciodir,fichero]).
ops([buscarf,fichero,novaciodir,bool]).

su(ultimocreado,crearf(_907,_912,_907)):-true.
su(vacio,crearf(_941,_942),false):-true.
su(vacio,creard,true):-true.
su(~,false,true):-true.
su(~,true,false):-true.

sb(borrarf,_1066,crearf(_1066,_1068),_1068):-true.
sb(buscarf,_1101,borrarf(_1101,_1109),false):-true.
sb(buscarf,_1137,crearf(_1137,_1145),true):-true.
sb(buscarf,_1173,creard,false):-true.
sb(or,true,_1206,true):-true.
sb(or,false,_1238,_1238):-true.
sb(&,false,_1270,false):-true.
sb(&,true,_1302,_1302):-true.
```

COMENTARIOS AL LISTADO

1.- El nombre del objeto será el de la Base de Datos en que se almacenará la traducción del objeto.

2.- La declaración de sorts origina la construcción de dos listas, que contienen:

i) Los sorts de nueva creación (nueva).
ii) Los sorts importados del entorno (importada).

Esta importación de sorts supone consultar (consult) los ficheros en que se almacenan tales objetos ya traducidos.

3.- La declaración de subsorts origina la construcción de una lista que contiene los subsorts jerárquicamente ordenados, comporbándose que no existe circuito en dicha declaración.

4.- Los operadores declarados con el perfil que aparece en las ok-ops dan lugar a la construcción de cláusulas con símbolo de predicado "ops", que almacenan nombre y tipo de los respectivos argumentos del operador en cuestión, (así como su codominio), para una posterior comprobación estática de tipos en la traducción de la declaración de variables y en la formalización de los axiomas ecuacionales, siempre que los tipos en el perfil hayan sido declarados como sorts o subsorts.

según la aridad del operador se efectúan dos tipos de tratamiento:

- Operadores cero-arios: Inclusión en la tabla de constantes.

- Resto de operadores: Declaración como operadores Prolog.

```
op(P,A,F).
```

donde: P=5 es la prioridad del operador
A=fx/fxx es su asociatividad.
F=⟨nombre_del operador⟩ es su nombre

5.- La traducción de la declaración de variables supone construir la tabla de símbolos de traductor, si tuvo éxito la comprobación estática de tipos, y lós identificadores de variable no se repiten.

6.- Los axiomas ecuacionales se traducen según el siguiente criterio:

- Se agrupan todos los operadores de igual aridad bajo el mismo símbolo de predicado, por ejemplo:

"su" para los unarios (buscar_f,vacio...).
"sb" para los binarios (borrar_f,crear_f...).

Esto, siempre que el analizador de ecuaciones no detecte errores de sintaxis o semántica en la ecuación.

3.- EL EVALUADOR

El sistema incorpora un predicado evaluador que reescribe los términos del Algebra Libre, enunciado del problema a resolver, proporcionando un resultado canónico ya no reescribible (1). Se ha dotado al evaluador de mecanismos para solucionar el problema de no terminación en la reescritura de términos, derivado del uso de reglas que expresan la propiedad conmutativa marcando cualquier regla conmutativa utilizada en el proceso de reescritura - para impedir su uso hasta encontrar un término diferente al que provocó su aplicación.

Como ejemplo de funcionamiento se recoge la evaluación del siguiente término en el universo de los directorios:

```
?-evaluar.

TERMINO A EVALUAR:

vacio(crear_f(f,crear_d)).

RESULTADO:

false.

MORE ? (S/N)
n
```

4.- EL ENTORNO.

Las opciones que se han implementado en el menú permiten gestionar una biblioteca de objetos sobre las que se podrán realizar las siguientes acciones:

i) INTERPRETAR (clave I/i)

Su funcionamiento viene esquematizado por la siguiente expresión regular:

[Dar universo [Dar término especificación. Obtener resultado]$^+$]$^+$

El universo se puede proporcionar en forma interactiva, o bien dando el nombre de un fichero fuente ya existente en el entorno. En cualquier caso se realiza su traducción antes de entrar en el bucle de evaluación.

ii) ALMACENAR FUENTE (Clave A/a)

Esta opción permite salvar un objeto LALI especificado cuando se pasó por la opción anterior tomando como fichero - fuente el USER, o por la opción TRADUCIR UNIVERSO (que se tratará mas adelante, en las mismas circunstancias).

iii) VISUALIZAR FICHERO (clave V/v)

Esta opción muestra en pantalla el contenido del fichero a visualizar.

iv) DESTRUIR FICHERO (clave D/d).

Elimina del directorio el fichero especificado.

v) SALVAR OBJETO (clave S/s)

Crea un fichero conteniendo la traducción del último objeto especificado, es decir, el programa objeto Prolog que resulta de la opción TRADUCIR/INTERPRETAR.

vi) TRAER FICHERO (clave T/t)

incorpora a la base de datos Prolog el fichero Prolog requerido.

vii) ESPECIFICAR UNIVERSO (clave U/u)

Realiza la traducción del objeto especificado tanto de forma interactiva como a partir de un fichero fuente de existencia previa.

viii) EVALUAR TERMINO (clave E/e)

Llama al evaluador y proporciona el resultado.

ix) MODIFICAR OBJETO (clave M/m)

Permite modificar, añadir o eliminar cláusulas en algún programa objeto e incorporar el nuevo programa resultante de la modificación a la biblioteca del sistema.

x) SALIDA A PROLOG (clave X/x)

Devuelve el control al nivel superior del intérprete Prolog.

A continuación se presenta un modelo de listado de ejecución, donde se puede ver con detalle el repertorio de opciones que ofrece el menú, así como el tratamiento de errores realizado por el sistema, con especial énfasis en el que realiza el traductor manipulando el programa fuente.

```
OPCIONES:

        I. INTERPRETAR
        V. VISUALIZAR FICHERO.
        A. ALMACENAR FUENTE.
        D. DESTRUIR FICHERO.
        S. SALVAR OBJETO.
        T. TRAER FICHERO.
        U. ESPECIFICAR UNIVERSO.
        E. EVALUAR TERMINO.
        M. MODIFICAR OBJETO.
        X. SALIDA A PROLOG.

Selecciona opción: i

FICHERO FUENTE:

nuevof.

importando ... int

TERMINO A EVALUAR:

top(push(1,s)).
```

```
RESULTADO:
1
MORE ? (S/N)
5
TERMINO A EVALUAR:
top(push(1,pop(push(2,empty)))).
RESULTADO:
1
MORE ? (S/N)
n
OTRO UNIVERSO ? (S/N)
n
OPCIONES:

        I. INTERPRETAR.
        V. VISUALIZAR FICHERO.
        A. ALMACENAR FUENTE.
        D. DESTRUIR FICHERO.
        S. SALVAR OBJETO.
        T. TRAER FICHERO.
        U. ESPECIFICAR UNIVERSO.
        E. EVALUAR TERMINO.
        M. MODIFICAR OBJETO.
        X. SALIDA A PROLOG.

Selecciona opcion: v
NOMBRE DEL FICHERO:
nuevof.
obj bstadk;
sorts stack/ int;
subsorts<ne-stack <ok-stack   stack;
ok-ops
empty : > ok-stack;
push : int ok-stack > ne-stack;
pop : ne-stack > ok-stack;
top : ne-stack > int;
depth : ok-stack > int.
vars
i : int ;
s : ok-stack.
ok-eqs
depth(empty)=cero;
depth(push(i,s))=inc(depth(s));
pop(push(i,s))=s
top(push(i,s))=i.
jbo
OPCIONES:

        I. INTERPRETAR.
        V. VISUALIZAR FICHERO.
        A. ALMACENAR FUENTE.
        D. DESTRUIR FICHERO.
        S. SALVAR OBJETO.
        T. TRAER FICHERO.
        U. ESPECIFICAR UNIVERSO.
        E. EVALUAR TERMINO.
        M. MODIFICAR OBJETO.
        X. SALIDA A PROLOG.

Selecciona opcion: s
NOMBRE DEL FICHERO:
pilas.
```

```
OPCIONES:

        I. INTERPRETAR.
        V. VISUALIZAR FICHERO.
        A. ALMACENAR FUENTE.
        D. DESTRUIR FICHERO.
        S. SALVAR OBJETO.
        T. TRAER FICHERO.
        U. ESPECIFICAR UNIVERSO.
        E. EVALUAR TERMINO.
        M. MODIFICAR OBJETO.
        X. SALIDA A PROLOG.

Selecciona opción: v
NOMBRE DEL FICHERO:
pilas.
nueva([stack]).
nueva([int]).
importada([int]).
importada([]).
v([i]).
v([E,J,s]).
tds([]).
tds([]).
tds([_562,_564,int]).
tds([i,int]).
tds([s,ok-stadk]).
ops([]).
ops([]).
ops([inc,int,int]).
ops([cero,int]).
ops([empty,ok-stack]).
ops([push,int,ok-stack,ne-stack]).
ops([pop,ne-stack,ok-stack]).
ops([top,ne-stack,int]).
ops([depth,ok-stack,int]).
su([]).
su([]).
su(top,push(_803,_808),_803):-true.
su(pop,push(_837,_833),_833):-true.
su(depth,push(_867,_868),inc(depth,_868)):-true.
su(depth,empty,cero):-true.
su(inc,_924,_925):-integer(_924),_925 is _924 + 1.
su(inc,_967,_967+1):-true.
su(_996,_997,_998):-_1004=[_996,_997],_998=..1004.
sb([])
sb(_1053,_1054,_1055,_1056):-_1063=
   =[_1053,_1054,_1055],_1056=..1063.
OPCIONES:

        I. INTERPRETAR.
        V. VISUALIZAR FICHERO.
        A. ALMACENAR FUENTE.
        D. DESTRUIR FICHERO.
        S. SALVAR OBJETO.
        T. TRAER FICHERO.
        U. ESPECIFICAR UNIVERSO.
        E. EVALUAR TERMINO.
        M. MODIFICAR OBJETO.
        X. SALIDA A PROLOG.

Selecciona opción: x
```

```
yes
?-
```

5.- AMPLIACIONES.

El trabajo queda abierto a posibles ampliaciones, entre las que cabe destacar las siguientes:

1.- Facilitar ayudas de ejecución por pasos (trazas...), para analizar si el sistema de reescritura asociado exhibe un comportamiento dotado de la propiedad de confluencia y terminación finita.

2.- Posibilidad de introducir cualificadores en el perfil de los operadores para dotarles de las propiedades de conmutatividad (CONM), asociatividad(ASSOC), y existencia del elemento identidad (ID: nombre_del_neutro), lo que evitará la introducción de las correspondientes ecuaciones explicitando tales propiedades.

3.- Parametrización de los objetos-con lo que se deberá introducir el concepto de TEORIA (Th), VISTA (View), y expresiones entre módulos (Make). (2)

6.- CONCLUSIONES.

El sistema implementado es la materialización de la maqueta de compilación propuesta para el Lenguaje de Especificación formal Algebraica LALI (1), y del entorno asociado a éste.

El sistema se apoya en técnicas suficientemente difundidas como para ser motivo de presentación de este trabajo, que abre perspectivas cómodas en la implementación de lenguajes.

7.- REFERENCIAS.

(1) RAMOS SALAVERT, ISIDRO.
"Un lenguaje de programación con tipos abstractos interpretados en Prolog: otra forma de compilar"
Revista de Informática y Automática, Abril-Junio 1984.

(2) GOGUEN, JOSEPH A.
"Parameterized Programming".
IEEE Trans. Software Engineering, Vol-10 N-5, Septiembre 1984.

(3) COLMERAUER, ALAIN.
"Prolog, lenguaje de la Inteligencia Artificial".
Mundo Científico, nº 41, pags. 1072-1082, Noviembre 1984.

(4) CLOCKSIN, W.F. & MELLISH, C.S.
"Programming in Prolog"
Springer Verlag, 1981.

(5) OREJAS, F.
"Tipos abstractos de datos y semántica de los lenguajes de programación".
Escuela Informática de Verano, Caldas de Bohí, Julio 1981.

(6) DERANSART, P.
"Derivation de programmes Prolog a partir de Specifications Algebraiques".
Note interne I.N.R.I.A., 1982.

(7) GUTTAG J.V., HOROWITZ E., MUSSER D.R.
"Abstract Data Types and Software Validation".
Communications ACM, December 1978, Vol.21 N. 12.

PROTOLOG
a Conceptual Schema Facility for automated Prototype Generation

González del Río, A.; Alpuente,M.; Casamayor, J.C.; Pastor, M.A.;
Ramírez, M.J.; Ramos, I.
Departamento de Sistemas Informáticos y Computación.
Universidad Politecnica de Valencia. Cno. de Vera s/n, Valencia (Spain)

ABSTRACT

This paper presents PROTOLOG: a Conceptual Schema Facility (CSF) for automated Prototype Generation. We introduce a new approach to conceptual modelling, which is proposed as an alternative to the operational and deductive approaches and covers the existing gap between them. The introduction of time concept is essential in the model and allows, not only to deal with historical information but also to dispose of a better expressiveness in the language. In this way, more accurate models of reality (as in the unified view of dynamic and static constraints) can be obtained. On the other hand, rapid prototyping makes the design phase easier, which allows the validation of the requirements. PROTOLOG/PG automatically translates PROTOLOG models into executable prototypes.

INTRODUCTION.

The construction of software components starting from a specification of the problem has two aspects that we are interested in outlining:

- the first one is the inherent paradigm supported by the language (i.e.: deductive, operational ...)
- the second one is the possibility of executing the phrases of the specification language automatically (generating object code or interpreting the specification).

Logical specification languages emerged at the moment when the execution of its phrases was possible and efficient. In a similar way, algebraic specification languages were implemented when rewriting techniques had reached a practical status.

The associated paradigm , i.e. the ability to capture the static and dynamic aspects of the Universe of Discourse (UoD in the following, see [1]) and the way they are represented in the language constructs originates different approaches.We will consider two of them, the former being state-oriented and the second action-oriented:

- Conceptual Deductive Models (DM).
- Conceptual Operational Models (OM).

The OM includes the concept of event/action [2][3] and defines, in an explicit way, the changes that occur in the Information System (i.e. the transitions in the associated Information Base (IB)). The DM is based on the derivation rule concept. The transitions scheme is implicit in the model. The characteristics of the two approaches have been compared and evaluated in [4] and [5] and the preferences point to the DM because of its more declarative nature and its better expressiveness (in particular with respect to information locality).

However, a specification is the initial step in the process of implementing an Information System (IS) and we need to translate (or to interpret) it in order to get the final result. In relation to the executability property, the OM paradigm seems better suited than the DM one and an usual way of execution is to generate a prototype (by hand or automatically).

Prototyping is becoming one of the key questions in IS implementation. It is almost the only way to verify or evaluate a given specification. With the aim of generating prototypes for OM, some attempts have been made. In most cases, it is not assumed that operations, constraints and queries can have access to any past state of the IB. As far as we know, the only work that makes this assumption is the one described in [6].

The DM approach (advocated by [7]) is more robust than the OM with respect to changes in the system [4] but operational statements are closer to executability. Attempts to build prototypes from deductive models have been reported in [8] and [9].

One way of executing DM is to translate them to semantically equivalent operative models, and we hope that this translation will be simplified by using an intermediate model that we will call Multiple States Operational Model (MSOM) or pseudo-deductive model , following Olive's lexicon (a more suited term would be Operational Deductive Model). MSOM supports a paradigm that can be used to specify IS in an elegant and powerful way.

We have implemented an automatic prototype generator (PROTOLOG/PG) based on the MSOM paradigm. It has been successfully applied to build non trivial prototypes automatically and verify the consistency of the source specifications. PROTOLOG/PG is built over a Relational DataBase System (implemented with research purposes) based on the Domain Oriented Relational Calculus approach and is operative in an IBM AT running PROLOG I.

This paper is organized as follows: first, using an inventory control and purchasing study case, formal concepts and specifications constructs of the PROTOLOG/PG Conceptual Scheme Facility are presented. Section 2 discusses about prototyping aspects of conceptual models specified in PROTOLOG. Finally, Section 3 presents conclusions and directions for further research.

1. THE PROTOLOG CSF.

As in the ISO proposal, we consider the Universe of Discourse at a specified time t -UoD(t) - as all those entities that exist at t. The knowledge about the UoD(t) is described in the Information Base at time t -IB(t) - and in its conceptual schema. We will call **current entities** the set of entities that exists in the UoD(t). An entity is current during a time interval. The largest possible time interval is the life span of the information system $[T_0,T_f]$.

Also the events, which produce changes in the UoD, take place at a specific time t, and we will say that they occur in a durationless time interval.

In conceptual models, IB(t) is decomposed into two disjoint sets: the Basis -B(t) - that contains all the elementary sentences of IB(t) and the Derived, -D(t) - that includes all those sentences that can be deduced by using the derivation rules [4].

The operational model paradigm associates entity sentences with the Basis,and describes the events in terms of operations on B(t). The derived set contains the entity sentences that can be deduced from other entity sentences.

In the deductive-operational models, as in the deductive ones , the Basis corresponds with the sentences related with the events, and the derived set corresponds with those entity sentences which can be deduced from the Basis .

Our conceptual model is based in a deductive-operational paradigm that is

González del Río, A., Alpuente, M., Casamayor, J.C., Pastor, M.A., Ramírez, M.J., Ramos, I. Protolog : A conceptual Schema Facility for Automated Prototype Generation. 7th. IASTED International Symposium. 1989.

formalized as an extension of the first order predicate calculus. Event classes and entity classes are expresed as predicate formulas. Constraints and derivation rules are axioms expressed in clausal form. We use the well known case study of inventory control and purchasing to introduce this components: there is a firm which sales and purchases products. The products have a given price that the firm can change. This firm would like to know: the quantity on hand of a given product (stock), the last month's sales of a product p (m-sales) and the sales made, the day before, for a product group (d-sales). The meaning of constraints appears as comments in the example. So we have four basic components to model the UoD. These components are:

i) Base predicates, which correspond to the event types, and define the structure of the event sentences. For example, in our case study: addprod (p,g,pr,t).

ii) Derived predicates , which are the intensional description of the entity sentences. For example: stock (p, q, t).

Both basic and derived predicates are syntactically constructed with variables ranging over predefined domains. These predicates have always an extra parameter that we call time-parameter which is the last one in the parameter list of every predicate. As previously said, the states of the IB are parametrized by the time. For every instant t of the system time life, IB(t) represents the set of current entities (and events) at t. These current entities at some instant t are currents at t'>t unless otherwise is specified. Thus entities are currents (or not) over time intervals that are explicitly defined. We consider these time intervals as sets of instants.

In the system there are built-in predicates that define temporal relations between intervals. These are the predicates **before** and **previous**, which are binary predicates. They have the following meaning: before (t_1, t_2) expresses the fact that time interval t_1 begins before time interval t_2. Previous is defined by:

previous (t_1, t_2) -> before (t_1, t_2) and
$\nexists t_3$ (before (t_1, t_3) and
before (t_3, t_2))

There are other built-in system predicates over time intervals. The newday (day,t) and newmonth(month,t), that are event-type predicates, define the beginning of a day(day,t) and of a month(month,t) respectivelly. These predicates allows to handle another built-in predicates as in-day(day,t), in-month(month,t)... They express that a given instant (or set of instants) t is contained in the set t1 of instants in wich day(day,t1), month(month,t)... are current.

iii) Constraint axioms, which are specified as action-oriented descriptions. We agree with Weigand [9] that this is a more natural way to express restrictions than the state-oriented descriptions. With every base predicate may be associated several constraint axioms. These axioms reflect the allowed changes in the UoD. No difference is made between dinamic and static constraints. The incorporation of time in the model allows an unified way of treating them. Moreover, in our model, dinamic constraints can reference anyone of the former states in the information base.

Constraints are expressed in clausal form. The left part of the clause is a base predicate. The right part is a collection of derived predicates and built-in predicates. For example the dinamic rule:

newprice (p,pr,t) -> $\exists$t1 (price(p,pr1,t1) and
previous(t1,t) and pr1<pr)

iv) Derivation rules, which are the mechanism used to deduce derived predicates. In a similar way as constraints, derivation rules are also expressed as action-oriented descriptions. Derivations are axioms in the first order theory that is used to specify a particular conceptual schema. Derivation rule axioms use two predefined metapredicates to model the fact that a given entity becomes current from a specific time t in the UoD. This fact can be expressed using the **Current** metapredicate. The **Not-Current** metapredicate has the opposite efect. This metapredicate states that a given entity (which was current from a previous time t until the moment) becomes not current in UoD (t) ,where t is the current time.

Derivation rules are expressed in clausal form. The left part of the clause is a base predicate. The right part is a collection of metapredicates over derived predicates, base predicates and built-in predicates. For example the rule:

addprod (p,g,pr,t) -> Current (product (p,g,t)) and
Current (stock (p,0,t)) and
Current (price (p,pr,t)) and
Current (M-sales (p,0,t))

means that when the event addprod occurs in UoD (t) then the entities product, stock, price and m-sales become current since t. All the variables in the axioms are unirversally quantified.

Below we give the specification for the inventory and purchasing case study.

DOMAINS
product : string
group : integer
quantity : integer
time : time-interval
price: integer
day: string
month:string

BASE PREDICATES
addprod (p:product,g:group,pr:price,t:time)
delprod (p:product,t:time)
addgroup(g:group,t:time)
sale (p:product,qs:quantity,t:time)
purchase (p:product,qp:quantity,t:time)
newprices (p:product,pr:price,t:time)
new-day (d:day,t:time)
new-month (m:month,t:time)

DERIVED PREDICATES
product (p:product,g:group,t:time)
price (p:product,pr:price,t:time)
stock (p:product,qs:quantity,t:time)
group (g:group,t:time)
month (m:month,t:time)
day (d:day,t:time)
M_sales(g:group,qm:quantity,t:time)
D_sales(p:product,qd:quantity,t:time)

CONSTRAINTS

addprod (p,g,t) -> $\exists$t1(addprod (p,g,t1) and $\nexists$t2 (delprod (p,g,t2) and
before (t1,t2) and before (t1,t)))
/* product p cannot be duplicate */

delprod (p,g,t) -> $\exists$t1 (addprod (p,g,t1) and $\exists$t2 (delprod (p,g,t2)
and before (t1,t2) and before (t1,t)))
/* only existing products can be deleted */

delprod (p,g,t) -> $\nexists$t1 (addprod (p,g,t1) and before (t1,t))
/* only existing products can be deleted */

addgroup(g,t) -> $\exists$t1 (addgroup(g,t1) and before (t1,t))
/* existing groups cannot be added */

sale (p,qs,t) -> $\exists$t1 (addprod (p,g,t1) and$\nexists$2 (delprod (p,g,t2) and
before (t1,t2) and before (t1,t)))
/* only existing products can be saled */

sale (p,qs,t) -> $\nexists$t1 (addprod (p,g,t1) and before (t1,t))
/* only existing products can be saled */

sale (p,qs,t) -> $\exists$t1 (stock (p,qt,t1) and previous (t1,t) and qs<qt)
/* quantity on hand must be positive */

purchase (p,qp,t) -> $\exists$t1 (addprod (p,g,t1) and $\exists$t2(delprod (p,g,t2)
and before (t1,t2) and before (t1,t)))
/*only existing products can be delivered */

purchase (p,qp,t) -> ∄t1 (addprod (p,g,t1) and before (t1,t))
/*only existing products can be delivered */

newprice (p,pr,t) -> ∃t1 (addprod (p,g,t1) and ∃t2 (delprod (p,g,t2)
and before (t1,t2) and before (t1,t)))
/*only existing products can be priced */

newprice (p,pr,t) -> ∄t1 (addprod (p,g,t1) and before (t1,t)))
/*only existing products can be priced */

newprice (p,pr,t) -> ∃t1 (price(p,pr1,t1) and previous(t1,t) and pr1<pr)
/* prices must not decrease */

DERIVATION RULES

addprod (p,g,pr,t)-> Current(product (p,g,t)) and Current(stock (p,0,t))and
Current (price (p,pr,t)) and Current (M-sales(p,0,t))

delprod (p,t) -> ∃g Not-Current (product (p,g,t)) and
∃qs Not-Current (stock (p,qs,t)) and
∃pr Not-Current (price (p,pr,t))

addgroup(g,t) -> Current(group (g,t)) and Current(d-sales(g,0,t))

sale (p,qs,t) -> ∃qt (Not-Current(stock(p,qt,t)) and
Current(stock(p,qt-qs,t))) and
∃qm (Not-Current(m-sales(p,qm,t)) and
∃pr (price (p,pr,t1) and previous(t1,t)) and
Current (m-sales (p, qm+pr*qs ,t)))) and
∃g (product (p,g,t2) and previous(t2,t)) and
∃qd (Not-Current (d-sales(g,qd,t)) and
Current (d-sales (g, qd+pr*qs,t))))

purchase(p,qp,t)-> ∃qt(Not-Current(stock(p,qt,t) and
Current(stock(p,qt+qp,t)))

new-prices(p,pr;t) -> ∃pr1 (Not-Current(price(p,pr1,t)) and
Current(price(p,pr,t)))

day(d,t)-> ∃qd,g(Not-Current(d-sales(g,qd,t))and Current(d-sales(g,0,t)))

month(m,t)-> ∃qm,g(Not-Current(m-sales(g,qm,t)) and
Current(m-sales(g,0,t)))

2. PROTOLOG/PG.

2.1. FROM SPECIFICATION TO IMPLEMENTATION

We will present how a PROTOLOG model can be designed and translated into an executable prototype. Prolog has been used as the implementation language for both the PROTOLOG/PG system (an automatic prototype generator that has as input the high level specifications written in PROTOLOG) and the generated prototypes. The main requirements of the system were:

- »» Assisting the specifier in the Specification Design.
- »» Automatically generating the prototypes with no programmer intervention. The prototype would have all the functions of the final system, althought might be less efficient.
- »» Capability to embody optimization tools.
- »» The system would allow the process of refining specifications and generating new prototypes.

1. The time formulation

In the beginning of the system time life T_0, none of the entities in $UoD(T_0)$ is current. Afterwards, because of the event ocurrences, entities will become current.

At the conceptual level, derivation rules define when entities become current or not current. At the implementation level, the time-parameter is a time-interval. Its lower bound is defined by the application of the **current** metapredicate. The upper bound is defined by application of the **not-current** metapredicate. The IB, as a model of the specification, is a set of predicate sentences parametrized by time-intervals. All those entities that are current at some instant t will be represented in IB(t) as predicate sentences which will have an undefined upper bound in its time-interval parameter. As an implementation detail, this undefined value is in fact the upper bound of the system time life T_f.

Let us give a more detailed explanation of the time-interval handling in the system.: **current** $(p(a_1,a_2 \ldots a_{n-1},t_{current}))$ incorporates the predicate p to the Information Base with an associated time-interval $[t_{current},t_{final}]$. And **not-current** $(p(a_1,a_2 \ldots a_{n-1},t_{current}))$ closes the validity interval associated with the predicate p. Thus predicate p will have an associated interval $[t_{initial}, t_{current}]$, where $t_{initial}$ is the time when the predicate became current.

Using these two metapredicates, the derivation rules become :

'purchase(p,q_p,t)<--> not-current (stock(p,q_s,t)) and
q is q_s+q_p and current(stock(p,q,t)).

One special feature of the base predicates is the durationless interval associated with them, i.e. purchase(p,q,t) occurs over an interval [t,t], where t is the time when the event happens.

2) Constraints syntax :

In [9] it is stated that Prolog can test the integrity rules but it cannot enforce them, since every rule should be checked after each update.

When an event occurs in the Universe of Discourse, and its incorporation to the Information Base is implied by an input requirement, its associated constraints try to be satisfied. A constraint is a Prolog-like clause whose satisfaction assumes that the incorporation of the predicate should violate the integrity of the base. For every action, the associated constraints are introduced in a negated form and, if true, a violation message is issued. If the constraints are not violated, no message appears and a derivation rule is applied. For example:

newprice (p, pr, t) <- ∃t1 (price (p, pr1, t1) and
previous(t1,t) and pr1>pr) and
message (' PRICES CANNOT BE DECREASED ').

The PROTOLOG syntax foresees that one error must be associated with each constraint in the model specification .

2.2. SYSTEM ARCHITECTURE

PROTOLOG/PG has two components: an Analyzer and a Generator. The analyzer performs the syntactic analysis of the specification, in interaction with the designer. When the design is finished, it is saved in the specified file, if it is correct. The Generator first deals with automatic generation of a menu, in agreement with the requirements. It defines the system database and automatically generates the prototype. Domains, attributes, predicates, relations and constraints must be defined in the database. The prototype is an executable program written in Prolog, which loops for each Life Cycle instant and can be stored to be later consulted and executed. The system works by pulses, one of them for each instant of the life period.

We have a conversational system that can be controlled by menus.The implemented options allow the management of a Library of specifications . The following options are allowed : designing and saving a specification, generating and running a prototype, going into the I/O menu...

The I/O menu allows:

»» The selection of events that occur at the current instant.
»» The reception of the associated information:
» The verification of constraints and, if one is not satisfied,the rejection of the input.
» Derivation and keeping all the derived information in the base.
»» The selection of outputs: the corresponding information is obtained from the base and supplied to the user.

3. CONCLUSIONS.

- We have defined and implemented a CSF (PROTOLOG). It allows the specifiers to build formal specifications.We have seen that these formal specifications are closer to their conceptualizations of the UoD specified.This property (closeness),makes specifications easier to read,understand and design.

- Another success is the ability to translate PROTOLOG models into an executable prototype automatically. However, the counterpart of the model is the non-locality of the information. In order to make incremental designing easier, this problem must be investigated.

- As PROTOLOG allows the definition of derivation rules, our work explores the possibility, outlined in [4], of building pseudo-deductive models with operational languages.This fact could lead to the missing link of "transforming a deductive model into a single state operational one ". And this aspect is the designer's main problem in the task of implementing deductive models .It also affects the System Design phase in the I. S. Life Cycle .

- We have shown the usefulness of Prolog as a tool for conceptual modelling. This capability of Prolog has been questioned in some recent papers ([9]) by arguments that, in our opinion, are erroneous.

- In most models references to *any* past state of the IB are required, either in constraints, derivation rules or *queries*. We have proved that the operational approach is adequated to deal with historical information, if a temporal dimension is associated with the predicates.

- PROTOLOG/PG is not a mock-up. The practical application of the system to real projects , indicates that this system is useful and operational.

REFERENCES

[1] ISO/TC9/SC5/WG3 "Concepts and terminology for the conceptual schema and the Information Base" Report n 695 . 1982.

[2] Mylopoulos, J.; Bernstein, P.; Wong, H., "A language facility for designing Interactive Database intensive" Applications ACM_TODS, vol 5, n 2, 1980,pp. 185-207.

[3] Borgida, A.; Mylopoulos, J., "Generalization/Specialization as a Basis for Software Specification", "On conceptual modelling", Springer Verlag, 1984, pp.191-283.

[4] Olive, A., "A comparison of the operational and deductive approaches to conceptual information system modelling", Proc. IFIP 86, pp. 91-96. North Holland Elsevier, 1986.

[5] Sistac, J., "Construcció automatica de prototipus de sistemes d'informació, a partir de models conceptuals deductius, descrits amb el llenguatge DADES", Tesis Doctoral, U.P.B. Barcelona. 1987.

[6] Feather, M., "Mapping for Rapid Prototyping", ACM Sigsoft SCN, vol 7, n 5, 1982, pp. 17-24.

[7] Bubenko, J.A., "On concepts and strategies for requirements and information analysis", SYSLAB report, n 4, 1981. Chalmers University of Technology Goteborg, Sweden.

[8] Sistac, J.; Olive, A., "DADES/GP: A prototype generator from deductive conceptual models of Information Systems", Convention Informatique 86, Paris, 1986.

[9] Weigand, H., "Conceptual models in Prolog", Proc. of the Database Semantics, vol DS-1, 1985 , pp. 7-11.

[10] Gallaire, N.; Minker, J.; Nicolas, J.M., "Logic and Database: A deductive approach", Comp. Surveys, vol 10, n 2, 1984.

[11] Reiter, R., "Towards a logical reconstruction of relational database theory ", in "Logic and Databases", Gallaire and Minker (Ed). Plenum Press, 1984, pp. 191-238.

A Conceptual Scheme Specification System for Rapid Prototyping.

I. Ramos*, J. H. Canós*, R. Forradellas*, J. Oliver**.

* Departamento de Sistemas Informáticos y Computación.
(DSIC-UPV) Universidad Politécnica de Valencia.
Camino de Vera s/n. 46020 Valencia (Spain).

** Departamento de Sistemas Lógicos.
Universidad de Castilla-La Mancha.
Carretera de las Peñas Km. 3,2 02049 Albacete (Spain).

Abstract

This paper presents Functional MicroOBLOG (FMOL), an Object-Oriented Database Conceptual Scheme specification language, as a powerful tool for software development, based on the automation paradigm [1]. FMOL introduces the formal specification techniques in the Information Systems field. The foundations are the Abstract Object Type (AOT) theory [2]. With this concept, we obtain finer approaches to reality, and we solve the modelling problems in a more natural way.

The Conceptual Scheme Specification System (CSSS) FMOL produces automatically prototypes from Conceptual Scheme Specifications in a user-friendly environment.

Introduction.

Prototyping transforms the classical software development lifecycle in order to reduce the high maintenance costs derived from poor software requirements specifications. An alternative approach to this lifecycle is the automation-based paradigm, that uses two main principles to develop software in an efficient way:

- formal specification of problems, using logic and algebraic formalisms.
- rapid prototyping through compilation of formal specifications.

Logical programming [7] offers powerful and easier-to-understand languages. More precisely, functional programming is the basis of so-called specification languages that, using the concept of abstract data type (ADT) and term rewriting, are the formal tool to build executable specifications (by equational deduction).

On the other hand, it is important the influence that the object-oriented (OO) paradigm is taking in the Information Systems field as a finer approach to model the reality. More precisely, the view of the Universe of Discourse (UoD) as a society of interacting objects that interact via event sharing, is crucial in the later development of any conceptual scheme (CS) modelling.

The specification languages mentioned above remain a little bit distant from the Information Systems OO view, and they are ambiguous many times because they do not distinguish between events and objects' attributes; indeed, the concept of object (ADT + local hidden state) has no place inside them.

A. Sernadas (Op. cit.) proposes as an alternative to this lake of expresiveness: the integration of object-oriented and functional paradigms in order to obtain algebraic and OO DBCS specifications, using the AOT logic as an alternative to the 'classic' ADT.

FMOL language takes this approach and appears as a mean to express precisely state and time aspects on the DB field. These aspects are not supported by relational logic.

A first version of a CS Specification environment has been developed at the DSIC-UPV, written in BIM_Prolog and C, on a Sun 3/60 machine.

The system is composed by a syntax driven editor, a translator that transforms FMOL CS to Axis specifications, and a prototyper, all of them integrated in a user-friendly enviroment.

Using Axis (Operational semantics by term rewriting) [4] as the operational interpreter that allows rapid prototyping of CS specifications.

1. The Language.

The Functional MicroOBLOG specification language (FMOL), has been designed for the description of Conceptual Scheme in Object Oriented Data Base (OODB) specifications, taking an algebraic and object oriented perspective [5].

Based in OBLOG [3], where an object encapsulates Data, Memory, Control and Interacting Mechanisms.

An object is a 4-tuple:

$$(X, A, \Omega, \alpha)$$

where: **X**: is the events set.

A: is the attributes set.

$\Omega \subset X^*$: is the lifecycle set, equivalent to the admisible event sequences.

$\alpha: X^* \dashrightarrow obs(A)$: is the observability function, using it, we will obtain the attribute values as a function of the event sequences.

For a description of the language component let us see the conceptual scheme for a Library System in the following Library.fmol file:

Ramos I., Canós, J.H., Forradellas, R., Oliver, J. A conceptual schema specification system for rapid prototyping. 8th IASTED International Symposium. Austria. Feb. 1990.

```
conceptual_scheme library;
 domains  nat(0), bool(false), string(" ");
 class book;
  constant_attributes
          code: string key;
          title: string
  variable_attributes
          available(b:book):bool;
    equations   r: reader
      available(b)=false if T=T1'loan(b,r);
      available(b)=true if T=T1'return(b,r)
      available(b)=true if T=T'newbook(b)
                        else available(b) at T1;
    end_equations;
          isoflibrary(b:book):bool;
    equations
      isoflibrary(b)=true if T=T1'newbook(b)
                        else isoflibrary(b) at T1
    end_equations;
   private_events
          newbook(b:book)
   shared_events (reader)
         loan(b:book,r:reader);
         return(b:book,r:reader)
 end_class;
 class reader;
  constant_attributes
          readern: nat key;
          name: string
  variable_attributes
          isareader(r:reader):bool;
    equations
      isareader(r)=true if T=T1'newreader(r)
                        else isareader(r) at T1;
      isareader(r)=false if T=T1'delreader(r)
                        else isareader(r) at T1
    end_equations;
          bookno(r:reader):nat;
    equations n:nat b:book
       bookno(r)=n if T=T1'loan(b,r) and
                        bookno(r)=n-1 at T1;
       bookno(r)=n if T=T1'return(b,r) and
                        bookno(r)=n+1 at T1
    end_equations;
   private_events
          newreader(r:reader);
          delreader(r:reader)
   shared_events (book)
         loan(b:book,r:reader);
         return(b:book,r:reader)
 end_class;
 relationship hasbook (b:book r:reader)
  constant_attributes
          date:string;
          number:nat
  variable_attributes
          hasbook(b:book,r:reader):bool;
    equations
      hasbook(b,r)=true if T=T1'loan(b,r);
      hasbook(b,r)=false if T=T1'return(b,r)
                        else hasbook(b,r) at T1
    end_equations;
   private_events
          loan(b:book,r:reader) new;
          return(b:book,r:reader) destroy;
 end_relationship;
 traces  T,T1:string(events);r:reader;b:book;
   T=T1'newreader(r) if not isareader(r);
   T=T1'delreader(r) if isareader(r)
                        and bookno(r)=0;
   T=T1'newbook(b) if not isoflibrary(b);
   T=T1'loan(b,r) if isareader(r) and
                        isoflibrary(b) and
                        vailable(b) and
                        bookno(r)<5;
   T=T1'return(b,r) if hasbook(b,r);
   T=T1 otherwise
 end_traces;
end_conceptual_scheme.
```

LIBRARY.FMOL FILE

The example shows three object types:

- Primitive Objects: are ADT without state.

in the example: **DOMAINS** (nat, bool, string)

- Elementary Objects: are AOT with state, persistence,...

in the example: **CLASS** (book)

- Complex Objects: builded using elementary objects.

in the example: **RELATIONSHIP** (hasbook)

At the moment it is possible to define the Conceptual Scheme as a society of interacting objects (UoD). This society is a complex object whose members are interelationated components.

A specification in Functional MicroOBLOG has two parts:

1.-the DATA subspecification part (**DOMAINS**), the data types (ADT) are defined here and they are used in the conceptual scheme specification.

2.-the OBJECT subspecification part (**CLASS, RELATIONSHIP**), here, we define the UoD as an object universe (AOT). It has two parts with respectively:

- structural aspects:

KEY mechanisms, **CONSTANS ATTRIBUTES**

and

- behavior aspects:

EVENTS and **TRACES**

Returning to the example a **CLASS** has the following elements:

a) an identifier (Book, Reader, ...)

b) **ATTRIBUTES** represent the local state at each time instant, according to each one of the class intances. There are two sorts:

CONSTANT ATTRIBUTES: with a value given at the instance creation.

VARIABLE ATTRIBUTES: they depend on the system state in an axiomatic way (via **EQUATIONS**).

c) **EVENTS**. They produce modifications in the variable attributes values. There are two sorts:

PRIVATE EVENTS: they affect the variable attributes of just one class.

SHARED EVENTS: implement the interaction between classes (using event sharing).

The **RELATIONSHIP** part, specifies complex objects and the interaction between shared events. It has the same elements that **CLASS** has.

The last element is **TRACES**: allowed event sequences producing the general system state.

2. The Translator.

Now, we will describe the process followed by the translator, to translate source code (written in FMOL) to object code (in Axis).

First, we describe briefly the characteristics of the object enviroment Axis: it is formed by a compiler, a term rewriting interpreter and a specification database. Axis has a language for specifiying abstract datatypes [4] using first order equations; they are used to define the required behaviour. It was designed at the Information Systems Centre of HP Labs.

Ihe Axis specifications may be oriented as rewriting rules used to reduce terms to their canonical form. It is similar to OBJ2 [6].

The mathematical semantic is given using many sorted algebras, and the operational semantic is implemented by conditional term rewriting. Axis specifications are modular and they include parameterisation, renaming, summation and instantiation of modules and allow sharing or copying of subspecifications.

An Axis specification includes a signature (set of sorts and operators).

Each operator is associated with a sequence of sorts that represent the domain and range sorts of its arguments and result.

A presentation is a signature together with some equations on that signature. A signature comprises some sorts and operators and provides vocabulary for writing specifications.

In Axis, there are three kinds of modules:

- **Parameterised SPEC**
- **Unparameterised SPEC**
- **PROPS**

SPEC modules define particular algebras, whereas PROPS modules define classes of algebras.

The keywords SPEC and ENDSPEC, delimite SPEC modules. The SPEC keyword is followed by an identifier naming the module, and two modules must not have the same identifier.

Syntactically, the body of simple SPEC and PROPS modules are the same:

SORTS <identifiers>
OPS <op. declarations>
FORALL <variable declarations>
AXIOMS <axioms>

SPEC modules are used to specify particular data types and each one denotes the initial algebra in the classes of algebras satisfaying the axioms. The modules: Boolean, Natural and Identifier are built-in in AXIS. We don't need all the expresivenes of the object language for implementing our translation satisfactorily.

We establish a syntactic correspondence between the diferent aspects of both languages.

The keyword CONCEPTUAL_SCHEME and END_CONCEPTUAL_SCHEME are substitued during the translation by SPEC and ENDSPEC.

The equivalence between FMOL and Axis appears in the following table:

SOURCE LANGUAGE	OBJECT LANGUAGE
conceptual_scheme E1	SPEC E1
class C1,C2...	SORTS E1,C1,C2...
relationship R1,R2...	OPS R1,R2...
constant_attributes CA...	OPS CA...
variable_attributes VA...	OPS VA...
equations	FORALL
x:T1	x:T1
y:T2	y:T2
...	...
eq1;eq2...	AXIOMS: for VA...
private_events PE1,PE2...	OPS PE1,PE2...
shared_events SE1,SE2...	OPS SE1,SE2...
preconditions (for **traces**)	OPS pc1, pc2...
end_conceptual_scheme	ENDSPEC

The object code in Axis is the following library.ax file:

```
   SPEC library
SORTS library book reader
OPS      nil: -> library
         available: book library -> bool
         isoflibrary: book library -> bool
         _: id ->book
         newbook: book library -> library
         _:nat ->reader
         isareader: reader library -> bool
         bookno: reader library -> nat
         newreader: reader library -> library
         delreader: reader library -> library
         loan: book reader library -> library
         return: book reader library -> library
         hasbook: book reader library -> bool
         pcnewreader: reader library -> bool
         pcdelreader: reader library -> bool
         pcnewbook: book library -> bool
         pcloan: book reader library -> bool
         pcreturn: book reader library -> bool
FORALL   x,y: book
         s,t: reader
         l1,l2: library
AXIOMS for pcnewreader:
         pcnewreader(s,l1)=NOT isareader(s l1)
AXIOMS for pcdelreader:
         pcdelreader(s,l1)=(isareader(s,l1))
                           AND (bookno(s,l1)==0)
AXIOMS for pcnewbook:
         pcnewbook(x,l1)=NOT isoflibrary(x,l1)
AXIOMS for pcloan:
         pcloan(x,s,l1)=(isareader(s,l1)) AND
                        (isoflibrary(x,l1)) AND
         (available(x,l1)) AND (bookno(s,l1)<5)
AXIOMS for pcreturn:
         pcreturn(x,s,l1)=hasbook(x,s,l1)
AXIOMS for newbook:
         newbook(x,l1)=l1 IF NOT pcnewbook(x,l1)
AXIOMS for newreader:
    newreader(s,l1)=l1 IF NOT pcnewreader(s,l1)
AXIOMS for delreader:
    delreader(s,l1)=l1 IF NOT pcdelreader(s,l1)
AXIOMS for available:
         available(x,nil)=false
         available(x,loan(y,s,l1))=
           IF (pcloan(y,s,l1)) AND (y==x)
           THEN false ELSE available(x,l1) ENDIF
         available(x,newbook(y,l1))=
           IF pcnewbook(y,l1) AND (y==x)
           THEN true ELSE available(x l1) ENDIF
         available(x,return(y,t,l1))=
           IF (pcreturn(y,t,l1)) AND (y==x)
           THEN true ELSE available(x,l1) ENDIF
    available(x,newreader(s,l1))=available(x,l1)
    available(x,delreader(s,l1))=available(x,l1)
AXIOMS for isoflibrary:
    isoflibrary(x,nil)=false
    isoflibrary(x,newbook(y,l1))=IF (pcnewbook(y,l1))
    AND (y==x) THEN true ELSE isoflibrary(x,l1) ENDIF
    isoflibrary(x,newreader(s,l1))=isoflibrary(x,l1)
    isoflibrary(x,delreader(s,l1))=isoflibrary(x,l1)
    isoflibrary(x,loan(y,s,l1))=isoflibrary(x,l1)
    isoflibrary(x,return(y,s,l1))=isoflibrary(x,l1)
AXIOMS for isareader:
    isareader(s,nil)=false
    isareader(s,newreader(t,l1))=IF (pcnewreader(t,l1))
    AND (t==s) THEN true ELSE isareader(s,l1) ENDIF
    isareader(s,delreader(t,l1))=IF (pcdelreader(t,l1))
    AND (t==s) THEN false ELSE isareader(s,l1) ENDIF
    isareader(s,newbook(x,l1))=isareader(s,l1)
    isareader(s,loan(x,t,l1))=isareader(s,l1)
    isareader(s,return(x,t,l1))=isareader(s l1)
AXIOMS for bookno:
  bookno(s,nil)=0
  bookno(s,loan(x,t,l1))= IF pcloan(x,t,l1) AND (s==t)
  THEN (bookno(s,l1)+1) ELSE bookno(s,l1) ENDIF
  bookno(s,return(x,t,l1))= IF pcreturn(x,t,l1)
  AND (s==t) THEN bookno(s,l1)-1 ELSE bookno(s,l1) ENDIF
  bookno(s,newreader(t,l1))=bookno(s,l1)
  bookno(s,delreader(t,l1))=bookno(s,l1)
  bookno(s,newbook(x,l1))=bookno(s l1)
AXIOMS for loan:
  loan(x,s,l1)=l1 IF NOT pcloan(x,s,l1)
  loan(x,s,loan(y,t,l1))=loan(y,t,loan(x s,l1))
    IF pcloan(x,s,loan(y,t,l1))
    AND pcloan(y,t,l1) AND (NOT x==y)
AXIOMS for return:
  return(x,s,l1)=l1 IF NOT pcreturn(x,s,l1)
  return(x,s,return(y,t,l1))=return(y,t,return(x,s,l1))
   IF pcreturn(x,s,return(y,t,l1)) AND pcreturn(y,t,l1)
   AND (NOT x==y)
AXIOMS for hasbook:
  hasbook(x,s,nil)=false
  hasbook(x,s,loan(y,t,l1))=IF pcloan(y,t,l1)
    AND (x==y) AND (s==t)  THEN true
    ELSE hasbook(x,s,l1) ENDIF
  hasbook(x,s,return(y,t,l1))=IF pcreturn(y,t,l1)
     AND (x==y) AND (s==t) THEN false
     ELSE hasbook(x,s,l1) ENDIF
  hasbook(x,s,newbook(y,l1))=hasbook(x,s,l1)
  hasbook(x,s,newreader(t,l1))=hasbook(x,s,l1)
  hasbook(x,s,delreader(s,l1))=hasbook(x,s,l1)
ENDSPEC
```

LIBRARY.AX FILE

3. The Prototyper.

The Prototyper executes the specifications. With this purpose, it relies on the Axis term rewriting system, interchanging messages through an Unix™ pipe. The Prototyper covers two different aspects:

- class and instance mechanisms
- CS animation

CS animation is carried-out executing sequences of admisible events that are represented as system traces (terms that are used to observe the system state at every moment). Only the events for wich preconditions hold will be incorporated in the trace. The Prototyper will ask the term rewriting system for the value of these preconditions in the state represented by the current trace. A possible system state (a trace) for our example may be the following:

loan('Hamlet','William',
newreader('William',newbook('Hamlet',nil)))

where **nil** is a constant that denotes the initial system state.

The prototyper has two overloaded operators: **new** and **destroy**, they create and destroy class' instances respectively. It also has the boolean operator **exists**, it computes the existence or not of the class' instances at any moment.

When an instance is created (using new), the system asks the user for the values of the constant attributes.

It is also possible at every moment to consult the variable attributes values of the instances(the state); these values are obtained from the system trace and the axioms used as rewriting rules.

The Prototyper performs as a menu-driven program, and the main procedures offered are:

> work with classes
>>new
>>destroy
>>exists
>>Carrier (set of actual class' instances)

> work with relationships
>>relationship value
>>Carrier (relationship extension)

> event occurrence (given in a menu driven way)

In our working enviroment, the menu permits to select an existent file with a FMOL specification created at once or we can create, using the oriented graphic editor, a new specification in another file.

If, afterwards, choosing between the options of our menu we choose to perform the specification, the prototyper will be a interface between the Axis system and the user. Automaticaly, it will generate the chains wich, by rewriting, will reproduce the evolution of the system. Moreover, we can save the current situation for, in a new working session, to begin by it.

Although relationships are also objects, their instances are created when shared events hapen. At this moment, values for the constant attributes (if any) are defined.

4. Conclussions.

We have presented FMOL as a powerful tool to CSS. The use of FMOL taking an information systems point of view is very advantageous, with respect to the ADT specification languages, because FMOL reflects reality in a more direct way throug the OO approach (the semantic gap is narrow).

Future versions of FMOL system will enrich the language in order to dispose of:

- class inheritance mechanisms, class operatores: specialitation, tupling ...
- a way to express integrity constraints using modal temporal logic.
- a graphic environment for the edition and execution of CSS.
- a OODB interface, using a Object Oriented query language.

At the moment it is full operational with the explained perfomances.

References

[1]: Baltzer, R. et al.:'Software Development', IEEE Computer Science Press, 1989.

[2]: Sernadas, A. et al.: 'Abstract Objet Types: a temporal perspective', Coloquium on Temporal Logic and Specification, Springer-Verlag, 1989.

[3]: Sernadas, A. et al.:'Object-Oriented Specifications of Databases: on algebraic approach', VLDB, 1987.

[4]: Coleman, D. et al.: 'The Axis Specification Language'. Tecnical Report HPL-BRC-TM-88-020, Software Engineering Dept. Hewlewtt-Packard Laboratories, Bristol, U.K., 1988.

[5]: Ramos, I et al.: 'Object Oriented Specification and Prototyping of Database Conceptual Schemes', Invited Coference Congreso Nacional de Informática, Querétaro, Mexico 1989.

[6]: Futatsugi, K. et al.: Principles of OBJ2. In Symposium of Principles of Programming Languages, pages 52-66, ACM 1985.

[7]: Lloyd, J. W.: 'Foundations of Logic Programming' Second Extended Edition, Springer-Verlag, 1987.

Logics and OO-Data Bases:a declarative approach

I.Ramos

Dept.Information Systems and Computation
Valencia University of Technology
46020 Valencia (Spain)

ABSTRACT

Three perspectives are adopted in the paper to define the Object Oriented (OO) Paradigm:the General System Theory principles adapted to Information Systems (IS)[1], the Abstract Object Types (AOT) or Objectual Logic (OBLOG)[2] and clausal or equational logics. In this approach the OO-Data Base (OO-DB) specification (the system view or Conceptual Scheme (CS)) is a formal theory in this blend of logics. Its model is the OO-DB extension. Two Object Definition Languages (ODL) are implemented: FMOL and RMOL. Two first order languages used as User Languages (UL) are associated with them. The operational semantics of the underlaying logic (implemented by deduction) together with some software sugar constitutes an OO-DB Management System.

1. INTRODUCTION

One of the tasks of the IS Analyst is to specify its system view of the object system in order to produce the Conceptual Model specification. Several models (relational, deductive, OO...)and different linguistic frameworks are used to perform this task. Historically the IS models have been associated with the data models. In the relational model the IS is viewed as a set of entities with attributes, relationships etc. in the Deductive Data Base (DDB) model as a first order theory. In both models nothing is said (inside the model) about the behavioral manner of the system.

In the OO-DB model the IS is viewed as a collection of interacting objects. Each object is a self-contained operational unit (the system behavior is specified at that level). A unique model is used for specifying structural and behavorial aspects; the semantic gap is also narrower than in the other two cases.

Once the system view and the model are fixed, a good linguistic framework is required to specify the IS Conceptual Model. What is good depends on subjective criteria but the following characteristics of the specification language are widely accepted: precise(formal), declarative (not imperative), executable (allowing prototyping)

We will use a blend of AOT (that captures the descriptive concepts of the IFIP WG8.1 approach in a precise way) and clausal logic in RMOL or equational logic in FMOL as the linguistic framework.

In Section 2 IFIP's descriptive concepts are presented.

To be precise: the language RMOL/FMOL is a blend at the semantic level of AOT and clausal/equational logic used as ODL to specify Open Passive IS (OPIS) following the OO model.

The underlying data model is:the Conceptual Model specification is a theory in this blend of logics. It has the OO-DB extension as model (declarative semantics) and the deductive mechanism as the operational semantics.

In RMOL/FMOL the OO-DB is implemented over the standard/initial model of the clausal/ equational theory which is equivalent to the IS specification. The operational semantics is implemented by using SLDNF-Resolution/ Conditional Narrowing. If an OPIS specification is equivalent to a theory (we will see how this theory is built in Section 4), we can use its first order associated language as UL of the OO_DB. We define the ULs in Section 3 , with its ODL, and give some examples.We build an OO-DBMS by adding software sugar. Finally in Section 5 we make some concluding remarks and comment on future extensions of our work.

A. M. Tjoa et al. (eds.), *Database and Expert Systems Applications*
© Springer-Verlag/Wien 1990
Ramos, I. Logics and OO-Data Bases: a declarative approach. Proc. International Conference on Database and Expert Systems Applications (DEXA 90), 1990.

2. THE OO-MODEL

Let us give the descriptive concepts we use on the OO model following IFIP approach.
An object is a self contained unit. Which will have properties (attributes).
It will have an state that is the state of the set of its attibutes. When an object is created, it is brought into existence and when it is destroyed, it finishes its existence(life). Each object has a name that identifies it during its existence. An object has static/ dynamic aspects in the sense that it does not change/ changes its state over time. An object may be active/passive if it is/is not involved as necessary for a change of state to take place.The change of state concept is a very important one. The creation/destruction of the object are strong cases of state changes. Weak changes occur when there is a change of state in an existing object which is not destroyed. We call process the cause of a change of state of an object. It is limited in time. It has elementary parts responsible for a single unified change of state that we call events. An event is an abstraction of a change of state in the object system. It is discrete, has no duration and occurs at a certain point in time. The event concept is relative to the abstraction level. We call behavior the dynamic manner of an object. The object exists during a temporal period, its life, that is a limited coherent amount of time. It has a duration. Events happen in a certain point of time (a period having no duration).
By means of the abstraction mechanisms we put together objects in types.We call class a set of properties all of which characterize certain objects. A class is a precise characterization of structural and behavioral properties which a collection(or type) of objects all share [3].
The operator instance-of works on a class and yields a member of the corresponding type as the result.The operator population-of yields the type for a class. We may have classes and subclasses denoting types and subtypes.Subtype populations are subsets of the supertypes populations. Subclasses are defined by adding more properties to the superclasses (specialization). We define generalization/specialization as relations between classes and subclasses at the same abstraction level.

3. THE LINGUISTIC FRAMEWORK

Let us present our linguistic paradigm following the AOT approach and adapted to the previous descriptive concepts.
We will have a specification level of the Conceptual Scheme where RMOL/FMOL are used(they are the ODLs) and a user level with its corresponding ULs.The two levels share the same logic paradigm.

The object definition language (ODL)

An object class may be formalized as a 4-ple **{X,A,T,ob}** where **X** is the event set and **A** is the set of typed (classed) attributes.Their values at a given instant are the object state.When an attribute has no assigned value (at creation time for example) it takes the main constant of the type as the default value [4].**T** is the set of allowed lives (processes or traces) for the objects of the class.**ob** is the observation function.The **ob** domain is **T** and the codomain is the set of pairs {attribute,value}. We can say that the objects are observable processes.
Let us see an example. If the system domain is a library where books are loaned to readers, they are returned after a period... and certain rules need to be respected like " it is not possible to loan a book which is not available", etc. We can specify the class of the books in the following way.

```
conceptual_scheme library;
 domains  nat(0), bool(false), string(" ");

 class book;

       constant_attributes
              code: string key;
              title: string

       variable_attributes
              available(b:book):bool;

                equations   r: reader
                    available(b)=True if T=T1'newbook(b)
                                  else available(b) at T1;
                    available(b)=False if T=T1'loan(b,r);
                    available(b)=True if T=T1'return(b,r)
                end_equations;

              is_of_library(b:book):bool;

                equations
                   is_of_library(b)=True if T=T1'newbook(b)
                                  else is_of_library(b) at T1
                end_equations;

       private_events
              newbook(b:book)

       shared_events (reader)
              loan(b:book,r:reader);
              return(b:book,r:reader)
end_class; ...
```

In the example we easily identify the attributes and the events. We use preconditions to specify when an event is relevant. In this case it is concatenated to the old trace T1 to give the new one T: T=T1.newbook(b) **if** not-is-of-library(b). With trivial meaning if b is a book.Other events have similar treatment. Relevant events will change the state of the variable atributes according to **ob**. In the example: available(b)= **true if** T=T1.newbook(b) **else** available (b) **at** T1. The function **ob** is given axiomatically by using equational logic. When we use FMOL.Other classes will be specified in a similar manner, like **class** reader, etc. and combined by using the class operators for building complex classes. This process continues until we have completely specified the IS(the Conceptual Model will be the class specifyng the object system). Class instances are named by using object surrogates (in the example strings) following the key mechanism used [4].

A class may not contain events and variable attributes. It is a primitive class.Its type is an Abstract Data Type (ADT). Primitive classes denote the data subspecification. They are used as object surrogates and attribute classes.Their instances always exist, they are not created nor destroyed and they have not a changing state. For example: **class** Nat,Bool with obvious types {0,1,2,...},{true,false} If a class has events and variable attributes and is not built by using class operators,it is an elementary class like the class book. We call complex classes those defined by using class operators. At the moment we have only one kind of complex classes in our ODLs that we call **relationships**.

Objects interact by sharing events.A shared event (as a loan) for class book and reader(not specifyed) is a private event for the relationship has-book. Shared events will participate in the lives of the objects with the same name but different arity. An instance will be the pair {class,surrogate}.i.e:{book,"b1"}.

In this model an IS will be specified as a composition of interacting classes that is a class. Its type is a Society of interacting objects modelling the IS. This way of modelling allows a constructive and interactive manner of specifying IS.

Let us see how the system supports the OO principles: encapsulation is guaranteed by the fact that objects are only acceded by using their attributes (to observe their states) or their events (to change them). Attributes and events are two well defined interfaces of acceding to objects.

Objects instances persist in time changing their state. They are the OO-DB extension that is in the main memory when it is used and in the back memory when is idle.

Independence: the CM is specified in a very abstract way independently from any concrete representation of data or requirements.

Homogeneity:the same declarative model is used at any level. The underlying "data model" is always the same.

Communication: objects communicate by sharing events. More complex ways of communication will be introduced when we add OA (Open Active) IS facilities.

Inheritance is implemented by algebraic enrichment in FMOL and by adding the new properties in RMOL.

We have other "non-standard" characteristics like the class operators which allow the definition of abstraction mechanisms like agregation/part-of (by tupling/projecting) and others which are easily implementable in our model. We also have a deductive way of defining complex classes which will open the possibility of having Deductive OO-DB in a similar way to the DDB field[5].

User languages (ULs)

We will define the associated UL to RMOL in an inductive way:

syntax of the UL

Terms of primitive class t are:

i.An instance of class t
ii.A variable of class t
iii.Only i,ii are **Terms**

Terms-a of primitive class t are:

i.If **a** is an attribute of class t,**C** is a class having **a** as attribute and **kc** as the class of the key attributes and **ikc** is one of their instances, then **a(ikc)** is a **Terms-a** of classe t.
i.e: available("b1")
ii.Only i. are **Terms-a**

Atom

i.If C is a class with constant attributes a1..ar and t1..tr are **Terms** then **C(t1,...,tr)** is an **Atom**.
Nat(0),book("b1","Hamlet")

ii.If f,g are **Terms** or **Terms-a** of the same class then f**or**g is an **Atom**.
or are some relational operators:
no-of-books(reader("r1"))<6

iii.If e is an event with arity class **ae** and **iae** is one of its instances then **e(iae)** is an **Atom**:loan("b1","r1")

iv.Only i,ii,iii are **Atom**

wff

i.An **Atom** is a **wff**

ii.If A,B are **wff** then **not** A A,B A;B A<-B are **wff**

iii.Only i,ii are **wff**

semantics of the UL

We take the object surrogate types and the types of the primitive classes as the domain of the interpretation. In RMOL only two constructions need auxiliary interpretation: -let **lambda** be the semantic denotation function,C(v1,...,vr) an instance of a non-primitive class C then
lambda(C(v1,...,vr))=true **if** exists (C(v1,...vr)) **else** false
-let be e(v1,...,vn) an instance of the event e then
lambda(e(v1,...,vn))=true **if**
happens(e(v1...)) **and** pre-cond(e(...))=true
else false

exists is defined in the following way:
exists(C(...)):-new(C(...)),where **new** is the instance creating event.If we introduce a time-stamping [6], we have
exists(C(...,t)):-new(C(...,t1)),t1<t,
not destroy(C(...,t2)),t2<t,t1<t2.
destroy is the instance destroying event.

happens(e(...)) means that the environment activates e(...) and **pre-con** is the event pre-condition (the event is relevant).
Examples of **wff** are:
book(b,-,t),has-book(b,"r1",t),t=6

used as a query will give the number of books that reader r1 has at time instant 6.

reader(r,-.-),no-of-books(y,-)<6
used as an integrity constraint expresses that no reader will have more than 5 books.Obviously no-of-books will be a variable atribute of the class reader.

4. OO-DB AS THEORIES

Once the UL has been defined as a first order language for RMOL(in FMOL things happen in a similar manner), let us see how the equivalent clausal theory is obtained.

i.primitive class axioms

They incorporate the **primitive types**: nat(0), nat(1)...bool(true),...in an extensional or intensional way by using axioms. The most usual are predefined.

ii.elementary class axioms

Let **kc** be the constant attributes of an elementary class C. We add an axiom per class: C(**kc**,t):-exists(C(**kc**,t))

iii.complex class axioms

They implement how complex classes are defined.For example:
hasbook(b,r,t):-loan(b,r,t1),t1<t,
not(return(b,r,t2)),t2>t1,t2<t1.

iv.relevant events

a)If e(...) is a relevant event then e(...) is an axiom: loan("b1","r1",15)
b)Event pre-conditions are added as integrity constraints: for loan(b,r,t) we have: is-of-library(b,t),avalaible(b,t), is-a-reader(r,t),no-of-books(r,t)<6

v.attribute axioms

For every object instance of a class we will add
a)An axiom per constant attribute.For ex.:
title("b1","Hamlet",-)
They are redundant but we add them to have a unified treatment of the attributes.
b)The clauses defining the variable attributes. For example:

```
no-of-books(r,n,t):-loan(b,r,t),
                    no-of-books(r,n-1,t-1);
                      return(b,r,t),
                    no-of-books(r,n+1,t-1);
                    no-of-books(r,n,t-1).
```

vi.Prolog axioms

They are implicit in the Prolog interpreters that are used.
Note: by using forward inference we will also add the object or attribute instances for axiom types ii,iii,vb. The underlying Deductive Data Base will then be redundant.
Let us see the equivalent theory for the example:

```
ii.elementary class axioms
   book(b,r,t):-exists(book(b,r,t))
   reader(r,n,t):-exists(reader(r,n,t))
iii.complex class axioms
   has-book(b,r,t):-loan(b,r,t1),t1<t,
   not(return(b,r,t2),t2>t1,t2<t)
iv.relevant event axioms
   new("b1","Hamlet",1)
   loan("b1","r1",27)
   ...
 v.attribute axioms
   title("b1","Hamlet",1..-)
   ...
```

This equivalent theory is the DDB which is used for the evaluation of the **wff** of the UL. It is automatically generated from the ODL specification by a translator. These translators, editors, and an interactive environment constitute an OO-DB Management System.
In FMOL we proceed in a similar way. The object theory for a FMOL specification is a presentation in AXIS[7] or RAP[8], two functional languages.In the object theory the classes are represented by sorts. For each Conceptual Scheme **ec** we add the sort ec as the last sort in the arity of each operator.This will be the interest sort.Its type is T (the set of traces). Attributes,events, event pre-conditions and relations will be represented as operators (the only expressive mean in a presentation). The corresponding UL will be defined in a similar manner (the only predicate is =); then the **wff** will be sets of equations with logical variables.

5.CONCLUDING REMARKS

The two specification languages R/FMOL represent a blend of logical paradigms with good specification properties,a declarative style of the IS Conceptual Model following the OO model. It has good locality properties and a deductive style [9].

These languages allow the specification of OPIS. The system runs on SUN 3/60 work stations.BIM_PROLOG [8] is used as the implementation language and the object language in RMOL (AXIS or RAP in FMOL).It includes editors, translators and prototypers.
We are already working on future extensions of our work: incrementing the language expresivity to specifying OAIS and introducing a more flexible way of axiomatizing time (it will be a declared class).
Now the prototyper acts as an OAIS modelling the global environment, but each class may be viewed as a prototyping environment. Blend FMOL and RMOL to have a powerful expressivity with the three emerging specification paradigms: objectual, relational and functional.
We are also implementing graphical and friendly environments.

REFERENCES

[1] P.Lindgreen ed."A Framework of Information System Concepts".FRISCO. Interim Report.IFIP WG8.TG.90.
[2] A.Sernadas "Abstract Object Types:a temporal perspective".Colloquium on temporal logic and specification. Springer Verlag 89.
[3] P.Deutsch.In records of a discussion on the type concept.SIGPLAN Notices vol.16 n°1.ACM 81.
[4] J.Goguen et al."Unifying Functional Object Oriented and Relational Programming whith logical semantics". Research Directions in OO Programming. MIT Press 87.
[5] G.Gallaire et al."Logic and Databases:a deductive approach".Computing Surveys 84.
[6] R.Kowalsky."Logic as a Database language".New Generation Computing 87
[7] D.Coleman et al."The AXIS specification language".Tech.Rep.HP-Labs.88.
[8] A.Gesser et al."RAP".Tech.Rep. Univ.Passau 88.
[9] A.Olive."A comparison of the operational and deductive approaches to Conceptual System modelling".Proc.IFIP 86.
[10]BIM_PROLOG.ISS,Everberg.Belgium.

OO AND ACTIVE FORMAL INFORMATION SYSTEM SPECIFICATION.

Isidro Ramos, Oscar Pastor, Vicente Casado.

DSIC (DEPARTAMENTO DE SISTEMAS INFORMATICOS Y COMPUTACION) UNIVERSIDAD POLITECNICA DE VALENCIA
46071 VALENCIA (SPAIN)

ABSTRACT

Formal methods for Information System Specification is a field with a wide work background. Using an Object-Oriented model with a well-defined logical framework it is possible to execute the specifications as deductions in the formal first order theory equivalent to each specification, as was shown by Ramos [2]. The system implemented there is a powerful tool for rapid prototyping and constitutes a useful environment for open and passive IS that we call MOL (Micro-Object-Logic). Using ideas for graphical object oriented design from [4] , a wide spectrum OO specification language is presented in Bearing et al.[5] dealing with **O**pen and **A**ctive **S**pecification of **IS (OASIS)**. The present paper is a first working version of this enhanced OASIS, built by adding to MOL active capabilities for expressing active relationships between objects by means of the triggering relationships. By active we mean that the objects involved are necessary for changes of state to take place, which is done by defining triggering conditions.

We then present what we mean by OO model. The third section states our class definition. The fourth presents the provided interaction mechanisms between objects and finally we present the prototype as it has been implemented, giving the syntactic language constructions and commenting on the most relevant implementation features.

I. INTRODUCTION

According to Wand and Lindgreen [1] [6], the world is made up of objects. Objects describe concrete beings, that is, specific entities rather than types or classes. We put objects together in types by means of the abstraction mechanisms. We call class the set of properties, all of which characterize certain objects. A class is a precise characterization of structural and behavioural properties that a collection (or type) of objects all share.

Objects are known to (or observable by) the world through their properties. The specific set of properties used to describe a given object depends on the point of view and the purpose of modelling. We recognize properties only through attributes

The value of an attribute at a certain time is called a state variable. The set of state variables defines the state of the object.

From a static point of view, the allowed states of an object are determined by the set of static object laws, being these laws constraints on the allowed combinations of the values of attributes (static constraints).

A transition between two states is termed an event. An event is an abstraction of a change of state in the object system. It is discrete, has no duration and occurs at a certain point in time. An event will occur (will be relevant) if a given precondition associated with it is satisfied. These preconditions constrain the occurrence of events.

From a dynamic point of view, we can also have temporal laws describing relationships between valid states (dynamic constraints)

Objects are not independent. We have the sharing of events as a first mechanism of interaction between objects. We can also have active relationships by means of a triggering mechanism. Potentially, every condition defined in terms of the relevant object events and/or attributes may serve as a trigger.
A trigger is a relationship between an event and one or more other events that at a certain level of abstraction expresses the cause for the proper agents to carry out the execution of other events. A triggering relationship is defined by giving the event to trigger, the destination and the condition that will trigger it when fulfilled.

II. CLASS DEFINITION

Following the formal logic and OO framework presented in Ramos[2], we define a class as a first order theory, composed of an alphabet, a first order language, a given set of axioms of different types and a set of rules of inference.

The alphabet of our theory is made up of the set of class events and attributes.

Between the set of events, we can distinguish between **private and shared**. Private events are those belonging to only one class. Shared events appear in more than one class.

The set of attributes is composed of **constant** attributes and **variable** attributes. Constant attributes do not change their value during the object existence. Variable attributes change on occurrence of their relevant events according to dynamic laws.

The axioms composing our theory are those presented as follows:

1.-Relevant events, those which represent successful occurrences of events, i.e., events that have been activated by a valid agent and have satisfied the constraints associated to them.

2.- Static integrity constraints: those which state static relationships between attributes that must hold in every state to be considered valid. They are built using a first order logic whose wff are:

1.- a predicate representation of an attribute is a wff
2.- if A, B are wff, then A,B A;B not(A) are wff
3.- our wff are only those obtained by 1,2.

3.- Dynamic integrity constraints: those which establish dynamic properties from the events and/or attributes, relating more than one state. They are expressed by means of a temporal extension of the first order logic used in i), adding the events to the alphabet of our wff.

4.- Variable attributes definition: those which express the axiomatic definition of the variable attributes in terms of their relevant events. The chosen expressivity (clausal, functional or both)

D. Karagiannis (ed.), *Database and Expert Systems Applications*
© Springer-Verlag Wien 1991

Ramos, I.; Pastor, O.; Casado, V. OO and Active Formal Information System Specification. Proc. International Conference on Database and Expert System Applications (DEXA 91) Vol. II. 1991.

characterizes the specification language version. When using a clausal style for defining the variable attributes we have R-OASIS (RELATIONAL OASIS). Using a functional one we have F-OASIS (FUNCTIONAL OASIS). Blending both together we have a functional and relational expressivity.

Taking a relational logic approach, the variable attribute definition is given by clauses. The head is an attribute name and the body is the declarative attribute definition following its relevant events and/or its attributes.

attribute:-axiomatic_definition

5.- Event preconditions: these axioms relate every relevant event with the conditions that must hold for the successful event occurrence. The condition is expressed in terms of events and attributes of the theory representing the class being defined. We can represent it easily using a clausal expressivity, its head being the event and its body the corresponding condition.

event:-precondition

6.- Triggering relationships :they define active relationships between objects, expressing which conditions may potentially serve as triggers of events from other classes. We can express them in the form:

destination::event:-wff

which means that the given event will be activated and sent to the indicated destination (an object, all the objects of a class,etc.) as soon as the condition expressed by wff will hold. This condition will be given using the events and attributes belonging to the alphabet of the theory, as in the previous case. We can realize that we have a special kind of event preconditions that relate events of any other class with conditions expressed in terms of events and attributes of the class being defined.

III. INTERACTION BETWEEN OBJECTS

Objects are not independent due to the existence of special relationships between them.

Having defined our classes as first order theories, we can consider two interaction mechanisms between objects:

1.- shared events
2.- triggering relationships

We are going to determine both mechanisms in terms of the induced relations between the corresponding theories.

A. SHARING OF EVENTS

We have already stated that shared events are those belonging to more than one class. This implies that a relation exists between the theories sharing this kind of events, expressed in the existence of different types of axioms using the same shared event in the theories considered (those sharing the event).

The relationship between classes produced by the shared events allows us to identify the existence of an aggregated complex class that takes these events as private.

B. TRIGGERING

The triggering relationships between classes are our tool for modelling an active pattern of communication in our object society. This is done by specifying for each class which events can be asked for and from which classes when a given condition is satisfied. We can determine if the event is to be sent to all the objects of the class, to a special subset of them or to individual instances.

The triggering axioms presented above are those theory components which take into account the expression of these active communication patterns.

IV. THE SPECIFICATION LANGUAGE

Uing from MOL works and dealing with an OO, open and passive specification of IS, we present an active enhancement by means of the introduction of triggering expressions. This leads us to our Active-MOL (OASIS) in which the introduction of an active pattern of communication allows us to specify which events will be sent to their corresponding destination classes once the defined triggering condition will hold.

The structure of the enhanced language, how the specification is executed and the most relevant features of the implemented environment are discussed subsequently.

A. STRUCTURE

A Conceptual Schema, as a definition of an object society, is built according to the following template:

Conceptual_Schema <name>
<specification body>
end_Conceptual_Schema

The body is composed of the specification of the **domains, the elementary** and **the complex** classes, the three constructions provided by OASIS.

Domains (primitive classes)

The domains denote the data subspecification that are used as object surrogates and attribute classes. Their type is an ADT. Our object society will be built taking them as the basic data level used to declare elementary classes.

The actual OASIS implementation provides as predefined the basic domains nat, bool and string with their usual operations.

We treat **time** as domain, implicitly defined and necessary for applying the Kowalski event calculus to the specification language. It appears as the last argument of the predicates representing events and attributes, fixing respectively the instant of an event occurrence or the time in which we want to make an observation (i.e. to know some attribute values). We represent time as discrete and left-bounded using the natural numbers.

Each domain declared has an associated default value, the main constant, given between parenthesis after the domain name:

nat(0), bool(true),...

It is considered the default value for non-defined attributes.

Elementary classes

Elementary classes are those built without using class operators. They

define the structural and behavioural properties that a collection (or type) of objects all share.

Their syntactic representation follows the template:

```
        CLASS NAME
   ATTRIBUTES
     CONSTANT
      <ATTRIBUTE_NAME:CLASS_NAME> KEY,
      <ATTRIBUTE_NAME:CLASS_NAME>
       ...
     CONSTRAINTS
       <ADMISSIBILITY CONDITION FOR CONSTANT
ATTRIBUTES>
     VARIABLES
      ATTRIBUTE_NAME(ELEMENTARY_CLASS_NAME,
DOMAIN_NAME,TIME)
        CLAUSES <VAR.DEF>
            <CLAUSAL_DEFINITION>.
       ...
   EVENTS
      PRIVATE
       <EVENT_NAME>(<PARAMETERS>,TIME);
        ...
      SHARED
                <EVENT_NAME>(<PARAMETERS>,TIME);
        ...
   PRECONDITIONS
     <EVENT_NAME>(<PARAMETERS>,TIME) IF

        <EVENT_ADMISSIBILITY_CONDITION>
     ...
   TRIGGERING

DESTINATION_CLASS::<EVENT_NAME>(<PARAMETERS>,TIME)
IF
              <TRIGGER_CONDITION>
     ...
   END CLASS
```

First, we specify the structural properties of the class by means of the attribute declaration. We distinguish between constant attributes, those whose values do not change with time, and variable attributes, those whose values change over time depending on the relevant event occurrences. Between the constant attributes, we identify the key that is the object surrogate for class instances. For every variable attribute declared, we give its axiomatic definition in terms of their relevant events and/or attributes. The actual implementation uses a relational or clausal style for the deductive definition. So, we are dealing with an active and relational version of MOL (R-OASIS).

Using as an example the specification of an artificial lake (see fig.1), we will begin by declaring their constant and variable attributes. Constant attributes are its name (acting as key), capacity security limit and minimum reserve level. We declare the actual level as a variable attribute, giving its axiomatic definition in terms of its relevant events (those changing the level value). In the example, we do it using a relational expressivity, defining the level value in an instant t in function of its previous value (in the instant t-1) and its relevant events occured at time t.

```
ELEMENTARY CLASS ARTIFICIAL_LAKE
        ATTRIBUTES
          CONSTANT
            NAME:STRING KEY;
            CAPACITY:NAT;
                SEC_LIMIT:NAT;
                MIN_LEVEL:NAT;
          STATIC CONSTRAINTS
                              CAPACITY>SEC_LIMIT AND
SEC_LIMIT>MIN_LEVEL
          VARIABLES
          LEVEL(ARTIFICIAL_LAKE,NAT,TIME)
                                          C L A U S E S
L:ARTIFICIAL_LAKE;R:RESERVOIR;Y,C:NAT;
                                    T,T1:TIME;
           LEVEL(L,Y,T):-T1 IS T-1 AND LEVEL(L,Y1,T1) AND
            ((NOT(UNLOAD(L,C,T) OR LOAD(L,C,T) OR
                                    PASS_AL(L,R,C,T) OR
PASS_R(L,R,C,T)) AND Y IS Y1)                    OR
(UNLOAD(L,C,T) AND Y IS Y1-C) OR
                (LOAD(L,C,T) AND Y IS Y1+C) OR
                (PASS_R(L,R,C,T) AND Y IS Y1-C) OR
                (PASS_AL(L,R,C,T) AND Y IS Y1+C)).
          EVENTS
                PRIVATE
                 LOAD(L,C,T)
                 UNLOAD(L,C,T)
                SHARED
                 PASS_R(L,R,C,T)
                 PASS_AL(L,R,C,T)
          PRECONDITIONS
                UNLOAD(L,C,T) IF LEVEL(L,Y,T) AND Y-
C>MIN_LEVEL
          TRIGGERING
                SELF::DESTROY IF LEVEL(L,Y,T) AND
Y>CAPACITY
                SELF::UNLOAD(L,C,T) IF LEVEL(L,Y,T) AND
Y>SEC_LEVEL
                OBJECT:: STOP IF LEVEL(L,Y,T) AND
Y<CAPACITY
        END ELEMENTARY CLASS
```

Fig. 1: example of a class specification in OASIS.

An important feature of this enhanced OASIS is the possibility of specifying static integrity constraints between the constant attributes. Objects will be brough into existence only if this constraints on their constant attributes hold.

In our example, we can state a constraint to avoid having a security limit greater than the capacity.

Afterwards, we declare the events that are relevant to the class. We distinguish between private and shared events. Private events are those that only affect the elementary class being defined. We have two overloaded private events that are those creating and destroying class instances (**new** and **destroy**). We also have four specials private events acting over any instance of our object society and control the liveness state of the object. They can make an object active (**start**), or pause it while remaining active (**pause**), or to reactivate it again (**resume**) or, lastly, leave its active state (**stop**). These events act over any object, independently from the class to which they belong.

Shared events affects several elementary classes simultaneously. They allow us to define object interactions, and can act as private events of a complex aggregated class, relating the component classes (those sharing these events)

Again in our example, we have declared private events of the artificial lake class those of water **load** and **unload**. We could declare water exchange shared events between artificial lakes and water reservoirs related to them. The names assigned to these events are **pass_r** (representing the pass of a given quantity of water from an artificial lake to a reservoir) and **pass_al** (representing the movement of water in the other way).

Next we specify for every private event its admissibility condition. It states which condition must hold in order to have a valid event occurrence. We specify it in terms of events and/or attributes of the class we are defining.

For the new and destroy overloaded private events, we have resp.ectively the non existence/existence of the affected object as the default precondition.

Similarly, for the special activity events we have implicitly predefined preconditions. For **start**, the referenced object must not have received any of the other activity events. For **pause**, the object implicated must be alive in the sense that it must be in an active state (it has received a start or resume event before this one). For **resume,** the object must also obviously exists and has to be in a paused state. For **stop,** the only condition is the object existence.

In our prototype, all these events are predefined and assumed by default together with their corresponding preconditions.

In the chosen example, a precondition for the private event of releasing water would be define stating that the level of the artificial lake is greater than the minimum security level.

Last, we define the triggering relationships that will make our specification active. This activity is one of the most interesting features of OASIS, and allow us to evolve from an open and passive specification of information systems (OPSIS) given in MOL to another open and active specification of IS (OASIS).

Our system uses a similar mechanism to the used in KNOs. The difference consists of the messages sent. In KNOs the messages are objects as any other object that can also send messages, having a cyclic definition of object. OASIS uses the proper events as messages, considering them as degenerated objects whose life is the instant in which they are produced, having no attributes or events.

They are expressed by means of the syntax:
<DESTINATION>::<EVENT>
IF <EVENT_ADMISSIBILITY_CONDITION>
where destination specifies the receivers of the event, event is the triggered event and we specify the condition that will activate the trigger mechanism in the event´s admissibility condition.

We have four key words for the destinations:

1.- **self:** the triggered event destination is the object itself .

2.- **object:** the event destination is a unique object that will be asked for to trigger the event. Obviously, the object must be in conditions to receive this event, otherwise the trigger will fail.

3.- **class:** we send the triggered event to all the actual class instances (the class of the object considered).

4.- **society:** the event destination is all the instances of our object society.

When the destination is all the class instances, an event copy is triggered for every object, checking if they are valid. The trigger is considered successful if at least one of the event copies triggered is valid.

As an example of our trigger system expressivity, an artificial lake instance could send to itself a destroy event (a suicide!) if its capacity is exceeded. Or trigger a water unload event if its security level is reached.

Complex classes

Complex classes are those defined using class operators. The actual OASIS version provides at the moment only one kind called **relationships.** They represent an aggregation between classes that takes their shared events as private. When defining such a relationship we are dealing with a complex class whose instances have as surrogate space the Cartesian product of the surrogate spaces of the component classes. This new complex class will have its own set of attributes and events as emergent properties.

Their syntactic definition is given using the same constructs we use for elementary classes definition, with one difference: we have to specify explicitly what the events for creating (new) and destroying (destroy) complex class instances are.

An example including the presented language features is given in the appendix. It is about a simple river dams system involving a set of artificial lakes that are related to several water reservoirs. We can exchange water between any artificial lake and its related reservoirs in both directions. The specification declares two elementary classes (artificial_lakes and reservoir) and one complex class stating a relationship between them (the channel class).

B. IMPLEMENTATION

A specification in OASIS being equivalent to a clausal formal theory, as shown in Ramos[2], we are able to translate our specifications to a relational programming language, having the possibility of executing (or animating) them, with the added active capabilities. Every Conceptual Model specification is a first order theory, having the OO-DB extension as model (declarative semantics) and the deductive mechanism as the operational semantics. The OO-DB is implemented in R-OASIS over the standard model of the clausal theory equivalent to the IS specification.

In this way, we have built a prototyping environment composed of a graphical syntax driven editor, a translator that translates OASIS Conceptual Schemas into PROLOG logic programs and a prototyper that allows us to animate our specifications.

The translator reads a textual specification in OASIS and analyzes it syntactically. Then it generates the PROLOG logic program that represents the clausal theory to which the specification is equivalent.

The prototyper animates the specification, saving it when a prototyping session finishes, and loading the saved context to initiate another one if desired (object persistence).

The prototyper has two operating modes:

- manual: this is the mode which allows the introduction of events by external agents and the execution of queries about the society state. It provides a working environment similar to the one shown in Ramos et al.[7], although it has no active capabilities

- automatic: this mode deals with the event triggering. The implemented mechanism consider triggers to be service requests to other objects at given time instants. It allows us to deal with an active system, in which all the specified class triggering conditions are continously checked (following a turn-around strategy).

At the beginning of a session time takes the value 0. Every valid event is included in the system with its corresponding time occurrence point. Time is then incremented in one unit. We represent time points by natural numbers. Issues dealing with more complex time representations are being developed now (as discussed in Barbic et al. [3]).

The prototyper begins the animation in an automatic mode and we have to generate an interruption (in a controlled way) for to change to manual mode operation. In our implementation we capture the signal **sigint** of UNIX generated from the keyboard by means of typing [CONTROL-C]. To do this we use an interruption management system from PROLOG that treats this sign properly. Afterwards, the system checks possible holding triggering conditions. If it finds one, the prototyper generates the trigger by itself if possible, or asks the user for the needed parameters. In both cases, the user is informed about the triggers in progress as well as if they succeed or fail (because of precondition violation)

The events that can be triggered are both those defined in the specification and those predefined. The prototyper checks the correctness of the triggered event destination.

V. CONCLUSIONS

Starting from a formal class definition and from a precise characterization of what we mean by OO, we have built a working environment that gives us a powerful set of software tools for specifying open and active Information System within an OO and logic model. We can execute our specifications generating the logic programs that represent the clausal theory equivalent to the specification.

We are already working on future extensions of the OASIS environment, according to Bear et al [5]. In short, those extensions are:

i) definition and implementation of classical abstraction mechanisms as class operators. This will give us a constructive way of defining Conceptual Schemas.

ii)implementing graphical and friendly environments, using OO design methods.

iii) developing a more flexible way of axiomatizing time.

iv)improve the system expressivity by dealing with concurrence and paralellism.

v) implement different OASIS versions in order to have a wide expressivity spectrum for the Specification Language..

Our prototype runs on SUN 3/60 and HP 9000 workstations. It includes editors, translators and prototypers. BIM-PROLOG and ZYX_PROLOG respectively are used as the implementation languages.

REFERENCES

[1] Y.Wand. "A Proposal for a Formal Model of Objects" Object-Oriented Concepts, Databases and Applications Kim,W.Lochovski,S. eds. : pp.537-559 ACM Press Addison-Wesley 1989

[2] I.Ramos "Logics and OO-Data Bases: A Declarative Approach." DEXA 90 Springer Verlag 1990.

[3] F.Barbic, R.Maiocchi, B.Pernici "Automatic Deduction of Temporal Information." Universidad Politécnica de Milano. Interim report. 1990

[4] S.Bear,P.Allen,D.Coleman,F.Hayes. "Graphical Specification of Object Oriented Systems" OOPSLA 90

[5] S.Bear,D.Coleman,P.Hayes,O.Pastor:"OASIS: An Object-Oriented Specification Language" (submitted to DOOD 91)

[6] P.Lindgreen ed. "A framework of Information Systems Concepts" FRISCO Interim Report. IFIP WG8.1 1990.

[7] I.Ramos et al. "A Conceptual Schema Specification for Rapid Prototyping" XI-IASTED Conference on Applied Informatics .- Insbruck 1990

APPENDIX

```
CONCEPTUAL SCHEMA RIVER_DAMS
ELEMENTARY CLASS ARTIFICIAL_LAKE
      ATTRIBUTES
        CONSTANT
          NAME:STRING KEY;
          CAPACITY:NAT;
               SEC_LIMIT:NAT;
               MIN_LEVEL:NAT;
        STATIC CONSTRAINTS
                          CAPACITY>SEC_LIMIT AND
SEC_LIMIT>MIN_LEVEL
        VARIABLES
        LEVEL(ARTIFICIAL_LAKE,NAT,TIME)
                                        C L A U S E S
L:ARTIFICIAL_LAKE;R:RESERVOIR;Y,C:NAT;
                                    T,T1:TIME;
        LEVEL(L,Y,T):-T1 IS T-1 AND LEVEL(L,Y1,T1) AND
         ((NOT(UNLOAD(L,C,T) OR LOAD(L,C,T) OR
                                    PASS_AL(L,R,C,T) OR
PASS_R(L,R,C,T)) AND Y IS Y1)                    OR
(UNLOAD(L,C,T) AND Y IS Y1-C) OR
               (LOAD(L,C,T) AND Y IS Y1+C) OR
               (PASS_R(L,R,C,T) AND Y IS Y1-C) OR
               (PASS_AL(L,R,C,T) AND Y IS Y1+C)).
      EVENTS
            PRIVATE
             LOAD(L,C,T)
             UNLOAD(L,C,T)
            SHARED
            ..PASS_R(L,R,C,T)
             PASS_AL(L,R,C,T)
      PRECONDITIONS
            UNLOAD(L,C,T) IF LEVEL(L,Y,T) AND Y-
C>MIN_LEVEL
       TRIGGERING
            SELF::DESTROY IF LEVEL(L,Y,T) AND
Y>CAPACITY
            SELF::UNLOAD(L,C,T) IF LEVEL(L,Y,T) AND
Y>SEC_LEVEL
            OBJECT:: STOP IF LEVEL(L,Y,T) AND
Y<CAPACITY
END ELEMENTARY CLASS

ELEMENTARY CLASS RESERVOIR
      ATTRIBUTES
        CONSTANT
          RNAME:STRING KEY;
          RCAPACITY:NAT;
               RSEC_LIMIT:NAT;
               RMIN_LEVEL:NAT;
        STATIC CONSTRAINTS
              R C A P A C I T Y > R S E C _ LIMIT    AND
RSEC_LIMIT>RMIN_LEVEL
        VARIABLES
        RLEVEL(RESERVOIR,NAT,TIME)
                                        C L A U S E S
R:RESERVOIR;L:ARTIFICIAL_LAKE;Y,C:NAT;
                                    T,T1:TIME;
        RLEVEL(R,Y,T):-T1 IS T-1 AND RLEVEL(R,Y1,T1) AND
         ((NOT(RUNLOAD(R,C,T) OR RLOAD(R,C,T) OR
                                    PASS_AL(L,R,C,T) OR
PASS_R(L,R,C,T)) AND Y IS Y1) OR
(RUNLOAD(R,C,T) AND Y IS Y1-C) OR
               (RLOAD(R,C,T) AND Y IS Y1+C) OR
               (PASS_R(L,R,C,T) AND Y IS Y1-C) OR
               (PASS_AL(L,R,C,T) AND Y IS Y1+C)).
      EVENTS
            PRIVATE
             RLOAD(R,C,T)
             RUNLOAD(R,C,T)
            SHARED
             PASS_R(L,R,C,T)
             PASS_AL(L,R,C,T)
      PRECONDITIONS
            UNLOAD(L,C,T) IF LEVEL(L,Y,T) AND Y-
C>MIN_LEVEL
       TRIGGERING
            SELF::DESTROY IF RLEVEL(L,Y,T) AND
Y>RCAPACITY
            SELF::RUNLOAD(R,C,T) IF RLEVEL(R,Y,T) AND

Y>RSEC_LEVEL
            CLASS:: PAUSE IF RLEVEL(L,Y,T) AND
Y<RCAPACITY
END ELEMENTARY CLASS

COMPLEX CLASS CHANNEL(ARTIFICIAL LAKE, RESERVOIR)
      VALUE
L:ARTIFICIAL_LAKE;R:RESERVOIR,T,T1,T2:TIME;C:NAT;
       CHANNEL(L,R,T):-(PASS_R(L,R,C,T1) AND T1<=T) OR

PASS_AL(L,R,C,T2 AND T2<=T).
   EVENTS
            PRIVATE
                 PASS_R(L,R,C,T)
                 PASS_AL(L,R,C,T)
     PRECONDITIONS
    PASS_R(L,R,C,T) IF EXISTS(L,T) AND EXISTS(R,T).
    PASS_AL(L,R,C,T) IF EXISTS(L,T) AND EXISTS(R,T).
END COMPLEX CLASS
END CONCEPTUAL SCHEMA.
```

O. Pastor, I. Ramos, J. H. Canós
Dept. Sistemas Informáticos y Computación (D. S. I. C).
Universidad Politécnica de Valencia
Camino de Vera S/N. Apartado 22012
46071 Valencia (Spain)
email: PLO&GARBI.AII.UPV.ES
PHONE: +34 6 3877350 FAX: +34 6 3877359

Abstract

Taking a Deductive and an Object Oriented approach to specify Information Systems has become a rich research topic. Within this framework, RMOL [11] and its active evolution OASIS [13] [9] are two object- oriented specification languages that provide a clausal expresiveness for dealing with open and passive/active Information Systems, integrating an objectual and a clausal logic from the specification sake, and keeping an efficient and well-defined operational semantics by SLDNF-resolution for prototyping.

A weak point of RMOL was that its clausal expresiveness was restricted to normal clauses (those composed by literals, possitive or negative), loosing the full expressive power of general first order formulas.

This paper presents an enhanced RMOL version allowing to use program statements [8] without restrictions. A full first order expresiveness has been added to the clausal and objectual Specification Language (SL). The Deductive Data Base (DDB) field is reached by accomodating the first order theory equivalent to the specification (represented as a logic program in Prolog) to the input required by the DDB prototype presented in [4].

An environment composed of a set of Software tools has been built, with the common objective of using formal specification techniques based on clausal and objectual logics, their fusions and extensions, providing a higher expresiveness without loosing a well defined declarative and efficient operational semantics. With these techniques, the validation of software by automatic prototyping is guaranteed, and furthermore, they allow to approach topics such as consistency, completeness etc. in a formalized framework. They also allow to deal with System Dynamics animating the DDB prototype resulting from the specification.

1.- INTRODUCTION

A proposal for solving the problems related with production of software of quality (the so-called "crisis of Software") is the use of an automation-based life cycle [1] that, starting from the user requirements in the first Analysis step, lead us to a formal and executable specification. Formal because it will be done using a formal language and executable because this language will be provided with an associated operational semantics in a Logic framework.

From the specification sake, it is becoming more popular to use the Object-Oriented paradigm, that views an Organizational System as a society of interacting objects. To be precise, we have to define which is

Pastor, O. Ramos, I. Canós, J.H. Object Oriented and Relational Specification of Information Systems. Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS. 1991.

our OO framework giving a clear definition of what an object is and explaining what do we mean by object interaction.

The expressive power of our specification language will be increased if the most relevant properties of the three emergent programming paradigms are merged, i.e. relational, functional and OO [6]. RMOL [11] was created in this line, merging the relational and the OO paradigms and constituing a powerful software tool for specifying open and passive Information Systems. In this environment, FMOL [12] was an alternative development allowing to specify IS from a functional and OO point of view

In both cases, being the specification equivalent to a first order theory (clausal in RMOL, equational in FMOL), it is possible to produce automatically the logic program corresponding to the specification by means of translators that act as very high level compilers.

The implemented RMOL version uses a normal clausal notation that accepts only Horn clauses with negation. To improve the SL with the use of program statements [8] (those first order formulas of the form **A<-W** where the head A is an atom and the body W is an arbitrary first order formula) was a main goal to achieve for increasing the expressive power of the specifications.

On the other side, the Deductive Data Base (DDB) field has produced during the last years a well founded set of results allowing the implementation of DDB's prototypes working properly if determinated properties are satisfied. This is done by restricting the first order theories by considering only well fixed subsets.

The present work links an OO and relational specification level (represented by RMOL) and the DDB world (represented by the DDB prototype implemented in [4]). The expresiveness of RMOL has been enriched allowing to use program statements, and a translator has been implemented producing automatically a general first order theory. This constitutes the database scheme specification that is the input for the DDB prototype quoted before. In that way it is possible to handle the resulting DDB prototype equivalent to the initial specification.

This DDB prototype generated in an automated way has the following characteristics:

1.- It can analyze formulas and databases to check that they belong to certain classes of formulas (domain-independent, range-restricted, allowed, etc.) and databases (domain-independent, allowed, hierarchical, defined, normal, stratified, etc.). These classes of formulas and databases are used to ensure certain properties (absence of floundering, soundness, completeness, etc.) mainly related to the used SLDNF operational semantics.

2.- It can translate a specification of a database scheme into an operational database. This operational database is built by Prolog's rules. If the database belongs to the class of general databases, this translation needs, as a first step, to obtain an equivalent normal one.

3.- It can execute queries and updates in the current database. As in the previous point, if a query is a well-formed formula, the execution of the formula requires its normalization. An update can be a set or insertions and/or deletions of multiple facts and a set of insertions and/or deletions of deductive rules. When an update is executed, it is necessary to check the Integrity of the updated database.

In consequence, a Software Production Environment is implemented that, starting from an OO specification written in RMOL, produces automatically a DDB prototype. It is possible to animate it, having in fact a DDB arquitecture dealing with dynamic aspects from an OO and relational specification language.

The structure of the paper is the following: first what do we mean by OO is precised; second, the built prototyping environment is presented, distinguishing clearly between the specification level (how to specify the considered Information System using the SL) and the user level (dealing with how the resulting DDB is viewed by the user). After precising how the first order theory equivalent to a given specification is built, focus is set on the DDB prototype queries and animation, reviewing finally the most relevant implementation features.

2.- THE OO MODEL

Following [14] [5], the world is made of objects. Objects describe concrete beings, that is, specific entities rather than types or classes. By means of the abstraction mechanisms we put together objects in types. We call class a set of properties all of which characterize certain objects. A class is a precise characterization of structural and behavioural properties that a collection (or type) of objects all share. A class is elementary if it is not defined in terms of any other class. Complex classes are defined from one or more component classes, and among their characteristic set of properties there are is subset coming from this fact.

Objects are known (or observable) to the world through their properties. The specific set of properties used to describe a given object depends on the point of view and the purpose of modeling. We recognize properties only through attributes

The value of an attribute at a certain time is represented by a state variable. The set of state variable values defines the state of the object.

From a static point of view, the allowed states of an object are determined by the set of static object laws, being these laws constraints on the allowed combinations of the values of attributes (static constraints).

A transition between two states will be termed an event. An event is an abstraction of a change of state in the object system. It is discrete,

has no duration and occurs at a certain point in time. An event will occur (will be relevant) if a given precondition associated with it is satisfied. These preconditions constraint the occurrence of events.

From a dynamic point of view, we can have also temporal laws stating relationships between valid states (dynamic constraints).

Objects are not independent. We have event sharing as a first mechanism of interaction between objects. We can have also active relationships by means of a triggering mechanism. Potentially, every condition defined in terms of the relevant object events and/or attributes may serve as a trigger. A triggering relationship is defined giving the event to trigger, the destination and the condition that will trigger it when satisfied.

2.1. CLASS DEFINITION

According to the formal logic and OO framework presented in [11], we define a class as a first order theory, composed by an alphabet, a first order language, a given set of axioms of different types and a set of inference rules.

The alphabet of our theory is made up of the set of class events and attributes.

The set of events is composed of **private** and **shared** events. Private events are those belonging to only one class. Shared events appear in more than one class.

The set of attributes is composed of **constant** attributes and **variable** attributes. Constant attributes do not change their value during the object existence. Variable attributes change on the occurrence of their relevant events according to dynamic laws.

The axioms composing our theory are those presented next:

i) relevant events, representing succesful occurrences of events, i.e., events that have been activated by any valid agent and have satisfied the (optional) constraints associated to them.

ii) static integrity constraints: they state static relationships between attributes that must hold in every state to be considered valid. We represent them as relations whose terms are attributes.

iii) dynamic integrity constraints: they establish dynamic properties on the events and/or attributes, relating more than one state. They are expressed by using a temporal extension of the language used to represent static constraints, including events.

iv) variable attributes definition: they express the axiomatic definition of the variable attributes in terms of their relevant events. The chosen expresiveness (clausal, functional, both) characterizes the specification language version. Taking a relational logic approach, they

are represented by clauses, being their head an attribute name and stating their body the declarative attribute definition in function of their relevant events and/or attributes.

attribute:-axiomatic_definition

v) event preconditions: these axioms relates every event with the conditions that must hold to have a succesful event occurrence. The condition is expressed in terms of events and attributes of the theory representing the class being defined. We can represent it easily using a clausal expresiveness, being its head the event and its body the corresponding condition:

event:-precondition

It will be seen later how the RMOL Object Definition Language provides syntactic constructions to specify classes according to the given definition.

3.- SOFTWARE PRODUCTION ENVIRONMENT.

The working Software Production Environment provides an implementation of an automated life cycle, allowing to produce and manipulate DDB prototypes from OO and relational specifications. In fig.1 a general system representation is shown.

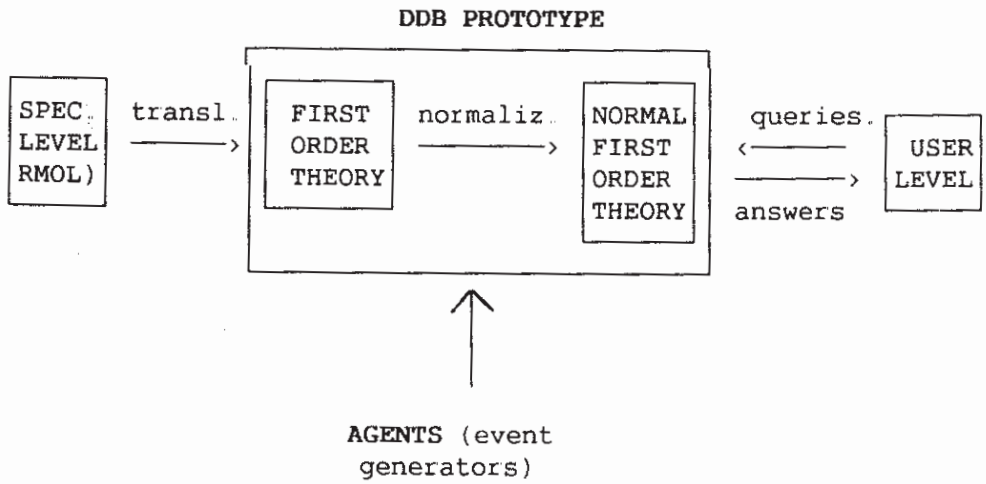


Fig.1: A graphical representation of the working environment

We have an specification level of the Information System, where an enhanced RMOL is used as object definition language. As a Specification in RMOL is equivalent to a first order theory [11], we represent this formal theory by means of a general logic program.

The enhanced RMOL has extended its expresiveness by allowing the use of program statements in the specifications. The equivalent first order theory is made up of a selection of program statements. It is used as input for a DDB prototype that will normalize the theory using an implementation of the LLoyd's algorithm [8]. After this operation, we

have a logic program in Prolog equivalent to the specification and constituting the core of a DDB prototype that is manipulated by the user.

The corresponding user level refers to the user language associated with the manipulation of the resulting theory (the DDB prototype).

A Conceptual Schema in the proposed environment is viewed as a society of interacting objects, allowing prototyping by generating the DDB equivalent to the specification. With respect to the System Dynamics, we have two approaches:

1.- to use Meta-Programming [7]. We must obtain the logic meta-program equivalent to a Specification in RMOL, defining properly the meta-predicates that will represent the System Evolution. A first set of results with this approach is presented in [10]

2.- to define a reflective logic [6] that considers a change of state from two possible perspectives:

i) evolving from one partial interpretation of the theory that represents the CS to another that includes the produced modifications. We will characterize in this way the so-called 'strong' dynamics.

ii)a substitution over the partial interpretation that represents the initial state (the state before the change). We characterize so the 'weak' dynamics.

An intensive work is being developed to formalize the System Animation within this framework.

3.1.- THE EXAMPLE

We use as an example for introducing the Specification Language a simple **Library** system. The library is made up of a set of readers, a set of books and a set of loans relating a book with its corresponding reader. A reader is identified by its **name** and has at any moment a **number** of books loaned. A book is identified by its **code**, it has a **title** and we want to know if it is or it is not **available** at a given time. **Loans** and **returns** occur in our System creating and destroying resp. loan occurrences. A given loan is identified by a **number** and has a **date** indicating when the loan has happened.

We will develop this example for explaining the new RMOL expresiveness.

3.2.- THE SPECIFICATION LEVEL

Corresponding to our OO framework and the given class definition as a first order theory, this new enhanced RMOL version provides syntactic constructs for specifying the three main specification units: **domains, elementary classes** and **complex classes**. All of them making use of a complete first order expresiveness. Let's be more precise with the structure of the language.

3.2.1. RMOL LANGUAGE STRUCTURE

A Conceptual Schema as a definition of an object society is built following the template:

Conceptual Schema <name>
<specification_body>
end Conceptual Schema

The body is composed by the specification of the domains, the elementary and the complex classes.

3.2.1.1 Domains.

The domains denote the data subspecification and are used as object surrogates and attribute classes. Its type is an ADT. Our object society will be built taking them as the basic data level used to declare elementary classes.

The actual RMOL implementation provides the basic domains **nat, bool** and **string** with their usual operations.

A special domain is the time, implicitly defined and necessary to use the Kowalski event calculus in our SL. It appears as the last argument of the predicates representing events and attributes, fixing the instant of an event occurrence or the time in which we want to do an observation (i.e. to know some attribute values). We represent the time using the natural numbers. More complex representations are now being developed, following [2].

Each domain has a default value associated, given between brackets after the domain name.

In our example, we use nat, bool and string as domains. So, we will write.

domains nat(0);bool(true);string('').

3.2.1.2 Elementary classes.

Elementary classes are those built without using class operators. They provide the structural and behavioural properties that a collection (or type) of objects all share.

Let's see their syntactic representation using as example the specification of the class book.

```
CLASS BOOK
      ATTRIBUTES
          CONSTANT
             CODE:STRING KEY;
             TITLE:STRING
          CONSTRAINTS
```

```
      CODE<100.000
    VARIABLES
      AVAILABLE(BOOK,BOOL,TIME)
        FORMULAS B:BOOK;V:BOOL;R:READER;T,T1:TIME
                 AVAILABLE(B,V,T):- SOME(R,SOME(T1,
                 (LOAN(R,B,T) AND V=FALSE) OR
                 (RETURN(R,B,T) AND V=TRUE) OR
                 (T1 IS T-1 AND AVAILABLE(B,V,T1)))).
      PLACE(BOOK,INT,TIME)
        FORMULAS B:BOOK;P,LOC:INT;T,T1:TIME
                 LOCATION(B,LOC,T):-
                 (MOVE(B,P,T) AND LOC=P) OR
                 (SOME(T1,T1 IS T-1 AND LOCATION(B,LOC,T1))).
  EVENTS
      PRIVATE
        MOVE(B,P,T)
      SHARED
          LOAN(R,B,T);
          RETURN(R,B,T)
    PRECONDITIONS
      MOVE(B,P,T) IF BOOK(B,T)
  END CLASS
```

First, we specify the structural class properties through the attribute declaration. We distinghish between **constant attributes**, those whose values don't change with time, abd **variable attributes**, those whose values change during the object life depending on their relevant event occurrences.

Among the constant attributes, we identify the key that constitutes the object surrogate generator for class instances. Taking the **book** class specification as example, constant attributes are **code** and **title**. In particular, **code** is the key used as object surrogates to identify individual class instances (i.e. individual books as objects).

We can specify static integrity constraints between the constant attributes. Objects will be brought into existence only if these constraints on their constant attributes hold.

In our class **book**, we can state that we accept only **code** values less than 100.000. We would write in the **constraints** paragraph: **code < 100,000**

For each declared variable attribute, we give its axiomatic definition in terms of their relevant events and/or attributes. RMOL does this definition by means of program statements. It uses a full first order expresiveness improving the old SL versions. Each variable attribute has a program statement related with it. Its head is an atom (the attribute name) and its body is a general first order formula stating the axiomatic attribute definition. We define the attribute value in a time t in function of their relevant events happened at t and its value in the previous instant (t-1).

For example, in our **book** class, we define if the book is **available** as follows:

```
available(B,V,T)<--- some(R,some(T1,
                        (loan(R,B,T) and V=false) or
                        (return(R,B,T) and V=true) or
                        (T1 is T-1 and available(B,V,T1)))).
```

assuming that all the variables that appears in the head are always universally quantified. We are stating that a book B is available in a time T if it exists a reader R such that he has returned the book B in T. If it has loaned B in T available is false. Otherwise, the value of available in T is equal to its value in T-1.

The next step on the specification process is to declare the events relevant to the class. We distinguish between **private** and **shared** events.

Private events are those affecting only the elementary class being defined. We have for every class two overloaded private events that are those creating and destroying class instances (resp. **new** and **destroy**). An example of a private event for our **book** class would be a **move** event representing changes in the book storage location in the Library.

Shared events affect two (or more) elementary classes simultaneously. They belong to the event alphabet of each of the classes that share them, and allow us to define object interactions, acting as private events of a complex aggregated class (presented below) that will relate the component classes (those sharing these events).

Coming back again to our **book** class, it has two shared events with a **reader** class: **loan** and **return**.

We follow with the so-called **preconditions** specification. Associated with every class event we can have an admissibility condition, stating which conditions must hold in order to have a valid event occurrence activated by a class instance. We declare these preconditions in terms of events and/or attributes of the class being defined.

For the **new** and **destroy** overloaded private events, we have resp. the non existence/existence of the involved object as implicit default preconditions.

3.2.1.3.- Complex classes

Complex classes are those defined by using class operators. The actual RMOL version provides at the moment only one kind of them called **relationships**. They represent an aggregation between classes that takes the respective shared events as private.

A defined relationship has the cartesian product of the component classes as its surrogate space. Being this complex class a new class, it can have its own set of attributes and events, with an extra constant attribute: the cartesian product of the key of the component classes. The key of the relationship can be this constant attribute, or optionally another one appeared as an emergent property of the aggregation.

Their syntactic representation is done as we do for elementary classes, but beginning with the special word **relationship** instead of class and indicating the component classes between brackets. We must also specify explicitly which are the events creating and destroying relationship instances. This is done by adding at the end of the corresponding private event declaration the word new or destroy.

In our Library System, we would declare a relationship **loans(book,reader)** stating that a loan is an aggregation having as components a reader and a book instances. The shared events **loan** and **return** between book and reader classes are taken as private events for the **loans** relationship. We write:

loan(book,reader,time) new
return(book,reader,time) destroy

for showing that the **loan** event is the one creating relationship instances and the **return** event the corresponding destructor.

The chosen **loans** relationship key is a **loan_number** constant attribute that is an emergent property of the new created aggregated complex class.

Another class operators corresponding to the classical abstraction mechanisms used in Semantic Modeling are now being developed to improve the RMOL semantic expressive power. In particular those of generalization/especialization and grouping/ownership. The active enhancement of RMOL already includes these class operators [9].

3.3.- FIRST ORDER THEORY EQUIVALENT TO A CLASS

Let's see the set of axioms that composes the first order theory to which a given class is equivalent, according to the class definition given in section 2, taking as example the **book** class of our Library System. Given a general first order formula F, existential and universal quantification are represented as some(x,F) and all(x,F) resp. The quoted theory is composed by the following set of axioms:

i) **domains axioms:** they incorporate the primitive types: nat(0), nat(1) ,... ,bool(true), ... in an extensional or intensional way by using axioms. The most usual are predefined. They provide the basic level over which classes are defined because they are used as object surrogates and attribute types.

ii) **class axioms.** Let k_C be the constant attributes of a class C. We add an axiom per class:

$$is\_present(C(k_C,t))$$

where $is\_present(k_C,t)$ indicates if the given class instance exists at time t. It is defined as shown:

$$is\_present(C(k_C,t)) :- some(t1,new_C(C(k_C,t1)),t1<t,not\ some(t2,destroy_C(C(k_C,t2),t2<t,t1<t2)))$$

being new and destroy the event resp. creating and destroying class instances.

We can manage with these class axioms the actual universe of objects for a given class.

iii) **relevant events**. If e(...) is a relevant event, then e(...) is an axiom. For example, **loan(b1,r1,10)** meaning that a loan event has happened in time 10, relating the book b1 with the reader r1.

iv) **event preconditions**. They are added as integrity constraints. For **loan(b,r,t)** we have for example:

book(b,t),reader(r,t),available(b,t),number-of-books(r,t)<6

v) **attribute axioms**. For every object instance of a class we will add:

a) An axiom per constant attribute. For example:

title(b1,Hamlet)

They are redundant, but we add them to have an unified treatment of the attributes.

b) The program statements defining the variable attributes. For example, for an attribute giving us the number of books of a reader we have:

```
number-of-books(r,n,t):-some(na,some(t1,
                  t1 is t-1 and number-of-books(s,n,t1) and
                  ((loan(b,r,t) and n is na+1) or
                    return(b,r,t) and n is na-1) or
                    (n is na)))).
```

We assume that all head variable are universally quantified.

We provide in this way an implementation of the class definition presented in 2). An intensive work is now in progress for including in our environment, static and dynamic integrity constraints and active components [13] [9] to cover all the class notion features.

The equivalent first order theory is the DDB which is used for the evaluation of the wff of the User Language, presented below. It is automatically generated from a RMOL specification by a translator. System dynamics (changes of state) are represented by inserting the relevant events in the theory.

3.4.- THE USER LEVEL

Having been defined a class as a first order theory, and being in particular the whole Conceptual Schema equivalent to a first order theory (built by parallel composition of all the component classes), we can associate it a corresponding first order language as the user language that will allow to query the DB state at any moment.

We take also into account the DB dynamics by animating the first order theory equivalent to the specification. This animation is done by introducing the relevant events in the theory.

3.4.1.- QUERIES

If a specification in RMOL is equivalent to a theory, we can use its first order associated language as User Language. We define next this UL in an inductive way.

Syntax of the UL.

Terms of domain d are:
i) A value of domain d
ii) A variable of domain d
iii) Only i),ii) are terms

Terms-a of domain d are:
i) If **C** is a class having **a** as attribute and k_C as the class of the key attributes, and i_{kC} is one of their instances, then **a(i_{kC},t)** is a terms-a of domain d. For example, being 'b1' the surrogate for a book instance, **available('b1',t)** is a terms-a.
ii) Only i) are terms-a.

Atoms:
i) If C is a class with constant attributes $a_1,\ldots,a_r$, and $t_1,\ldots,t_r$ are terms, then **C($t_1,\ldots,t_r$,t)** is an atom. For example, **book('b1','Hamlet',t)**.
ii) If f,g are terms or terms-a of the same domain, then **f rel-op g** is an atom, being rel-op some relational operators. For example, **number-of-books('r1',t)<6**
iii) If e is an event with arity class a_e and arguments a'_e and i_{ae} is one of its instances and i'_{ae} are the values of its arguments, then **e(i_{ae},i'_{ae},t)** is an atom. For example, **loan('b1','r1',6)**.
iv) Only i,ii,iii are atoms.

Well-formed formulas (wff).
i) An atoms is a wff.
ii) If A,B are wff then **not(A) A,B A;B A<-B** are wff
iii) If A is a wff and x is a variable, then some(x,A), all(x,A) (existencial and universal quantifier resp.) are also wff.
iv) Only i,ii,iii are wff.

Semantics of the UL

We take the object surrogate types and the types of the domains as the domain of the interpretation. We introduce also a time stamping, being the time the last argument of our terms-a and atoms. In RMOL only two constructions need auxiliary interpretation:

1.- let lambda be the semantic denotation function, and C($v_1,\ldots,v_r$) an instance of a class C. Then

lambda(C(($v_1,\ldots,v_r$,t)=true if is_present(C($v_1,\ldots,v_r$,t)) else false
where is_present(C($v_1,\ldots,v_r$,t))=true is defined in the following way:

existence-of(C($v_1,\ldots,v_r$,t)):- some(t1,new(C($v_1,\ldots,v_r$,t1)),t1<t,
not(some(t2,destroy(C($v_1,\ldots,v_r$,t2)),t2>t1,t2<t))

New and Destroy are resp. the instance creating and destroying events.

2.- Let be $e(v_1,...,v_r,t)$ an instance of the event e. Then

lambda($e(v_1,...,v_r,t)$)=true if happens($e(v_1,...,v_r,t)$) and pre-cond($e(v_1,...,v_r,t)$)=true else false

meaning happens(e(...)) that the environment activates e(...) and being pre-cond the corresponding event precondition.

Examples of wff are

book(b,_,t),loans(b,'r1',t),t=5

that used as a query will give the number of books that reader 'r1' has at time instant 6.

3.4.2.- DEDUCTIVE DATA BASE ANIMATION

In the Introduction, we said that the resulting DDB prototype allows to update the current database. Our environment is interested in one kind of updates only: those representing the insertion of relevant events. As we represent events as facts, an update will consist in the insertion of the fact representing the event. It will succeed if and only if the precondition associated to the event holds. In terms of the DDB, preconditions act as Integrity Constraints.

Any potential agent can activate class events that will change the System state. If the associated event precondition is satisfied, these new events are incorporated to the first order theory equivalent to the specification as axioms corresponding to the relevant events.

Our Theory grows while events happen in the System by inserting them. The System state is known through observations that give us the value of any object attribute depending on the sequence of events that the observed object put up with it.

In this way, the DDB prototype equivalent to a specification in RMOL changes in time and shows a behaviour represented as the proper increment of the DDB with the relevant events that happen.

3.5.- IMPLEMENTATION FEATURES

The actual working environment is running on SUN 3/XX and HP 9000 Series workstations. The system is composed of a RMOL graphical editor, a translator to a DDB, and a set of user tools that allow to animate and query the resulting DDB prototype. All of them integrated in a user-friendly environment. BIM-PROLOG [3] has been used as the implementation language.

4.- CONCLUSIONS AND FUTURE WORK

The implemented System constitutes a rapid prototyping Software Production Environment that allows us to produce automatically a DDB prototype from a high level specification language (RMOL). It constitutes a working implementation of an automated lyfe cycle for Sotfware production. RMOL uses a blend of the relational and OO paradigms. A clear

class definition is proposed for dealing properly with OO concepts. To develop a reflective logic semantics for RMOL specifications is now a main research topic in our group.

RMOL is an useful tool to be used in Object-Oriented Analysis and Design. We are increasing its expressive power adding class operators equivalent to the classical mechanisms of abstraction. Having already defined the aggregation by means of the so-called relationships, we are including specialization and grouping. We are also including the declaration of static and dynamic integrity constraints with the aim of improving the specification capacity. This is always done complying with our OO model and the given class definition.

Active capabilities has been added to the environment with the development of triggering relationships. This has produced OASIS as a natural enhancement of RMOL.

The resulting DDB prototype gives an interesting approach for linking high level specification languages as RMOL with the automatical production of DDBMS. This allows us to handle DDB that are equivalent to a formal specification, keeping all the results coming from the DDB world during the last years.

An advanced graphical environment is being developed, allowing any potential agent to 'open a class', acting in fact as a living object by generating relevant class events.

5.- REFERENCES

[1] Balzer,R. et al.:'Software Technology in the 1990s: Using a New Paradigm" IEEE Computer,Nov.1983.

[2] F.Barbic, R.Maiocchi, B.Pernici "Automatic deduction of temporal information." Politecnico di Milano. Interim report. 1990

[3] BIM-Prolog Reference Manual. ISS Everberg Belgium.1987

[4] J.C.Casamayor et al. "Deductive Data Base Management Systems: A Prototype".IASTED Conference on Applied Informatics 1990

[5] P.Lindgreen ed. "A framework of Information Systems Concepts" FRISCO Interim Report. IFIP WG8.TG.90,1990

[6] J.Goguen,J.Messeguer:"Unifying Functional,Object-Oriented and Relational Programming with logical semantics" Research Directions in OO Programming. MIT Press 1987

[7] P.M.Hill,J.W.Lloyd:"Meta-programming for Dynamic Knowdledge Bases". Technical Report CS-88-18, Dep.of Computer Science, University of Bristol,1988

[8] J.W.Lloyd: "Foundations of logic programming" Second Extended edition, Springer-Verlag 1987.

[9] O.Pastor,F.Hayes,S.Bear:"OASIS: An Object-Oriented Specification Language" (submitted to DOOD 91)

[10] O.Pastor:"Meta-programming in a OO and Deductive Environment for Software Production". Submitted to EDBT-92

[11] I.Ramos "Logics and OO-Data Bases: a declarative approach." Proceedings of DEXA 90. Springer Verlag,1990.

[12] I.Ramos,J.Canos,J.Oliver,R.Forradellas. "A Conceptual Schema Specification for Rapid Prototyping" Proc. of the IASTED Conference on Applied Informatics .- Insbruck 1989
[13] I.Ramos,O.Pastor. "OO and Active Formal Information System Specification". (to be published in DEXA-91)
[14] Y.Wand. "A proposal for a formal model of objects" Object-Oriented Concepts, Databases and Applications Kim,W.Lochovski,S. ed. : pags.537-559 ACM Press Addison-Wesley 1989

On the Use of Algebras as Semantic Domain of Object Societies

I. Ramos, J. Cuevas, J. Devesa, O. Pastor, J.H. Canós

Departament de Sistemes Informàtics i Computació

Universitat Politècnica de València

Camí de Vera, s/n 46020 València (Spain)

e-mail iramos@dsic.upv.es

November 9, 1999

Abstract

One of the main current research topics is to find a semantic domain for the Object-Oriented (OO) model. The interpretation of OO concepts has to reflect in a precise way the properties of the model, keeping at the same time good formal properties. One needs to be able in such a domain to cope with both structural and dynamic aspects. It is possible to use different sorts of logics (clausal, equational) to interpret the static aspects of objects; some extensions using temporal aspects (temporal logic, equational dynamic logic, ...) have been used to interpret the dynamic ones, but in these cases we lose an efficient operational semantics allowing the animation of objects and/or classes specifications. In software engineering, the need of such a semantics is crucial to give a meaning to the validation, verification, etc. of software via rapid prototyping.

In this paper, we use *term algebras* as the semantic domain for the object societies. Structural (*static*) aspects are thus handled using the traditional algebraic specification approach; to deal with dynamic aspects, a second-order extension of the equational logic is given. These second order aspects are managed in a first order style using syntactic machinery: operators and equational genericity. Doing things in

Ramos, I., Cuevas, J., Devesa, J., Pastor, O., Canós, J.H. On the use of Algebras as Semantic domains of Object Societies. Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS. 1992.

this way we have a well defined operational semantics allowing the animation of OO-specifications.

The ideas presented in this paper are in a incipient state; an example following them is completely presented in this paper. A first implementation has been done using the functional language and interpreter *Axis* as a reification of term algebras. The operational semantics is conditional term rewriting. The implementation, written in *Prolog by BIM*, runs over *Sun* workstations.

1 Introduction

Many attempts have been done in the last years to find a semantic domain for the Object-Oriented (OO) model. The interpretation of concepts such as *object*, *class*, *inheritance*, etc., has to reflect in a precise way the properties of the model, and at the same time to keep good formal properties. In such a domain, one needs to be able to cope with both structural and dynamic aspects. It is possible to use different sorts of logics (clausal, equational) to interpret the static aspects of objects; some extensions using temporal aspects (temporal logic, equational dynamic logic, ...) have been used to interpret the dynamic ones, but in these cases we lose an efficient operational semantics allowing the animation of objects and/or classes specifications. In software engineering, the need of such a semantics is crucial to give a meaning to the validation, verification, etc. of software via rapid prototyping.

In this paper, we use *term algebras* as the semantic domain: a class will be interpreted as a term of a given sort, an object as a ground term, etc. At the specification level, the functional expressiveness will cope with the classical OO concepts such as encapsulation, reification, and others, and allows an easy way to object interaction and inheritance (the classical abstraction operations: generalization, specialization, aggregation, part-of, ..., will be interpreted as operators over terms, and their semantics given in an axiomatic way). To deal with dynamic aspects, a second-order extension of the equational logic is given. These second order aspects are managed in a first order style using syntactic machinery: operators and equational genericity.

We will focus on the aplication of these ideas to OASIS, an open and active information systems specification language developed at DSIC. This paper is organized as follows. The next section makes a brief description of

the OO model underlying the OASIS language, as well as the basic langage constructs. Section 3 introduces the use of term algebras to establish the semantics of the object societies , and section 4explains how we do that with the OASIS specification language. The management of dynamic aspects is described in section 5. Appendices A to D show the complete source and object codes for the example discussed along the paper.

2 The OASIS framework

Starting from a view of information systems as societies of interacting objects, the OASIS model takes the concept of class as the base unit of the modelling process. Objects belong to classes, and are characterized by their structure (given in terms of *attributes*) and their behavior (described by the *events* than can happen in their lives). Complex classes can be built from simpler ones using class operators such as aggregation, generalization, ... Some of these complex classes are generated by object interaction, modelled in OASIS as event sharing and active relationships.

A class specification describes the structure and behavior of all its instances; appendix A shows the specification of a class named `person`; it's an elementary class because it has not been built from predefined classes. First, the structure of the class instances is defined in terms on the attributes; we distinguish between constant and variable attributes, depending on their ability to change its value during the objects' lives. Among the constant ones there is a distinguised attribute, the **key** that identifies uniquely each instance of the class.

The behavior is described by defining the events that can happen during the class' instances existence. We can define both private (or local in the sense of [Wie 91]) and shared events depending on the objects involved in the event occurrence. Each event has an associated precondition stating under which states of the object society the event can happen.

The last part of a class specification deals with the active side of the objects society. The keyword `triggering` starts the declarations of events that can happen without the influence of any external agent. Using a syntax similar to *Polka*([Dav 88]), the statements in this section indicate under which conditions a given event can be triggered.

3 Term algebras as the semantic domain for object societies

The problem of the Information Systems (IS) specification could be stated from our point of view as *how to define complex functions starting on the Abstract Data Types (ADT) world* using the facilities that functional languages usually offer. Moreover, when we want to specify an IS from an OO perspective, we must deal with two different and well known components of the object societies: the structure and the behavior. So, a functional and OO specification language should provide the specifiers with

- a *base language BL* in which a presentation $P = (\Sigma, E)$, being $\Sigma = (\mathcal{S}, \mathcal{O})$ the signature associated to P, reflects the structure of the objects society, and
- a *language extension* ΔL such that a presentation $P^* = (\Sigma^*, E^*)$ is used to describe its behavior.

The language BL+ΔL is just an algebraic specification of te OO specification language. So, a specification can be considered as a term (with variables) of BL+ΔL. A class can also be seen as a term, and a class instance would be a ground term of the language. Following this stream, we can define the usual OO concepts as *attribute, state function and state space (*[Wan 89]*),* ... in terms of functional concepts, and then to make all our discourse in an algebraic framework.

And a specification animation session will be composed by some theorem proving process against the equational theory built from the language BL+ΔL. We will represent the state of a given object by means of a ground term including all the object's story (see example on page **??**).

4 The Problem

We will focus on the OASIS framework. As we explained in the previous section, and according to the algebraic specification of the language given in Appendix B, we can see a specification like that shown in the Appendix A as a term:

```
class("person",
          attributes(
                        constant(
                                    key("dni"@nat),
                                    nokey("name"@string;"age"@nat;nil~c),
                                    constraints(...)
                                     ),
                        variable(...)
                        ),
           events(...),
           preconditions(...),
           triggering(...)
          )
```

having a function symbol for each keyword in the specification language. We also introduce a number of sorts to cope with the different kinds of terms allowed by the language.

When dealing with class instances, we can interpret objects as the second order term

```
\label{pagejem}
*ev("earn",1,500,
    *ev("expend",1,76,
        *ev("earn",1,100,
            *ev("create",1,"pepe"!12!nil~vc,nil~tb))))
```

We represent any object by the term representing its life, since the creation event until the last event happened. Constant attributes take their values at the object's creation time, whilst the evaluation of the variable attributes is done over the object's life. `*ev` is a generic constructor allowing us to manage second order aspects from a first order perspective.

5 The dynamics

References

[BCG 83] Balzer, R., Cheatman, T.E. and Green, C., *Software Technology in the 1990's: Using a New Paradigm*, IEEE Computer, Nov. 1983.

[BIM 91] *Prolog by BIM Release 3.0 Reference Manual*, ISS, Belgium, 1991.

[CDG 88] Coleman, D., Dollin, C., Gallimore, R., Arnold, P. and Rush, T., *An Introduction to the Axis Specification Language*, Technical Report, Hewlett-Packard Labs., Bristol, UK, 1988.

[Dav 88] Davison, A. *Blackboard Systems in Polka*, Dept. of Computing, Imperial College, London, 1988.

[IFI 90] Lindgreen, P. ed., *A Framework of Information Systems Concepts*, Interim Report from IFIP WG 8.1 Task Group FRISCO, 1990.

[PHB 92] Pastor, O., Hayes. F. and Bear, S., *OASIS: An Object-Oriented Specification Language*, to appear in Proceedings of the CAISE-92 Conference.

[RCO 89] Ramos, I., Canós, J.H., Forradellas, R. and Oliver, J., *A Conceptual Schema Specification System for Rapid Prototyping*, Proc. of the XI IASTED , Feb. 1990.

[RAM 90] Ramos, I., *Logics and OO-Databases: A Declarative Approach*, Proc. of the DEXA-90, Springer-Verlag, 1990.

[SSE 89] Sernadas, A. , Sernadas, C. and Ehrich, H.-D, *Object-Oriented Language Features for Information Systems Specification*, INESC 89.

[Wan 89] Wand, Y., *A proposal for a formal model of Objects*, in Kim and Lochowsky (eds), *Object-Oriented Concepts, Databases and Applications*, ACM Press, Addison Wesley, 1989.

[Wie 91] Wieringa, R.J., *A Formalization of objects using equational dynamic logic*, Proc, of the DOOD'91 Conference, Springer-Verlag, 1991.

A Example: the OASIS specification of the class Person

```
elementary class person
    attributes
      constant
        dni   : nat key
        name  : string
        age   : nat

      variable
        money(person): nat (0)
          equations: p: person; n,m: nat;
             money(p)= n if T=T1'earn(m)
                            and money(p)=m-n at T1.
             money(p)= n if T=T1'expend(m)
                            and money(p)=m+n at T1.
        rich(person): bool (false)

            % equations defining the attribute

    static constraints
        age < 100
        age >  18

    events
      private
        create(p) new
        remove(p) destroy
        earn(p,n)
        expend(p,n)

      shared
        % shared events declaration

    preconditions
```

```
        T=T1'expend(p,n) if money(p)>=n.

   triggering
        self::earn(p,100) if create(p).
end class
```

An Equational Constraint Logic approach to Conceptual Modelling[1]

María Alpuente María José Ramírez

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
Apartado 22012
46020 Valencia (Spain)
e_mail: maria(mjose)@dsic.upv.es

Abstract

One relevant approach for developing advanced database and knowledge-based systems advocates the use of logic programming technology [17,28,30]. Recently, the logic programming paradigm has been generalized to the framework of Constraint Logic Programming (CLP), a generic scheme for the introduction of constraints in logic programming defined in [24,25] and refined in [20]. In this framework, logic and equational programming have been integrated to define, as an instance of the scheme, a new declarative programming language, CLP(H/E), specialized in solving equations in equational theories [1,2]. In this paper we present, using the experimental language CLP(H/E), equational constraint logic programming techniques as an effective tool to support database applications. These techniques are able to operate with running specifications in two useful modes, parsing mode and generating mode, as they are fitted in themselves with an inferential capability which can be used for plan-generation [28,33,41].

1. Introduction

Among the main topics in the field of database design, Conceptual Modelling [10,19,34] plays a major role. An interesting direction for research on conceptual modelling is the investigation of tools for systematically deriving an operational system from its specifications (rapid prototyping [5]). It is a widely accepted thesis that this aim can be achieved by using a formal approach. As is well-known, a formal specification can be given an operational semantics which makes it feasible to consider the (executable) specification as a (very high level) prototype of the system. The different approaches to the problem of formally specifying the conceptual model of a database application differ in their level of mathematical accuracy and semantical clarity. Some attractive and well-established proposals reclaim ideas from the field of (algebraic) equational specification [13,14,37]. Also, the design of database applications in logic programming, as proposed in [17,28] and lately discussed in [30,31] supports conceptual modelling through logical tools [8,10,16,19,34,40].

A lot of research activity has been devoted lately to the integration of logic and equational programming, the motivation for which will not be repeated here (see [6, 12] for a survey; see also the collection in [11]). The usefulness of equational logic tools for conceptual modelling was discussed in [10,39,40]. States are denoted by compositions of update operations which, represented by function symbols, map states into states. In turn, equality and other relations are used to assert properties and relations between objects at a state. However, the problem of the executability of such specifications is not addressed, which raises the sensitive matter of deriving an answer from the axioms. None the less, because of the built-in equality and the chance of use of logical variables, this approach involves an enhanced capability for conceptual modelling which, in every sense, is strictly more powerful than the one in, either, the logic or equational styles alone [3].

Recently, the logic programming paradigm has been generalized to the framework of Constraint Logic Programming (CLP) [20,24,25]. Each instance CLP(χ) of this scheme is a programming language that arises by specifying a structure χ of computation. In this framework, logic and equational programming have been integrated to define, as an instance of the scheme, a new logic programming language, CLP(H/E), specialized in solving equations under a Horn equational theory E [1,2]. The intended structure is given by the finest partition induced by E on the Herbrand Universe H of the program. In the following, we will refer to such a structure as H/E. The advantages of this approach to the integration of logic and equational programming are that, since the language is an instance of the scheme, the fundamental semantic properties of (pure) logic languages (namely, the existence of simple and rigorous operational, fixpoint and model theoretic equivalent semantics) are automatically inherited by it. Moreover, the constraint solver can easily be embedded into a general CLP system and it can cooperate with other solvers.

In this work we present, using the executable specification language CLP(H/E), equational constraint logic programming techniques as an effective tool to support conceptual modelling of database applications. The key idea in our approach is to extend the methods in [3,16,22,39,40] by introducing constraints. We will show that these techniques are able to operate with running specifications in two useful modes, parsing mode and generating mode, since the built-in equality together with the intensional representation by means of constraints of the set of solutions to a goal fit these tools with an enhanced inferential capability which can be used for plan-generation [10,16,22,28,33,41].

This paper is organized as follows. In Section 2 we discuss the requirements of a guide database example, which will be used throughout the paper. In Section 3 we briefly review the language CLP(H/E) and its operational model (for a full characterization of the language we refer to [2]). Section 4 is devoted to illustrating the two useful modes, parsing mode and generating mode, we propose in order to operate with this kind of specification. Finally, Section 5 concludes.

It is assumed that the reader has a basic familiarity with the concepts of logic programming [4,30], CLP [24,25], conditional rewriting [27] and universal unification [38].

[1]This work was supported by CICYT under grant TIC 91 - 0425

Alpuente, M., Ramírez, M. J. An Equational Constraint Logic approach to Conceptual Modelling. In Ramos, I. and Tjoa, A. (eds), Proc. of the International Conference on Database and Expert Systems Applications (DEXA 92), 1992.

2. A guide example

The database of an employment agency was used in [39] in order to illustrate several possible ways of formally specifying a database application. Next we briefly recall the original informal specification. We assume the database to be initialized to an empty (nil) state. There, people apply for positions, companies subscribe by offering positions and companies hire candidates and fire employees. The following *integrity constraints* are imposed:

- a person may apply only once, thus becoming a candidate, losing this status when hired but regaining it if fired.
- a company may subscribe several times, the positive number of offerings being added up.
- only persons that are currently candidates may be hired and only by companies having free posts.

The allowed queries cover: checking whether a person is a candidate, whether a person works for a company and obtaining the number of free posts of a company.

In order to organize the above information, we conceive *updates* as operations mapping states into states and *queries* as operations (probably ranging over true or false) indicating whether or not a state has the corresponding property or indicating the result of some consultation about a state. So, the repertory of updates is:

nil	initialization without parameters. *prec.* none
apply(X,S)	a person X becomes a candidate. *prec.* X is not a candidate and X is not an employee
offer(Y,N,S)	a company Y offers N positions. The total number of free posts in Y will be increased by N. *prec.* N > 0
hire(X,Y,S)	a company Y hires a person X. The total number of free posts in Y will be decreased by 1 and X ceases to be a candidate. *prec.* X is candidate and Y has free posts
fire(X,Y,S)	a company Y fires a person X. Then X becomes a candidate again and the total number of free posts in Y will be increased by 1. *prec.* X works for Y

and the repertory of queries:

freepos(Y,S)	returns the total number of unfilled posts in the company Y
iscand(X,S)	tests whether a person X is a candidate (holds if X has applied and he is not employed)
isempl(X,S)	tests whether a person X is an employee (holds if X has been hired and not fired)
worksf(X,Y,S)	tests whether a person X works for a company Y (holds if X has been hired and not fired by Y)
haspos(Y,S)	checks whether Y has unfilled posts

The database, starting at an initial state, attains other states by means of updates. Each state T of a particular database can be denoted by some chosen composition of updates, leading from the initial state to T. Among the updates, we can distinguish a number of privileged operations (constructors) that serve to build up every data element. The state will be denoted by a well-formed term made only by constructor symbols. For the employment agency example, we choose the following operations as constructors: nil, apply, offer and hire, since it is assumed that a fire operation corresponds to a cancellation of a contract. Of course, fire operator should be chosen as constructor if, e.g., queries about past contracts were to be considered. In order to describe the effect of an update over the states, it is adequate to describe how it affects the constructor terms. It is useful to realize that some updates require the previous application of others if they are to be acceptable and productive (e.g., it is not possible to hire a person unless the person is a candidate and the company has free posts). Therefore, we will specify the (pre-)conditions to apply each update in order to enforce the integrity constraints. Just like updates and integrity constraints, the answer to the queries should be also defined by induction over the structure of the state.

Prior to the specification problem itself, comes the decision about which formalism to employ. From the above discussion it becomes evident that the formalism should provide, at least, the following features, all of which must be handled in a unified framework:

- equality and other relations (in order to define the effect of the updates and the answer to the queries)
- distinction between constructors and defined functions (in order to support the use of a constructive algebraic specification method)
- axioms between constructors (in order to easily reject unacceptable updates)
- a mechanism to enforce (integrity) constraints
- some kind of negation

In a pure logic formalism (without equality) the main drawback is that we have no possibility of defining functions. All functions in the conventional sense must be expressed as relations with an extra "result" argument (i.e. we define the graph of the function instead of the function itself) and function symbols serve merely as data constructors. However, the use of unification (the basic operation of logic programming languages) and its result, the logical variable, is the basis of some relevant programming features such as invertibility, partially determined data structures and non-determinism. Of course, through the equality relation, the functional notation comes in. In an equational logic formalism, we can formulate the problem both in a functional or in a relational style, having the possibility of choosing the formalism more suitable for each application. Interpreted function symbols can appear as arguments of relations and existentially quantified variables can appear as arguments of functions.

Constraint Logic Programming can be thought of as a generalization of Logic Programming where unification is replaced by the more general concept of constraint solving over a computation domain and the classic concept of computed answer substitution is replaced by the much more general and compact one of computed answer constraint [24,25]. If the integration of logic and equational programming is formalized within the CLP framework [1,2], the joined power of the three programming paradigms will be on hand.

3. The Equational Constraint Logic language CLP(H/E)

In this section we refer to a language which is an instance of CLP, as defined in [24]. Let us next briefly recall the essential concepts of the CLP framework as well as some necessary notions and results about equations, conditional rewrite systems and universal unification.

3.1. Preliminaries

A constraint logic programming language, CLP(χ), exists in the context of a particular structure χ which determines the meaning of the function and (constraint) relation symbols of some language. By Σ, Π and V (possibly subscripted) we denote denumerable collections of function symbols, predicate

symbols and variable symbols, respectively. $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the sets of terms and ground terms built on Σ and V. A (Π,Σ)-*atom* is an element $p(t_1,...,t_n)$ where $p \in \Pi$ is n-ary and $t_i \in \tau(\Sigma \cup V)$, $i=1,...,n$. A (Π,Σ)-*constraint* is a (possibly empty) conjunction of (Π,Σ)-atoms. The symbol ~ will denote a finite sequence of symbols. $\tau(\Sigma)$ is usually called the Herbrand Universe (H) over Σ.

DEFINITION 3.1.1. (CLP programs) Let $\Pi = \Pi_C \cup \Pi_B$ and $\Pi_C \cap \Pi_B = \emptyset$. A (Π,Σ)-*program* is a set of clauses of the form $H \leftarrow c \square$. or $H \leftarrow c \square B_1, ..., B_n$., where c is a (possibly empty) finite (Π_C,Σ)-constraint, and H (the head) and $B_1, ..., B_n$ (the body), $n \geq 0$, are (Π_B,Σ)-atoms. The $\square$ connective must be interpreted as logical conjunction.

(Π,Σ)-*programs*, (Π,Σ)-*atoms* and (Π,Σ)-*constraints* will often be called *programs, atoms and constraints*, respectively. A *goal* is a program clause with no head. An answer to a goal G is no longer a substitution but a constraint c such that:

$$(P, \mathcal{S}) \models (\forall)\,(c \Rightarrow G) \qquad \text{(logical semantics)}$$
$$P \models_{\chi} (\forall)\,(c \Rightarrow G) \qquad \text{(algebraic semantics)}$$

where P is a program, χ is the structure and $\mathcal{S}$ is a theory axiomatizing χ. CLP imposes some restrictions on χ, $\mathcal{S}$ and its relationships in order to guarantee the equivalence between both (logical and algebraic) declarative semantics.

The operational semantics of CLP can be defined in terms of transition systems [35]. We consider configurations of the form $< \leftarrow c \square A_1, ..., A_n .>$, where c is a satisfiable constraint and $A_1, ..., A_n$, $n \geq 0$, are atoms. The rule defining a *(P,χ)-computation step* between configurations is given in the following definition.

DEFINITION 3.1.2. ($\rightarrow_{CLP(\chi)}$ transition relation)

$$\frac{\chi \models (\exists)\ c \cup \tilde{c}}{< \leftarrow c \square A_1, ..., A_n > \ \rightarrow_{CLP(\chi)} \ < \leftarrow c \cup \tilde{c} \square B_1, ..., B_n >}$$

if there exist n variants of clauses in P, $H_j \leftarrow c'_j \square \tilde{B}_j$., $j = 1, ..., n$, with no variables in common with $\leftarrow c \square A_1, ..., A_n$, and with each other and $\tilde{c} = \{c'_1, ..., c'_n, A_1 = H_1, ..., A_n = H_n\}$. A terminal configuration has the form $< \leftarrow c \square >$, where c represents the computed answer constraint of the derivation.

DEFINITION 3.1.3.
A Σ-*equation* $s=t$ is a pair of terms $s,t \in \tau(\Sigma \cup V)$. A Horn equational Σ-*theory* E consists of a finite set of equational Horn clauses of the form $l = r \leftarrow e_1, e_2, ..., e_n$, $n \geq 0$, where the Σ-*equation* $l = r$ in the head is implicitely oriented from left to right and the literals e_i in the body are ordinary non-oriented Σ-*equations*. Σ-*equations* and Σ-*theories* will often be called *equations* and *theories*, respectively.

A Horn equational theory E can be viewed as a term rewriting system R where the rules are the heads and the conditions are the respective bodies. If all clauses in E have an empty body then E and R are said to be unconditional, otherwise they are said to be conditional. The equational theory E is said to be canonical if the binary one-step rewriting relation $\rightarrow_R$ defined by R is noetherian and confluent [27]. E is said to have extra variables when variables which do not appear in the left hand side (lhs) of the head of the clauses are allowed in the respective right hand side (rhs) and/or in the respective body. For syntactical characterizations of confluent theories refer to [29,32].

Each Horn equational theory E generates a smallest congruence relation $=_E$ called E-*equality* on the set of terms $\tau(\Sigma \cup V)$ (the least theory that contains all pairs $s = t$ such that $E \models s = t$)[2]. We will denote simply by H/E the finest partition $\tau(\Sigma)/=_E$ induced by E over the set of ground terms $\tau(\Sigma)$.

DEFINITION 3.1.4.
Given two terms s and t, we say that they are E-*unifiable* (or E-*equal*) iff there exists a substitution σ such that $s\sigma$ and $t\sigma$ are in the congruence $=_E$ (or, equivalently, $E \models s\sigma = t\sigma$). The substitution σ is called an E-*unifier* of s and t.

E-*unification* can be viewed as the process of solving an equation within the theory $=_E$. Since E-*unification* is only semidecidable, an E-*unification* algorithm can be viewed as a semidecision procedure for testing the solvability of equational constraints over the quotient H/E. Each instance of an E-*unifier* represents a solution over this structure. An E-*unification* procedure is complete if it generates a complete set of E-*unifiers* for all input equations A number of E-*unification* procedures have been developed in order to deal with conditional equational theories [15,21, 23,26]. Three approaches are relevant to the problem of computing the set of E-unifiers of two terms, namely flat-SLD-resolution [7,29], complete sets of transformations [21] and paramodulation [21] or some special kind of it, such as narrowing [7,18,26]. For instance, conditional narrowing has been shown to be complete for conditional theories satisfying different restrictions [18,21,26].

DEFINITION 3.1.5.
Let E be a Horn equational theory. A function symbol $f \in \Sigma$ is called irreducible iff there is no clause $(l = r \leftarrow e_1, e_2, ..., e_n) \in$ E such that $l \in V$ or f occurs as the outermost function symbol in l; otherwise it is a defined function symbol.

In theories where the above distinction is made, the signature Σ is partitioned as $\Sigma = C \cup F$, where C is the set of irreducible function symbols and F is the set of definite function ones. The members of C are also called *constructors*. The use of irreducible symbols is justified from the several optimization techniques defined in [15,23]. This affects the size of the search tree of the E-*unifiers* for a set of equations. In some cases, an infinite search tree can be reduced to a finite one.

3.2. The language CLP(H/E)

In this section we review the definition of the language CLP(H/E) [1,2]. Roughly speaking, CLP(H/E) programs consist of two sets of definite clauses. The relational part contains only clauses whose head is a (non-equational) atom and whose body may contain atoms as well as equational constraints (i.e. = is the only predicate symbol for constraints and it will be interpreted as semantic equality over the domain), whereas the equational part contains only clauses entirely built from equations.

DEFINITION 3.2.1. CLP(H/E) programs
Let $\Pi_C = \{=\}$, $\Pi = \Pi_C \cup \Pi_B$ and $\Pi_C \cap \Pi_B = \emptyset$. We define a CLP(H/E) (Π,Σ)-program as a (Π,Σ)-program augmented by a canonical Horn equational Σ-theory E without extra variables.
Some comments about the the language are in order:

- CLP(H/E) does not implement negation by failure[3]. We prefer

[2] We assume that interpretations obey the axioms of equality, i.e. the reflexivity, symmetry, transitivity and the substitutivity for functions and predicates. Hence, satisfiability and logical consequence are defined w.r.t. these axioms.

[3] For reasoning about negation, the theory E should be extended to a theory $\mathcal{S}$ being satisfaction complete, i.e. $\mathcal{S} \models \neg c$ whenever not $\mathcal{S} \models \exists c$. Hence the soundness and completeness results for finite failure and negation as failure in [24,25] are not inherited by the language as equational satisfiability is only semidecidable.

to handle negation by incorporating negative information in the form of rewrite rules, which are used to narrow equational constraints to false. An equational approach to negation is also taken in [15,32].

- Equational clause headers $f(t_1, t_2, ..., t_n) = t$ are explicitly oriented and written as $f(t_1, t_2, ..., t_n) \rightarrow t$.
- Some built-in operators, such as "<", ">" and *length* are assumed to be defined as usual.
- The standard sugared form of boolean expressions is allowed:

Raw form	*Sugared form*
$f(t_1, t_2, ..., t_n) = true$	$f(t_1, t_2, ..., t_n)$
$f(t_1, t_2, ..., t_n) = false$	$\neg f(t_1, t_2, ..., t_n)$

Table 1

In order to illustrate our ideas, we will go on with the example of the employment agency database. We describe its conceptual scheme as a formal theory in first order logic with equality. In our approach, we will present this theory as a CLP(H/E) program. The effect and preconditions[4] of each update and the answer to each query are uniformly described as CLP(H/E) program clauses. The declarative and operational semantics of the language thus provide formal support to the aspects of interpretation and executability of the theory. Correctness and completeness of the processes of query evaluation and integrity constraints checking is guaranteed by the equivalence results between both semantics [2]. A CLP(H/E) specification follows.

```
% SPEC agbd.

%                RELATIONAL CLAUSES

%                CLAUSES FOR IS CANDIDATE
iscand(X,apply(Z,T))  :- X=Z,pcapply(Z,T)        □ .
iscand(X,apply(Z,T))  :- ¬(X=Z),pcapply(Z,T)     □ iscand(X,T).
iscand(X,offer(W,M,T)) :- pcoffer(W,M,T)         □ iscand(X,T).
iscand(X,hire(Z,W,T)) :- ¬(X=Z),pchire(Z,W,T)    □ iscand(X,T).

%                CLAUSES FOR IS EMPLOYEE
isempl(X,apply(Z,T))  :- ¬(X=Z),pcapply(Z,T)     □ isempl(X,T).
isempl(X,offer(W,M,T)) :- pcoffer(W,M,T)         □ isempl(X,T).
isempl(X,hire(Z,W,T)) :- X=Z,pchire(Z,W,T)       □ .
isempl(X,hire(Z,W,T)) :- ¬(X=Z),pchire(Z,W,T)    □ isempl(X,T).

%                CLAUSES FOR WORKS FOR
worksf(X,Y,apply(Z,T))  :- ¬(X=Z),pcapply(Z,T)   □ worksf(X,Y,T).
worksf(X,Y,offer(W,N,T)) :- pcoffer(W,N,T)       □ worksf(X,Y,T).
worksf(X,Y,hire(Z,W,T)) :- X=Z,Y=W,pchire(Z,W,T) □ .
worksf(X,Y,hire(Z,W,T)) :- ¬(X=Z),pchire(Z,W,T)  □ worksf(X,Y,T).

%                CLAUSES FOR HAS POSITIONS
haspos(Y,apply(Z,T))  :- pcapply(Z,T)            □ haspos(Y,T).
haspos(Y,offer(W,N,T)) :- ¬(Y=W),pcoffer(W,N,T)  □ haspos(Y,T).
haspos(Y,offer(W,N,T)) :- Y=W,pcoffer(W,N,T)     □ .
haspos(Y,hire(Z,W,T)) :- Y=W,freepos(Y,T)>1,pchire(Z,W,T) □ .
haspos(Y,hire(Z,W,T)) :- ¬(Y=W),pchire(Z,W,T)    □ haspos(Y,T).

%                EQUATIONAL CLAUSES

%                AXIOMS BETWEEN CONSTRUCTORS
apply(Z,T)  → T  :- ¬pcapply(Z,T)
offer(W,M,T) → T :- ¬pcoffer(W,M,T).
hire(X,Y,T) → T  :- ¬pchire(X,Y,T).

%                CLAUSES FOR FIRE
fire(X,Y,apply(Z,T)) → apply(Z,fire(X,Y,T))  :- pcapply(Z,T).
                                                pcfire(X,Y,apply(Z,T)).
fire(X,Y,offer(W,M,T)) → offer(W,M,fire(X,Y,T)) :- pcoffer(W,M,T).
                                                pcfire(X,Y,offer(W,M,T)).
fire(X,Y,hire(Z,W,T)) → T                    :- X=Z,W=Y.
                                                pcfire(X,Y,hire(Z,W,T)).
                                                pchire(Z,W,T).
fire(X,Y,hire(Z,W,T)) → hire(Z,W,fire(X,Y,T)) :- ¬(X=Z),
                                                pcfire(X,Y,hire(Z,W,T)).
                                                pchire(Z,W,T).
fire(X,Y,T) → T                              :- ¬pcfire(X,Y,T).
```

```
%                CLAUSES FOR FREE POSTS
freepos(Y,nil)        → 0.
freepos(Y,apply(Z,T)) → freepos(Y,T)   :- pcapply(Z,T).
freepos(Y,offer(W,N,T)) → freepos(Y,T)+N :- Y=W,pcoffer(W,N,T).
freepos(Y,offer(W,N,T)) → freepos(Y,T)   :- ¬(Y=W),pcoffer(W,N,T).
freepos(Y,hire(Z,W,T)) → freepos(Y,T)-1  :- Y=W,pchire(Z,W,T).
freepos(Y,hire(Z,W,T)) → freepos(Y,T)    :- ¬(Y=W),pchire(Z,W,T).

%                CLAUSES FOR PCAPPLY
pcapply(X,nil).
¬pcapply(X,apply(Z,T))                   :- X=Z,pcapply(Z,T).
pcapply(X,apply(Z,T)) → pcapply(X,T)     :- ¬(X=Z),pcapply(Z,T).
pcapply(X,offer(W,M,T))→ pcapply(X,T)    :- pcoffer(W,M,T).
¬pcapply(X,hire(Z,W,T))                  :- X=Z,pchire(Z,W,T).
pcapply(X,hire(Z,W,T)) → pcapply(X,T)    :- ¬(X=Z),pchire(Z,W,T).

%                CLAUSES FOR PCOFFER
pcoffer(W,M,T)  :- M>0.
¬pcoffer(W,M,T) :- ¬(M>0).

%                CLAUSES FOR PCHIRE
pchire(X,Y,T)          → pchire_(X,Y,T) :- freepos(Y,T) > 0.
¬pchire(X,Y,T).                         :- ¬freepos(Y,T) > 0.
¬pchire_(X,Y,nil).
pchire_(X,Y,apply(Z,T))                  :- X=Z,pcapply(Z,T).
pchire_(X,Y,apply(Z,T)) → pchire_(X,Y,T) :- ¬(X=Z),pcapply(Z,T).
pchire_(X,Y,offer(W,M,T)) → pchire_(X,Y,T) :- pcoffer(W,M,T).
¬pchire_(X,Y,hire(Z,W,T))                :- X=Z,pchire_(Z,W,T).
pchire_(X,Y,hire(Z,W,T)) → pchire_(X,Y,T) :- ¬(X=Z),pchire_(Z,W,T).

%                CLAUSES FOR PCFIRE
¬pcfire(X,Y,nil).
¬pcfire(X,Y,apply(Z,T))                  :- X=Z,pcapply(Z,T).
pcfire(X,Y,apply(Z,T)) → pcfire(X,Y,T)   :- ¬(X=Z),pcapply(Z,T).
pcfire(X,Y,offer(W,N,T)) → pcfire(X,Y,T) :- pcoffer(W,N,T).
pcfire(X,Y,hire(Z,W,T))                  :- X=Z,Y=W,pchire(Z,W,T).
pcfire(X,Y,hire(Z,W,T)) → pcfire(X,Y,T)  :- ¬(X=Z),pchire(Z,W,T).
```

3.3. CLP(H/E) operational model

The standard way to integrate narrowing and resolution [7] is to use narrowing to generate the solutions which are then tested by the logic program. This method does not fit in the CLP scheme and requires some complex interactions between backtracking of the narrowing and resolution algorithms. Instead, narrowing can be used as a procedure to test the consistency of the new constraint, looking for a solution, since the constraint is proved consistent once a solution has been found. At first glance, one can think about building a CLP(H/E) system by simply embedding a sound and complete semantic unification procedure, such as narrowing, as the constraint solver within the CLP system. With this naïve constraint solver, the transition relation $\rightarrow_{CLP(H/E)}$ becomes the standard definition of derivation of goals in equational logic programming [21]:

$$\frac{\text{there is an } E\text{-unifier of } c \cup \tilde{c}}{\langle \leftarrow c \square A_1, ..., A_n \rangle \rightarrow_{CLP(H/E)} \langle \leftarrow c \cup \tilde{c} \square B_1, ..., B_n \rangle}$$

if there exist n variants of clauses in P, $H_j \leftarrow c'_j \square \tilde{B}_j$, $j = 1, ..., n$, with no variables in common with $\leftarrow c \square A_1, ..., A_n$ and with each other and $\tilde{c} = \{c'_1, ..., c'_n, A_1 = H_1, ..., A_n = H_n\}$.

This definition involves too much redundant computation since a solvability test for the whole set of constraints is required at each transition and this set grows monotonically. required at each transition and this set grows monotonically. The problem is that the solvability test might require a significant cost. In [1,2], the matter of incrementally reusing the work done in the previous step to reduce the cost of solving the new set was adressed. A scheme based on reexecution and pruning of the search tree, together with several strategies to obtain an incremental narrower as constraint solver for canonical Horn equational theories, was presented in a

[4] Let us notice that, due to the (equational) approach to negation that we follow, preconditions are formalized by means of functions to make possible the definition and deduction of its falsity.

formal setup. For the sake of simplicity, in the following we will think of a computation as a sequence of reduction steps accumulating (simplified) satisfiable constraints.

A preliminary version of CLP(H/E) has been implemented. The interpreter is written in BIM-Prolog and runs on SUN workstations. We use the extension to the conditional case of the narrowing algorithm in [7] as the kernel of the constraint solver. Our implementation is only an approximation to the operational model of the language because we employ a *left-to-right* atom selection rule and a *top-to-bottom depth-first* search strategy in the generation of derivation sequences.

4. Using CLP(H/E) for running database specifications

Given an application, we are able to exhaustively interrogate and validate its conceptual scheme by running several kinds of experiments on data tests since both, parsing and generating mode [10,16,40], are directly available.

i) Parsing mode
In this mode, given an initial state S and an update sequence t, the matter of checking whether the sequence t is able to perform the transition from the state S to a final state T (characterized through its observable properties) and under what assumptions the final state is achieved can be addressed. To simplify the answer, an specialized information processor that simplifies and hides the informations related to the satisfiability of preconditions (when they hold) in the obvious way is assummed to exist. The goals below correspond to the guide example.

• Do we reach a state where the company $E2$ has free posts by an update sequence in that the company $E1$ offers 2 positions and then the company e offers N positions?

```
: □ haspos(E2,offer(e,N,offer(E1,s(s(0)),nil))).
% Answers:
                E2 = e, N > 0;
                E2 = E1, ¬ (E2 = e), N > 0;
                E2 = E1, ¬ (N > 0);
                no
```

• Do we reach a state where p is a candidate by an update sequence in that the person Z applies, next Y applies and then X applies ?

```
:- □ iscand(p, apply(X,apply(Y,apply(Z,nil)))).
% Answers:

  X = p;
  Y = p, ¬ (X = p);
  Z = p, ¬ (Y = p), ¬ (X = p);
  Z = p, ¬ (X = p), ¬ pcapply(Y,apply(p,nil));
  Y = p, ¬ pcapply(X,apply(p,apply(Z,nil)));
  Z = p, ¬ (Y = p),
       ¬ pcapply(X,apply(Y,apply(p,nil)));
  Z = p, ¬ pcapply(Y,apply(p,nil)),
       ¬ pcapply(X,apply(Y,apply(p,nil)));
  no
```

ii) Generating mode
There are many problems, from different fields, that can be formalized by means of the notion of system provided with a repertory of actions able to transform it by producing a transition of state. The plan-formation problem [28,33,41] can be formulated as the discrepancy, in a given system, between the starting situation and another one to be reached. The solution to the problem requires the achievement of a plan, which consists of the sequence of actions able to produce the required transition. The specification of a plan formation problem involves the description of the current state (i.e. the situation to be altered), the goal situation and the set of actions able to cause the transformation.

For the assessment of our approach, it is important to remind that the state of a system, at every single moment, can be represented in two different ways [34,39]:
• as the set of facts which hold in the state, which are asserted to the database after they have been infered.
• as the sequence of updates which are needed to "build up" the state and that serve to infer the above information when it is required

The former representation immediately results in the frame rule problem or how to handle the fact that many of the properties which hold in a given state still hold after an action is performed. It is well-known that this problem has a satisfactory solution in first order logic if the formalization of state change is able to describe the information deleted and added when an action is performed (by means, e.g., of metalevel constructs as *holds* and *not-holds* [28]). The solution is then based on the introduction of a frame (meta-) axiom which establishes that all properties *(P)* which are not invalidated by an action *(A)* are still true in the new state (*holds(P,result(A,S)) :- holds(P,S), preserved(A,P)*). According to this approach, in [16,40] the problem of using the axioms that define the database for synthesizing a valid sequence of update operations leading to the goal state was adressed. However, their approach requires the use of some specially designed tools (*expert helpers* [16]) just as it occurs in the proposal presented in [9] which, based on the second approach to represent a state, requires nevertheless the compilation of the specification into an internal model which allows the deduction of the insertions and deletions of information induced by the previous occurrence or absence of events.

In this subsection we address the plan-generation problem making use of equational constraint logic programming techniques. According to the constructive methodology developed in previous sections, it is the second way to represent states that we use. Thanks to this fact, the problem is solved without the use of any specially tailored strategy, simply by using the inherent capability of (constraint) logic programming to infer (constraint) answers. The methodology in [22] is closed to ours since a form of equational resolution is used to generate plans and the generated plans correspond to the well-known concept of computed answer (sustitution) in logic programming languages. In [22] both situations and plans are represented as terms whereas we consider the case when goal situations are represented as goal clauses.

In generating mode, computed answer constraints defining update sequences to take place in order for the database to be transformed from a given initial state T to a final state S can be found. The state S to be reached is (partially) characterized by a conjunction of interacting goals, expressing the facts that must hold or not hold in S. Interaction is guided by the partial order between function symbols which was implicitly established by means of the preconditions. If a constructor symbol is a candidate to participate in the required update sequence S, its preconditions will be added as constraints to the derived goal by the operational mechanism, if they are satisfiable. If an inconsistency results, no sequence is generated. We can put, e.g., the following goals:

• Obtain an update sequence S leading to a state where the person p works for the company c and c has free posts:

```
:-    □ worksf(p,c,S), haspos(c,S).
   % Answers:

S = hire(p,c,T), pchire(p,c,T), freepos(c,T)>s(0);
S = apply(Z,hire(p,c,U)), ¬ (p = Z),
      pcapply(Z,hire(p,c,U)), pchire(p,c,U),
      freepos(c,U) > s(0);
S = offer(c,N,hire(p,c,U)), pchire(p,c,U), N>0; ...
```

• Obtain an update sequence S leading to a state where the person p is a candidate and the length of S is lesser than 4:

```
:-      □  length(S) < s(s(s(s(0)))), iscand(p,S).
   % Answers:

S = apply(p,T), pcapply(p,T);
S = apply(Z,apply(p,T)), ¬ (p = Z), pcapply(p,T),
              pcapply(Z,apply(p,T));
S = apply(Z,offer(E,N,apply(p,T))), ¬ (p = Z),
              pcapply(p,T),
              pcapply(Z,offer(E,N,apply(p,T)));
S = offer(E,N,apply(p,T)), pcapply(p,T);
S = offer(E,N,apply(Z,apply(p,T))), ¬ (p = Z),
              pcapply(p,T), pcapply(Z,apply(p,T));
S = offer(E,N,offer(F,M,apply(p,T))),
              pcapply(p,T);...
```

Let us notice that the intensional representation, by means of constraints, of the set of answers to a goal has the advantage that, in general, different answers are subsummed by a single answer constraint. E.g., the first answer constraint

```
S = hire(p,c,T), pchire(p,c,T), freepos(c,T) > s(0)
```

subsumes the sequences

```
S = hire(p,c,offer(c,N,apply(p,nil))), N > s(0);
S = hire(p,c,apply(p,offer(c,N,nil))), N > s(0);
S = hire(p,c,apply(Q,offer(c,N,apply(p,nil)))),
          ¬ (Q = p), N > s(0);
S = hire(p,c,offer(c,N,apply(Q,apply(p,nil)))),
          ¬ (Q = p), N > s(0); ...
```

which, in equational logic programming, should be obtained, e.g., by means of a complete narrowing procedure as the answer to the goal `:- □ worksf(p,c,S), haspos(c,S).`

5. Conclusions

We have shown the usefulness of equational constraint logic programming techniques to support Database design. Two interesting modes for dealing with database specifications have been discussed and formally related with the aforementioned techniques. We are currently working on some extensions of the language to deal with disequality [36] and (real) arithmetics.

References

[1] M. Alpuente and M. Falaschi. Narrowing as an Incremental Constraint Satisfaction Algorithm. In *Proc. PLILP'91*, Passau, Aug 1991. volume 528 of *LNCS*, pages 111-122. Springer-Verlag, Berlin, 1991.

[2] M. Alpuente, M. Falaschi and G. Levi. Incremental Constraint Satisfaction for Equational Logic Programming. Technical Report TR-20-91, Universita di Pisa, Oct 1991. In *Proc. ILPS'91 Workshop on Defeasible Reasoning and Constraint Solving*, pages 47-75, San Diego, Ca., Oct. 1991.

[3] M. Alpuente and M. J. Ramirez. Rapid Prototyping of Database applications in the logic + equational EUROPA environment. In *Proc. 7th Int'l Conf. on Expert Systems, Theory and Applications*, pages 62-66, Los Angeles, Ca., 1990.

[4] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen and J. Managing, editors, *Handbook of Theoretical Computer Science, volume B*. North-Holland, Amsterdam, 1990.

[5] K. Balzer. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, volume SE-11 (2), 1985.

[6] M. Bellia and G. Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, 3:217-236, 1986.

[7] P. Bosco, E. Giovannetti and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3-23, North-Holland, Amsterdam, 1988.

[8] J. Bubenko and A. Olivé. Dynamic or Temporal modelling? An illustrative comparison. *SYSLAB Working paper 117*, Univ. of Stockholm, Sweden, 1986.

[9] D. Costal. An approach to Validation of Deductive Conceptual Models. In *Proc. 2nd Int'l Workshop on the Deductive Approach to IS and DB*, pages 50-72, Alguablava (Catalonia), 1991.

[10] C.S. dos Santos, T.S.E. Malbaum and A.L. Furtado. Conceptual Modelling of Database Operations. *Journal of Computer and Information Sciences*, 10(5):299-314. Plenum Press, NY, 1981.

[11] D. De Groot and G. Lindstrom, (Ed.). Logic programming, Functions, Relations and Equations. Prentice-Hall, 1986.

[12] N. Dershowitz and A. Plaisted. Equational Programming. Machine Intelligence, 11, pages 21-56. Clarendon Press, 1988.

[13] W. Dosh, G. Mascari and M. Wirsing. On the Algebraic Specification of Databases. In *Proc. 8th VLDB Conf.*, Mexico, 1982.

[14] H.D. Ehrich, K. Drosten and M. Gogolla. Towards an Algebraic Semantics for Database Specification, Knowledge and Data. In *Proc. IFIP WG 2.6 Conf. on Database Semantics*, Albufeira, 1986. North-Holland, Amsterdam, 1986.

[15] L. Fribourg. Slog: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. 1985 IEEE Int'l Symp. on Logic Programming*, pages 172-185. IEEE Computer Society Press, 1985.

[16] A. Furtado and C. Moura. Expert Helpers to data-based Information Systems. In *Expert databases systems*. Benjamin-Cummings, 1986.

[17] H. Gallaire, J. Minker and J.M. Nicolas. Logic and Databases: A deductive approach. In *Computing Surveys*, 16(3):153-185, 1984.

[18] E. Giovannetti and C. Moiso. A completeness result for E-unification algorithms based on Conditional Narrowing. In *Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 157-167. Springer-Verlag, Berlin, 1986.

[19] M. Gustafsson, T. Karlsson and J. Bubenko. A declarative approach to Conceptual Information Modelling. In *Information Systems design methodologies: A comparative review*, pages 93-142. North-Holland, Amsterdam, 1982.

[20] M. Höhfeld and G. Smolka. Definite Relations over Constraint Languages. LILOG-Report 53, IBM-Deutschland GmbH, 1988.

[21] S. Hölldobler. Foundations of Equational Logic Programming, volume 353 of *Lecture Notes in Artificial Intelligence*, subseries of *LNCS*. Springer-Verlag, Berlin, 1989.

[22] S. Hölldobler and J. Schneeberger. A new Deductive approach to Planning. *New Generation Computing*, 8:225-244, 1990.

[23] H. Hussman. Unification in conditional-equational theories. In *Proc. EUROCAL'85*, volume 204 of *LNCS*, pages 543-553. Springer-Verlag, Berlin, 1986.

[24] J. Jaffar and J. Lassez. Constraint Logic Programming. Technical report, Monash University, 1986.

[25] J. Jaffar and J. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111-119. ACM, 1987.

[26] S. Kaplan. Fair conditional term rewriting systems: unification, termination and confluence. In *Recent Trends in Data Type Specification*, volume 116 of Informatik-Fachberichte, pages 136-155. Springer-Verlag, Berlin, 1986.

[27] J.W. Klop Term Rewriting Systems. Tecnical Report CS-R9073, CWI, Amsterdam. to appear in *Handbook of Logic in Computer Science, volume I*.

[28] R. Kowalski. Logic as a Database Language. In *Proc. Advanced Seminar on Theoreticdl Issues of Databases*. Cetrano, Italy, 1981.

[29] G. Levi, C. Palamidessi, P. Bosco, E. Giovanetti and C. Moiso. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Proc. 1987 IEEE Int'l Symp. on Logic Programming* pages 318-327. IEEE Computer Society Press, 1987.

[30] J.W. Lloyd. Foundations of logic programming. Springer-Verlag, Berlin, 1987. second edition.

[31] J. Minker, (Ed.). Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, Los Altos, Ca., 1988.

[32] J.J. Moreno and M. Rodriguez-Artalejo. BABEL: A Functional and Logic Programming Language based on a constructor discipline and narrowing. In *Proc. ALP'89*, Volume 343 of *LNCS*, pages 223-232. Springer-Verlag, Berlin, 1988.

[33] N. Nilsson. Principles of Artificial Intelligence. Springer-Verlag, Berlin, 1982.

[34] A. Olivé. A comparison of the Operational and Deductive approaches to Conceptual Systems Modelling. In *Proc. IFIP'86*, pages 91-96. North-Holland, Amsterdam, 1986.

[35] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[36] M.J. Ramirez, M. Falaschi and G. Levi. Conditional Narrowing with Constructive Negation. In *Proc. GULP'92*, Milano, Italy, 1992.

[37] A. Sernadas, C. Sernadas and H.D. Ehrich. Object-Oriented Specification of Databases: an algebraic approach. In *Proc. 13th VLDB Conf.*, pages 107-116. Morgan Kaufmann, Los Altos, Ca., 1987.

[38] J.H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7:207-274, 1989.

[39] P. Veloso, J. Castilho and A. Furtado. Systematic Derivation of Complementary Specifications. In *Proc. 7th VLDB Conf.*, Cannes, 1981, pages 409-421. IEEE Computer Society Press, 1981.

[40] P. Veloso and A. Furtado. Towards simpler and yet complete formal specifications. *Information systems: Theoretical and Formal aspects*. North-Holland, Amsterdam, 1985.

[41] D.H. Warren. Warplan: A system for Generating Plans. memo 76, Univ. of Edinburgh, 1974.

Objects as Observable Processes

Isidro Ramos,Oscar Pastor,Jose Cuevas,Jaume Devesa

Departamento de Sistemas Informaticos y Computacion (DSIC)
Universidad Politecnica de Valencia
Camino de Vera S/N
Apartado 22012
46071 Valencia.- Spain
phone: +34 6 3877350
fax: +34 6 3877359
email: plo@dsic.upv.es

November 9, 1999

1 Introduction

Research on the integration of the OO and the deductive approach for designing Information Systems and Databases has been a major topic of interest in the last years. The OO and deductive environment represented by **Oasis** ([1] [2] [3] [5] [4] [9] [8]) as OO Specification Language, and **OO-METHOD** ([6] [7]) as a methodological extension covering the Analysis and Design Software Production phases within an Automated Programming Paradigm, is the result of a research effort tackled during the last two years in the DSIC-UPV.

This Oasis environment used to be formalized using first order theories that evolve in time. This lead us to a Logic Programming environment with the complement of a monitor to deal with dynamics. But we believe that, as it will be shown later, it has a natural, sound and complete formal basis within the context of Dynamic Logic.

Ramos, I., Pastor, O., Cuevas J., Devesa , J. Objects as Observable Processes. Proc. of the 4th International Workshop of the Deductive Approach for DB and IS.1993.

A new formalization of Oasis according to this view is presented in this paper, and a process algebra is defined in order to characterize properly objects as observable processes.

Accordingly, the Specification Language has been enriched with two new sections: transactions and processes, intended to improve the language expressiveness to make easier the conceptual modelling process.

In the next section, the new ideas about the formal background of Oasis are presented. In section 3 the example used to comment some concepts is introduced. Next, an Algebra of Processes for Objects is defined and finally, the properties of the new Oasis sections are stated.

2 Formalization of Oasis

A class in Oasis is made up of a class name, an identification function for instances (objects) of the class and a type or template that all the instances share.

The identification function characterizes the naming mechanism used by objects. It gives a set of surrogates belonging to a predefined sort or to a sort defined by the user (the so-called domains in Oasis). They are imported in the class definition.

A type is the template that collects all the properties (structure and behaviour) shared by all the potential objects of the considered class.

Syntacticaly, it can be formalized as a signature that contains the relevant class properties represented as:

- A set of sorts with a partial order relation. Among these set of sorts, we have in particular the sort of interest (the class name) associated with the class being defined.
- A set of functions with a particular one: the identification function whose sort is the sort of interest.
- A set of (constant, variable and derived) attributes. They all have as domain the sort of the class, and as codomain the given sort associated to the considered attribute.

- A set of events, with the sort of the class as domain (plus any additional sort representing event information) and with the sort of the class (sort of interest) as codomain

  As presented on the OO Analysis models of OO-METHOD [6], every event occurrence will be labeled by an actor who inititates it. Following the approach taken by [11] when dealing with this actor notion, we will write **i:a** if the actor i initiates event a, and call i:a an action. i could be the environment or any object of a system class.

  We do not conceive an event with no actor. When defining an event, the designer is forced to state who will be able to activate it. In consequence, we will assume defined a sort A of **actions**.

  In Oasis, the ideas of initiative of the event execution and visibility of events are separated by introducing the actor and view notions respectively. As we will show later, with the declaration of interfaces we represent which events are visible to which actors.

  It is also important to state that our work is based on the determinism of an action a.

and where the following kinds of formulas hold:

- **Evaluations**; they are formulas of the form $\Phi\ [p]\ \Phi'$ whose semantics is given by defining a ρ function that, from a process p returns a function between possible worlds.

  $$\rho(p) = W \rightarrow W'$$

  In other terms, being a possible world for an object any valid state, the ρ function precises which transitions between objects states are valid after the execution of process p.

  It is important to realize that, within this dynamic logic environment, the formula Φ is evaluated in W, and Φ' is evaluated in W', being W' the world represented by the object state after the execution of the considered process.

- **Integrity Constraints**; they are formulas that must be satisfied in every world. We distinguish between static and dynamic.

Static integrity constrainst are those defined for every posible world. They must always hold.

On the other side, dynamic integrity constraints are those relating different worlds. They require the use of a temporal logic, with the corresponding temporal logic operators.

- **Preconditions**, that are formulas with the template $\neg\Phi\ [a]\ false$, where Φ is a formula that must hold in the world previous to the execution of action a.

- **Triggers.** They are formulas of the form $\Phi\ [\neg a]\ false$. $\neg a$ is the action negation. It means that a does not occur, and it is not specified what does occur.

  Thus, triggers are actions activated when the condition stated in Φ holds. The main difference between preconditions and triggers comes from the fact that in triggers there is an obligation to activate an action as soon as the given condition is satisfied. Triggers allow in this way to introduce internal activity in the object society that is being modeled.

- **Processes**. The introduction of processes in our OO work is the major interest of this paper. In consequence, we are going to present this topic carefully.

3 An example

We will use as example an Information System intended to control the expenses of an organization. Money to expend is shared between several **programs**, each one with an assigned initial ammount of money for their expenses.

When a program is used to **contract** a **supplier**, the **available** money of the program is reduced. **Advanced payments** can be done to suppliers. A program can **advance** money to a supplier before the formal contract seal takes place. In consequence, the granted advanced payments and the corresponding supplier **balance** must be controlled.

When the contract is formalized, the corresponding payment is done unless the supplier had been prepaid, in which case the ammount of money involved in the contract is none.

An advance payment does not reduce the program's available money until the contract is signed. Then, the money paid in advance to the supplier is justified, shortening accordingly his/her balance.

Finally, the cash register book is managed storing **income** and **expenditure** occurrences, modifying properly the associated balance.

The specification in Oasis of this example is presented in the appendix. Shortly, we should commment some relevant aspects:

- When a program is created, it has an initial **available** ammount of money. If money is **restored**, the available money is incremented. When a supplier is contracted, it is decremented. When we write in Oasis the formula defining the attribute *available*, we are expressing in a clausal first order logic with equality framework the dynamic logic formula associated to the corresponding attribute evaluation.

  For instance:

  1. available(P,T) = available(P,T-1) - N :- contract(P,S,N,T).
  2. available(P,T) = available(P,T-1) + N :- restore(P,N,T).

- An integrity constraint for programs is stated as the prohibition of having an available ammount of money higher than the initial assignment. This formula must hold in every state.

  available(P,T) <= initial_assignment(P,T);

- A precondition for an event contract to happen is not to expend more money than it is available when the contract is going to occur.

  contract(P,S,N,T) :- available(P,T) >= N.

- In the example. the actor of any action is the environment, and for simplicity we do not declare it.

We will comment transactions and processes later on. We will remark then the global transactions (those that relate actions or transactions from different classes). For instance, when a supplier collects an advanced payment, an annotation is generated in the register book. These two actions will be encapsulated as a global transaction.

Other examples include the situation where a program makes an expense in the form of a supplier payment and the corresponding annotation must also be generated, or when program money is restored and this income of money is properly annotated.

4 Objects as Processes in Oasis

In previous version of Oasis, an object is defined as an observable process whose atomic action are **events**. As it has been stated, the occurrence of these events characterizes the evolution (animation) of the Information System (IS) being specified according to the preconditions associated to them.

Within this environment, a specification of an IS in Oasis is equivalent to a dynamic logic theory, that is translated with "animating" purposes to ([3]):

1. a first order theory whose models represent the state of any object.
2. a monitor to deal with dynamics. The monitor is used to implement the reachability relation between valid states (or, we could say between possible worlds over a Kripke structure).

The models of the quoted first order theory are used to evaluate the state (set of attribute values) of the objects in the corresponding world.

The expressiveness of Oasis would be improved in two ways:

- By introducing actions extending events with actors.
- By allowing the language to deal with transactions.

In this way, Oasis would allow to specify the objects dynamics not only by using events, but also giving the designer the choice of grouping events in execution units that we will call **transactions**.

These 'molecular' units that are the transactions will characterize the object dynamics following a 'all or nothing' policy with respect to their component events: if a fail event is activated during the execution of a transaction, the resultant state will be the initial one. We will precise this notion formally later.

The state of any object is given by the value of its state variables (the set of its attributes values). When an object is in a given state, the set of

potentially valid states from the initial one determine a set of possible worlds reachable for the object in its actual state. This accessibility relationship is determined by the process specification that is the object.

To enrich Oasis with this expressiveness, a process algebra is needed. We are going to define a process algebra for objects (APO) in Oasis, following the approach presented by Wieringa ([11]), but adapted to our deductive and OO Oasis environment.

Next, we introduce formally the APO. Later, the subsequent new Oasis paragraphs dealing with transactions and processes will be presented.

4.1 An Algebra of Processes for Objects

The Algebra of Processes for Object that we are going to introduce is an order-sorted process specification that has three ordered sorts [1]: EVENT $\leq$ TRANSACTION $\leq$ PROCESS, and the following set of operations:

$\_+\_$:process process $\rightarrow$ process (**alternative**)
$\_\bullet\_$: process process $\rightarrow$ process (**sequence**)
$\_||\_$: process process $\rightarrow$ process (**parallel composition**)
$\_\rtimes\_$: process process $\rightarrow$ process (**left-merge**)
$\_|\_$: event event $\rightarrow$ event (**communication**)
δ_{MESS}: process $\rightarrow$ process (**encapsulation**)
α_β: process $\rightarrow$ transaction (**transactions**)
δ: event (**deadlock**)
ϵ: event (**transaction fail**)
e:event (**event names**)

The specification is parametrized by:

- a set E of constants of sort event, called event names. They come from the event declaration section of a class specification in Oasis.
- a set MESS $\subseteq$ E of messages. Their elements are events that cannot occur by themselves: they cannot occur except as part of a communication, as we will precise later.

[1] We use the sort EVENT instead of ACTION to make clearer the presentation. We could introduce our APO with actions instead of events in a straightforward way

- the synchronization function γ: EVENT $\times$ EVENT $\rightarrow$ EVENT. It states that two events can occur at the same tick of the system clock, giving a new name to this 'shared' event occurrence.

  Synchronization plus encapsulation formalize the shared event notion used in Oasis.

The equations of the APO are given next. We have followed the introduction of a Process Algebra presented in ([11]), but adapted to the Oasis framework and including transactions.

Given the variables x, y, z: PROCESS; e, e1, e2: EVENT; the axioms are the following:

$$
\begin{aligned}
x + y &= y + x && (1)\\
x + (y + z) &= (x + y) + z && (2)\\
x + x &= x && (3)\\
(x + y)z &= xz + yz && (4)\\
(xy)z &= x(yz) && (5)\\
x + \delta &= x && (6)\\
\delta x &= \delta && (7)\\
x + \epsilon &= \epsilon && (8)\\
\alpha\epsilon x\beta &= \epsilon && (9)\\
\alpha x\epsilon\beta &= \epsilon && (10)\\
e_1 \mid e_2 &= e_2 \mid e_1 && (11)\\
(e_1 \mid e_2) \mid e_3 &= e_1 \mid (e_2 \mid e_3) && (12)\\
\delta \mid e &= \delta && (13)\\
e_1 \mid e_2 &= \gamma(e_1, e_2) \;\; if \;\; \gamma(e_1, e_2) = \top && (14)\\
e_1 \mid e_2 &= \delta \;\; if \;\; \gamma(e_1, e_2) = \bot && (15)\\
x \parallel y &= x \bowtie y + y \bowtie x + x \mid y && (16)\\
e \parallel x &= ex && (17)\\
e \bowtie x &= ex && (18)\\
ex \bowtie y &= e(x \parallel y) && (19)\\
(x + y) \bowtie z &= x \bowtie z + y \bowtie z && (20)
\end{aligned}
$$

$$
\begin{aligned}
e_1 x \mid e_2 &= (e_1 \mid e_2)x && (21)\\
e_1 \mid e_2 x &= (e_1 \mid e_2)x && (22)\\
e_1 x \mid e_2 y &= (e_1 \mid e_2)(x \parallel y) && (23)\\
(x+y) \mid z &= x \mid z + y \mid z && (24)\\
x \mid (y+z) &= x \mid y + x \mid z && (25)\\
\delta_{MESS}(e) &= e \; if \; e\neg \in MESS && (26)\\
\delta_{MESS}(e) &= \delta \; if \; e \in MESS && (27)\\
\delta_{MESS}(\alpha x \beta) &= \delta_{MESS}(x) && (28)\\
\delta_{MESS}(x+y) &= \delta_{MESS}(x) + \delta_{MESS}(y) && (29)\\
\delta_{MESS}(xy) &= \delta_{MESS}(x)\delta_{MESS}(y) && (30)
\end{aligned}
$$

$\top$ means defined and $\perp$ means undefined

4.1.1 Comments

The intuitive semantics is basically the same given by Wieringa to its ACP in [10]. We summarize here these semantics and introduce formally the notion of transaction.

Events are atomic, they are not divisible in other events. Time is not formalized in the axioms of the process algebra, but intuitively an event execution takes one discrete tick of the clock. A duration should be modeled by two events, the start and the end of the duration.

A transaction is a process delimited with the *alpha* (transaction start) and *beta* (transaction end) events. The main characteristic of transactions is that they represent 'all or nothing' actions: if the transaction fail (what is represented by the ϵ event occurrence), the object executing the given transaction remains in its initial state. In other words, the object must execute all the steps included in the transaction (i.e. all the changes of state induced by the events involved in the transaction), or remain unchanged.

The fail of a transaction is modelized as the substitution of the process events (excluding *alpha* and *beta*) by the event ϵ that lets the affected object in its initial state.

This transaction definition fits the classical transaction notion used by the database community, where any transaction execution is also delimited by the starting and ending events.

In the next section, we will precise formally this transaction notion by introducing and defining a ρ function, with processes as domain, and a function between possible worlds as codomain.

x+y is the process that has a choice between x and y, x•y is the process x followed by the process y, and x || y is the process consisting of the parallel execution of x and y. $\bowtie$ is an auxiliary operator, x $\bowtie$ y being the parallel execution of x and y, in which the first event is the first event of x. The • is often omitted when no ambiguity arises, and brackets are used to disambiguate terms. δ_{MESS} is another auxiliar operator explained below, and δ is a special constant interpreted as *deadlock*. After δ nothing can happen.

The non-conmutativity of • formalizes the unidirectionality of the passage of time. Additionally, the absence of the axiom $x(y + z)$ formalizes the sensitivity of the equals sign for nondeterminism. $x(y + z)$ is not equal to $xy + xz$: the first process is deterministic, the second nondeterministic. We call the choice $x + y$ arbitrary because it is not specified which branch is chosen (is an underspecification).

x || y is the interleaving of x and y, having a choice between executing the first event of x, the first event of y, or a synchronous execution of a first event of both processes. For instance

$$x = (e_1 \bullet e_2) \parallel e_3 \text{ can be rewritten to:}$$

$$x = e_1e_2e_3 + e_1e_3e_2 + e_3e_1e_2 + (e_1 \mid e_3)e_2 + e_1(e_2 \mid e_3)$$

Communication is modeled distinguishing parallel composition from encapsulation of processes. When we have a term $e_1 \mid e_2$, according with the definition of γ this term is rewritten as an element of the Event sort, or as δ. The intuitive meaning of $\gamma(e_i, e_j)$ is that e_1, e_2 can occur synchronously, i.e. at the same click of the global clock. If $\gamma(e_1, e_2) = \perp$, $e_1 \mid e_2$ is a deadlock.

Encapsulation is achieved by using the δ_{MESS} operator. If MESS=$\{e_1, e_2\}$, then $\delta_{MESS}(x)$ is a process in which e_1 and e_2 cannot occur except as part of a communication.

So, γ defines which events **can** occur synchronously, and MESS defines which events cannot occur by themselves, they **must** occur as part of a communication. This mechanism (communication + encapsulation) formalize the shared event notion of Oasis.

The specification of the operations + and • with the axioms $1 \ldots 5$ is called a *Basic Process Algebra (BPA)*. Extended with the δ and ϵ constants and the equations $6 \ldots 9$ is called $BPA_{\delta\epsilon}$.

It can be demonstrated that APO is a conservative extension of $BPA_{\delta\epsilon}$ ([10]). This means that:

1. APO is a consistent extension of $BPA_{\delta\epsilon}$: given two closed terms $t_1, t_2 \in BPA_{\delta\epsilon}$ $APO \vdash t_1 = t_2 iff BPA_{\delta\epsilon} \vdash t_1 = t_2$

2. APO is a complete extension of $BPA_{\delta\epsilon}$: given a closed term $t_1 \in APO, \exists t_2 \in BPA_{\delta\epsilon}$ *with* $APO \vdash t_1 = t_2$

In consequence, every closed term of APO can be rewritten as a term containing only the + and • operations.

4.2 Formal notion of transaction

We start from a set E of events, whose members come from the event paragraph of the given Oasis specification. Intuitively, associated to every event e ∈ E, it exists a function ρ_0 that characterizes the change of state (or the transition between possible worlds) induced by e.

Definition 1 *Given a set of events E, and the set of possible worlds W, we define*

$$\rho_0 : E \rightarrow (W \rightarrow W)$$

ρ_0 is an atomic transition function defined for events. It is fixed by the evaluation formulas of the specification, where the effects of every event on their relevant attributes (those affected by its occurrence) are precisely specified according to the deductive dynamic rules that characterize the expressiveness of Oasis.

This definition is extended to transactions and processes in a natural way.

Definition 2 *Given $p \in APO$, and $\alpha, \beta \in E$, a transaction is defined as a term of the form*

$$\alpha\_\beta : p \rightarrow transaction$$

A transaction is a process delimited by the α (start transaction event) and β (end transaction event) that constitutes an execution unit. If the transaction fails before reaching the end, the resultant world must be the initial one.

Definition 3 *Given $\underline{e} \in BPA_{\delta\epsilon}$ that represents a transaction body* [2] *and being P the sort of processes specified in $BPA_{\delta\epsilon}$, we define the transaction transition function ρ_0^*:*

$$\rho_0^* : P \to (W \to W)$$

with the following axioms:

$$\begin{aligned}
\rho_0^*(\underline{e} + \underline{e'}) &= \rho_0^*(z) \;\; where\; z = choice(\underline{e}, \underline{e'}) \\
\rho_0^*(\underline{e} \bullet \underline{e'}) &= \rho_0^*(\underline{e'})(\rho_0^*(\underline{e})) \\
\rho_0^*(\underline{e}) &= \rho_0(\underline{e})\; if\; \underline{e} \in EVENT \\
\rho_0^*(\delta) &= \bot \\
\rho_0^*(\epsilon) &= I
\end{aligned}$$

The first axiom presents a non-deterministic choice, the second introduces a functional composition between possible worlds, and the last one characterizes the identity function I that correspond to the null transition induced by the ϵ event.

The next step is to formalize the non observability of the intermediate states of a transaction. Intuitively, the idea is to restrict the definition of the observation function to the final transaction state.

Definition 4 *Given a set A of cardinality n of attributes, a set of classes V_j (with $j : 1 \ldots n$) associated to them, and a possible world w_i, we define the observation function associated to w_i as:*

$$\models_{w_i} \;\; \forall j : 1 \ldots n, \exists a_j \in A, \exists v \in V_j / a_j = v$$

Definition 5 Non observability for intermediate transaction states*: Given the trace $w_1, w_2, \ldots, w_k$ of possible worlds followed by a transaction execution, it holds that:*

$$\forall i : 1, \ldots, k, and\, \forall j : 1, \ldots n, \;\; \not\models_{w_i} (\exists a_j \in A, \exists v \in V_j / a_j = v)$$

[2] We call transaction body the part of a transaction resulting when leaving the starting and ending (α, β resp.) transaction events.

5 Transaction and Process Specification in Oasis

According to the previous sections, Oasis has been enriched with two new paragraphs:

- a transaction specification paragraph to define the potential transactions executed by objects of any class
- a process specification paragraph to allow designers to define properly the process that characterizes every System object.

Within this process section, the term of the APO associated to every class is defined and the subsequent valid states for the instances of the class are stated as a result of this process specification.

The transactions specified in the corresponding transaction paragraph can be used to fix this APO term.

In the example we are using, after a program has been created, advanced payments, contracts or restoring events can be activated in any order. It must be taken into account that before the first money reposition, at least an expense must have been activated.

- PROGRAM = new_PROGRAM(P,T) PROGRAM0
- PROGRAM0 = advance_payment(P,R,N,T) PROGRAM0 + PROGRAM1;
- PROGRAM1 = contract(P,S,N,T) (PROGRAM1 + PROGRAM2)
- PROGRAM2 = advance_payment(P,R,N,T) PROGRAM2 + PROGRAM3;
- PROGRAM3 = restore(P,N,T) (PROGRAM1+destroy_PROGRAM(P,T))

Furthermore, there is a global transaction paragraph to define the transactions that involve several system classes. These global transactions are viewed in the Oasis framework as transactions of the paralel composition of the classes that are related due to the transaction specification.

For instance, when a supplier collect money of an advanced payment, an annotation must be done in the register book. This is expressed by defining the global transaction

EXPENDITURE_PROPOSAL = advance_payment(P,R,N,T)
ANNOTATION

where *advance_payment* is a supplier and program shared event and $ANNOTATION$ is the process associated to the annotation class.

We have a similar situation when a contract program event occur. There are two possibilities: the action of contract with annotation because the supplier is paid when the contract is done, and the action of contract when the supplier was paid in advance. In this case, a justify event must follow the contract occurence, what is encapsulated in the so-called justification transaction.

PAYMENT = contract(P,R,N,T) ANNOTATION + JUSTIFICATION

6 Conclussions

A enriched version of OASIS has been presented, including the capability of specifying transactions and processes. In this way, objects are viewed as observable processes within a logic dynamic framework, keeping the deductive properties of the OASIS expressiveness and, at the same time, formalizing the procedural view of objects by introducing an Algebra of Processes for Objects.

The tools and methods of the OO-METHOD methodology developed at DSIC-UPV are being updated to include from the earlier Software Production phases this added expresiveness, keeping the methodology faithful to the Automated Programming Paradigm within an OO model.

From the specification sake, current work is focused on introducing carefully in the OASIS Specification Language a Temporal Logic to deal with dynamic integrity constraints and giving the designer the chance of defining views to fix the user-system interface. The declaration of such a view notion would allow:

1. To give the user a virtual hierarchie fitting his needs.
2. To hide some class properties.
3. To modify structure and behaviour of objects.
4. To view an object from different perspectives.

References

[1] Canós,J.H.:Pastor,O. *Object Oriented and Functional Specification of Information Systems* in Proc. of DEXA-91 Conference, Berlin,1991

[2] Pastor,O.:Hayes,F.:Bear,S. *OASIS: An Object Oriented Specification Language* In the Proceedings of the CAiSE-92 Conference, Springer-Verlag, pag.348-363; Mayo 1992, Manchester (UK)

[3] Pastor O. *Diseno y Desarrollo de un Entorno de Producción Automática de Sotfware basado en el modelo OO* PhD thesis, DSIC-UPV, May 1992.

[4] Pastor,O.:Ramos,I:,Canos,J.H. *Object Oriented and Relational Specification Of Information Systems* In the Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS, Aiguablava (Catalonia), 1991

[5] Pastor,O;Casamayor,J.C. *A Deductive and Object Oriented Environment for Software Production* In the Proc. of the Workshop in Expert Systems, AI and Software Engineering Application, ILPS-91, San Diego (California).

[6] Pastor,O.;González, A. *OO-METHOD: An Object-Oriented Methodology for Software Production* n the Proceedings of the DEXA-92 Conference, Valencia (Spain)

[7] Pastor,O.;Ramos,I.;Canos,J. *From Analysis to Design in an OO and deductive environment* Accepted for presentation in DEXA-93 Conference, Prague.

[8] Ramos,I. *Logic and OO Databases: a Declarative Approach* Proc. of the DEXA 90, Springer-Verlag 1990

[9] Ramos,I:,Pastor,O.:Canos,J.H. *On the use of Algebras as Semantic Domains of Object Societies* In the Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS, Roses(Catalonia), 1992

[10] Wieringa R.J. *Algebraic Foundations for Dynamic Conceptual Models* PhD thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1990.

[11] Wieringa R.J.;Meyer J.-J.Ch. *Actors, Actions and Initiative in Normative System Specification* Annals of Mathematics and Artificial Intelligence, 1992.

610

From Analysis to Design in a Deductive and Object-Oriented Environment

Oscar Pastor López, Isidro Ramos, José H. Canós
Departament de Sistemes Informatics i Computació
Universitat Politecnica de Valencia. e-mail: {plo, iramos, joseh}@dsic.upv.es

Abstract. In this paper we present an Object-Oriented (OO) methodology for software production, and we focus on the analysis to design transition. OO-METHOD provides software tools covering the classical development phases of analysis, design and implementation, all this within a graphical, user-friendly working environment. Starting from a graphical system specification as the product of the OO analysis step, we produce the corresponding system specification in OASIS (a formal and OO specification language used as design tool). Finally, depending on the chosen working environment (logic or OO), we can obtain a final software product automatically by using the corresponding translator.

1 Introduction

To solve the *Crisis of Software*, we aim to change the methodological perspective by developping a Software Production Environment that focus on:

1. Life cycle modification via the introduction of rapid prototyping and automated programming [1].
2. The use of formal specification languages, in a deductive approach [2].
3. The use of an OO model.
4. The development of advanced graphical working environments.

In this paper we present an OO Methodology for Software Production (OO-METHOD), and we focus on the analysis to design transition. This OO system view is maintained throughout all the phases of the software life cycle provided by OO-METHOD.

The idea of blending the Object-Oriented and the Logic Programming paradigms within an automated programming life cycle and a deductive approach is within the basis of our proposal.

Section 2 on this paper constitutes its core, and introduces the main features of OO-METHOD, focusing on the transition from the analysis to the design, and stating how to obtain the resulting OO specification in OASIS. The basic concepts of the underlying OO model that we use can be found in [8, 6].

2 The methodology

OO-METHOD generates two OO graphic specifications: one dealing with structural system aspects and the other dealing with the behavioural counterpart. In

Pastor, O., Ramos, I., Canós, J.H. From Analysis to Design in an OO and Deductive Environment. Proc of the International Conference on Database and Expert System Applications (DEXA´93). 1993.

[6, 7] the analysis phase in OO-METHOD is presented in depth. The products resulting from the OOA step must be translated into an OASIS specification. OASIS [5] is a formal and OO Specification Language, used by OO-METHOD as its OO-design tool (see 2.2). Within a logic programming environment, and with an OASIS specification being equivalent to a formal dynamic logic theory [8, 4, 7, 11], the animation of the logic program that represents this theory constitutes the system implementation. This logic program is obtained directly from the OASIS specification by means of a translator. In fact, we automatically produce the final implementation from the graphical specifications built in the analysis phase.

An interesting extension is accomplished when the chosen implementation tool is an OO Programming Language (OOPL). The final goal is to provide an OO-CASE tool covering the development phases of analysis, design and implementation with a high level of automation, and within an OO and deductive framework.

In the remainder of this work , we use the following example. Let's consider a (simple) hospital consisting of doctors, patients and individual rooms. When a patient goes into hospital, (s)he occupies an available room. During his/her stay, one or more doctors look after him/her. Doctors provide services, labelled with a date and a price. When patients leave the hospital, an invoice is made up of their associated set of services.

2.1 OO Analysis

OO-METHOD proposal. Object-Oriented Analysis (OOA) emphasizes the building of real-world models, using an object-oriented view of the world. We define OOA from a twofold view: first as a *classification* process for identifying which classes compose our System Domain and which relationships exist between them. Second, as a process of identifying which are the *available states* for the detected classes, and how *changes of state* can be produced (i.e. which *events* will be relevant for a given class). Fulfilling an OOA has three main advantages over the classical structured analysis methods ([3, 10]).

1. We have the same model dealing with both the structural, static system aspects (data) and the behavioural, dynamic ones (processes).
2. We are using a model closer to the working of human cognition, consisting basically in capturing our understanding of the problem domain by means of a classification process.
3. Within an OO framework, the boundaries between analysis and design are somewhat fuzzy, dealing in both cases with classes, objects and their relationships and reducing the gap from analysis to design. Design is just a step along the way in the incremental, iterative development of a System that OO approaches provide.

The results of our OO-METHOD analysis process [6] will be the generation of two kind of graphics:

1. a *Class Configuration Diagram* (CCD). It is an extended semantic model showing the static system structure but also including the relevant events for every class.
2. for each class in a CCD, a *Classchart* (CCh) that represents its available states and the events that will cause the transition from one state to another.

This section is closed by presenting the CCD corresponding to our example and the CCh corresponding to the room class (figure 1). In (a), the system component classes are represented graphically (`doctor`,`patient`,`room`) with their shared events (`give-service`, `leave-room` and `occupy`) and a many-to-many aggregation relationship between doctors and patients labelled `services`.

(a) **(b)**

Fig. 1. Class Configuration Diagram (CCD) and Classchart (CCh) corresponding to the `room` class of the Hospital example

In (b), two potential states are considered for the `room` class (to be `available` or `engaged`), representing the events that can change any room object state (`occupy` , `leave-empty`).

2.2 From Analysis to Design : OO Design

OASIS is an OO specification language [5, 9] that provides a set of syntactic constructs to declare the classes that constitute the system domain viewed as a society of interacting objects. Briefly, OASIS has three main constructs:

- *domains* (primitive classes), used as object surrogates and attribute classes.
- *elementary classes*, and
- *complex classes*, those defined using the set of *class operators* provided by OASIS: aggregation, inheritance, association and parallel composition.

Every class has a set of *attributes* (state variables), *events* (private or shared class messages sent by agents), *preconditions* (admissibility laws associated to events) and *triggering relationships* (actors implementation).

Next, we map the graphical diagrams coming from the previous OOA phase into OASIS constructs, showing how we can obtain the corresponding OASIS specification in an automated way.

Starting from the Class Configuration Diagram. Every elementary class is transformed into an OASIS elementary class. Inheritance relationships introduce specialization, declared in OASIS by means of `using` statements. The situation with the aggregation relationships is richer: depending on the cardinality associated to the aggregation we have different design time choices. A one-to-one relationship's cardinality will produce an object-valued attribute either in any one of the involved classes or in both. A one-to-many relationship can be dealt with

1. by transforming the one side's class into an OASIS grouped complex class, or
2. by creating a new aggregated complex class in addition to the two related classes. This is the case when the considered relationship has attributes and/or events associated with it.

With the many-to-many relationships we have a similar situation.

Completing the Oasis Specification with the Classcharts. At this moment we have a pattern of the Specification that is lacking information about the System behaviour in terms of variable attribute definition, preconditions associated with class events and triggering relationships. We get this information from the CCh associated to every class.

Variable attribute definition. In OO-METHOD, there are four categories of attributes depending on how their relevant events modify them.

1. *Boolean-valued* attributes checking if the object is in a given state, reached by an `event1` occurrence and left by another `event2` occurrence. For an attribute `a` :

```
attribute(a,t)=b if (event1(a,t) and b=true) or
(event2(a,t) and b=false) or
(t1=t-1 and attribute(a,t1)=b)
```

   An example is the `available` attribute of the `room` class.
2. *Push-Pop* attributes, that have an `event1` *incrementing* their value and an `event2` *decrementing* it. We would have:

```
attribute(a,t)=n if (t1=t-1 and attribute(a,t1)=n1) and
(event1(a,n',t) and n=n1+n') or
(event2(a,n',t) and n=n1-n'))
```

The Doctors' attribute `number-of-services` is an example of this kind. It has an incremental event (`give-service`) and no decremental event.

3. *State-valued attributes*: those attributes whose value is given by the event that has brought the object to the given state:

```
attribute(a,t)=v if (event1(a,v',t) and v=v') or
(t1=t-1 and attribute(a,t1)=v)
```

   For example, if an event `occupy` of a Patient class leads a patient instance to the *in-hospital* state, a state-valued attribute would be the associated room number, an integer parameter of the `occupy` event.
4. Mixed attributes, defined in terms of the previous types.

Preconditions. The CCh associated to every class includes information about state transitions caused by the occurrence of events. This representation states a first component for the precondition associated to the event: the affected object must be in the source state of the event transition state. Additional components appear in the transition's label after the *if'* symbol, stating other conditions that must hold for having a correct event occurrence.

Triggering relationships. Associated with any event label, every CCh includes information about which other events may be triggered as a consequence of the state transition. This will produce the following OASIS sentence:

```
object::trigger-event if event1
```

where `object` is the addressee, `trigger-event` is its triggered event and `event1` is the actual event causing the change of the object's state. To close this section the specification in OASIS for the `doctor` class of our simple Hospital example is given:

3 Conclusions

OO-METHOD provides a software production environment with the following relevant aspects:

- It starts from an OO system view that develops basic Information System concepts, according to the most recent Information Systems concepts.
- A graphical notation to represent the analysis products is given. This notation includes static and dynamic graphical models that constitute a clear specification of the system to be designed.
- Within the OO approach used by OO-METHOD, the transition from analysis to design can be done in an automated way. The chosen OO specification language is OASIS, and the specification in OASIS corresponding to the Analysis products is directly obtained.
- As a specification in OASIS is equivalent to a dynamic logic theory (clausal or equational), the logic program equivalent to the specification constitutes an executable system prototype obtained directly from the specification.
- If our working environment is based on the use of OO Programming Languages, an advanced pattern of the final software product can be obtained in an automated way.

At the moment, an OO-METHOD implementation is being tested in several large software projects to check its real capacity as a software production methodology.

References

1. R. Balzer et al.:*Software Technology in the 1990s: Using a New Paradigm*. IEEE Computer Science Press,Nov.1983.
2. J. A. Bubenko, A.Olivé: *Dynamic or Temporal Modelling? an illustrative comparison*. SYSLAB Working Paper 117,Nov.1986.
3. C.Gane,T.Sarson: *Structured System Analysis*. Englewood Cliffs,NJ:Prentice Hall,1979
4. O.Pastor,I.Ramos,J,Canós: *Object Oriented and Relational Specification of Information Systems*. Second International Workshop on the Deductive Approach to Information Systems and Databases. Aiguablava,Sept.1991.
5. O.Pastor,F.Hayes,S.Bear: *OASIS: An Object-Oriented Specification Language*. In the Proceedings of the CAiSE-92, Manchester,UK.
6. O.Pastor,A.González: *OO-METHOD: An OO Methodology for S oftware Production*. Proceedings of DEXA-92, Valencia
7. O.Pastor. *Diseno y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo OO*. PhD Thesis, U.P.V., Valencia 1992.
8. I.Ramos: *Logics and OO-Data Bases: a declarative approach*. Proceedings of DEXA-90, Springer-Verlag,1990 pag. 348-363.
9. I.Ramos,O.Pastor: *OO and Active Formal Information System Specification*. Proceedings of DEXA-91, Berlin (1991)
10. E.Yourdon: *Modern Structured Analysis*.Englewood Cliffs,NJ:Prentice Hall,1989
11. R.J. Wieringa, *A formalization of objects using equational dynamic logic*, Proc. of the 2nd DOOD Conference, Springer-Verlag, 1991.

This article was processed using the LaTeX macro package with LLNCS style

Prototyping Object Oriented Specifications in an Algebraic Environment[1]

A. Toval(*), I.Ramos(**), O.Pastor(**)

(*)Dept. de Informática y Sistemas. Facultad de Informática. Universidad de Murcia.C/Sto. Cristo, 1 (30001) Murcia (Spain); Fax. + 34-68-835418; e-mail atoval@dif.um.es

(**) Dept. de Sistemes Informatics i Computació (DSIC). Universitat Politecnica de Valencia. Camí de Vera, s/n. E-46071 Valencia (Spain). e-mail {iramos,plo}@dsic.upv.es

Abstract.- This paper presents an algebraic environment aimed to achieve operational prototypes [3] from an object-oriented description of the Universe of Discourse (UoD). Based on the OASIS object-oriented specification language [15] and the works initiated in [16], two algebraic specifications are provided: the first formalizes the OASIS language; the second is a formalization of the lives of the objects in the UoD. From an OASIS description of the conceptual scheme, the object-oriented algebraic prototyper (OOAP) automatically generates a validated ground term of the term algebra of the OASIS specification. After a consistency analysis OOAP generates an OBJ3 prototype for that specification which is internally executed as a rewriting system. It is then ready to be used by developers and users in order to validate and verify functional requirements of the information system. An implementation of the OOAP has been carried out using the functional language OBJ3.

Keywords.- algebraic specifications, conceptual modelling, object-oriented prototyping, object-oriented specification.

1 Introduction

The objective of our work is to design and implement a prototyping environment called OOAP (Algebraic Prototyper) based on a formal framework to achieve object-oriented functional prototypes [3]. This environment makes possible the simulation of the operational characteristics of the system being developed, starting from an OASIS description based on the object-oriented (shortly OO) paradigm; OASIS [15] is a language for open and active information Systems specification. In previous works we investigated these issues starting from Structured Analysis models [14], but we have found formal descriptions much more promising than structured models [20] [19], as we stated in [16] which initiated the present work.

The approach used for our work evolved from mainly integrating two formal specification techniques: firstly the well-known algebraic specifications theory [5], for the formalization of both the OASIS object-oriented specification language, and the Information Base, that is to say the lives of the objects in the UoD. The OASIS algebraic specification allows the formal representation of any conceptual scheme as a ground term of the corresponding ground terms set; the Information Base formalization allows us to represent the life cycles of the objects and provides operations for

[1] The research reported in this paper has been partially supported by the Spanish Ministries of Education and Industry, CICYT project PROTESIS TIC 91-0425 (1991/1993)

Toval, A., Ramos, I. , Pastor, O. Prototyping Object Oriented Specifications in an Algebraic Environment. Proc. Of the International Conference on Database and Expert System Applications (DEXA'94), 1994.

observing the system; secondly, we adopt the formalization of the object-oriented paradigm, trying to put into the practice the ideas in the literature, in particular those proposed by Sernadas and Ehrich [17] [4] [10] [11].

Section 2 describes the formal object-oriented model that we adopt for our work, together with the most significant features of OASIS. Section 3 briefly introduces OBJ and sections 4 and 5 describe in more detail the kernel and information base formalization. A Library example illustrates the ideas presented.

2 A Proposal for an Object-Oriented Formal Model

Currently, most approaches to object-oriented conceptual modeling are rather informal [2] [13], although new formal approaches are emerging, including those for FOOPS [7], CMSL [23] or the above quoted references from Sernadas and Ehrich. In this section we review the notions of object, class and life-cycles that we adopt.

2.1 Basic OO Formal Concepts

Object.- Objects are members of classes; the class has a template of its members, specifying their common structure. Summing up, an object consists of an oid (object identifier) and the template $ob = (X,A,T,\alpha)$, where X is a set of events, A is a set of attributes, $T \subseteq X^{\sigma}$ is a set of life cycles ($X^{\sigma} = X^{*} \cup X^{w}$ denotes the set of finite and infinite streams over X). Finite sequences are called traces over X. A valuation mapping , $\alpha : X^{*} \rightarrow obs(A)$, maps each trace in X into a valuation of the attributes in A . For each attribute a in A we assume a data type, type(a), determining the values a can have. We define an observation over A as a set of attribute-value pairs, $y \subseteq \{(a1:d1),...(a_n:d_n)\}$, where a_i is in A and d_i is in $type(a_i)$ for $i=1...n$. obs(A) denotes the set of observations over A.

Object communication and behavior.- Object interaction in this model corresponds to event sharing (message passing) and triggering relations. The history of any object is reflected in its life cycle which is a sequence of permissible events. Other approaches consider only one sequence representing all the object lives cycles interleaved, but we prefer to separate life cycles for each object. This separation will facilitate concurrence considerations and modelling. The use of preconditions ensure that only permitted events will ocurr to a particular object, depending on its structure and current trace. Section 5 shows an example of life cycles for the Library example.

Class Hierarchies.- Classes can be related to each other in the form of hierarchies. OOAP allows that collections of objects be arranged using the well-known class operators generalization/ specialization, aggregation/ projection and classification/ instantiation. In addition, OOAP incorporates parallel composition of objects as well as the appropriate mechanisms to make easy the introduction of new, user-defined, class operators (extensible operators) by reusing built-in class operators and primitive operations of the underlying algebraic specifications.

2.2 The OASIS Specification Language

OASIS [15] is a language for open and active information systems specification into an object-oriented paradigm, developed in the DSIC. We omit from this paper a detailed description of OASIS, but we shall introduce it here rather informally, illustrating how the formal concepts previously defined are considered in OASIS. The concept of template of a class consists of a set of (constant and variable) attributes, variable attributes depend on the ocurrence of events; a set of (private and shared) events, private events have an effect only on objects of the class, while shared events involve objects of two or more classes; process aspects are modelled by the use of preconditions and the notion of agent is described by triggering relations. These concepts are considered in the next straightforward example, for a class named "Borrower" in a library information system, which also contains classes "Book" and "Loan", not detailed here:

```
conceptual scheme Library
                              /* Primitive classes
      domains: nat, bool, string, time
                              /* Elementary classes
      elementary class Borrower
            attributes
            constant
                  key id-number:nat
                      name: string
                      age: nat
                  static constraints  16 < age < 100
            variable T: Time, L: Book, B: Borrower
            books-on-loan (Borrower,Time) : nat
            formulae
            books-on-loan(B,T)=books-on-loan(B,T-1)+1 if borrow(L,B,T)
            books-on-loan(B,T)=books-on-loan(B,T-1)-1 if return(L,B,T)
            books-on-loan(B,T) = 0  if  join(B,T)
            events T: Time, L: Book, B: Borrower
                  private new       join (B,T)
                          destroy   remove(B,T)
                                    notify(B,T)
                  shared            borrow(L,B,T)
                                    return(L,B,T)
            preconditions T: Time, L: Book, B: Borrower
                  remove(B,T) if books-on-loan (B,T) = 0
                  borrow(L,B,T) if books-on-loan (B,T) < 5
                  return(L,B,T) if books-on-loan (B,T) > 0
            triggering T: Time, B: Borrower
                  self:: notify(B,T) if books-on-loan(B,T) = 4
      end elementary class

      elementary class Book   ............end elementary class
                               /* Complex classes
      complex class  Loan aggregation of Borrower, Book
          ..................end complex class

end conceptual schema
```

3 Algebraic Specifications and OBJ3

We assume the theory of order-sorted equational specification [6] [8] which extends the many sorted-case [5].

Algebraic specifications can be animated by means of a rewriting system, using an

appropriate algebraic specification language. Besides the criterion of availability, we found in the last release of OBJ3 [9] a number of advantages that encouraged us to use it as the support for developing our system: OBJ3 is based upon order-sorted equational logic supporting exception handling, overloading, multiple inheritance (not in the sense of object-orientation, but in the sense of subsorting), parameterised programming, module expressions,etc. The release used has an acceptable speed for reductions and the environment, although not yet very friendly, is sufficiently accessible. In designing some of the components of OOAP, in particular the user interface, we have also taken advantage of the OBJ3 facility for encapsulating Lisp code, augmenting the system with new capabilities in connection with the formal theory. Details of the OBJ3 language are given in [9].

4 OOAP Kernel: The OASIS Algebraic Specification

OOAP provides a manageable formalization of classical object-oriented concepts such as inheritance, classification, precondition, the notion of event,etc... using an algebraic framework. This is realized by the kernel of OOAP: an algebraic specification, named AS-OASIS=(S,OP,E), (being S a set of sorts, OP a set of operation and constant symbols and E a set of equations describing the semantics for the signature (S,OP)). The formal definition of the system, the prototype and the life cycles of the objects belong to the same algebraic framework, so ensuring maintenance of the good formal properties. By convention, in OBJ3, module identifiers are all upper case. A full description of all the OBJ3 modules used in this paper can be found in [21]; in particular AS-OASIS should contain, among other, the following elements:

```
obj  AS-OASIS is sorts Cs DomDef .
      protecting CC-LIST .

*** HEAD (constructors)
      op conceptual schema : ClassId* DomDef EcList CcList → Cs .
      op domains : TermList →DomDef .
***  some non-constructor
      op cclassid- : ClassId Cs → Class .
            *** Error Handling
      op Error. _ is not a class : ClassId → Ec .
      ........
*** some non-constructor (queries)
      op ckey : Class Cs → CAttribList .
      op cnokey : Class Cs → CAttribList .
      op cevents : Class Cs → EventList .
      ........ axioms ......
endo
```

The sort Cs names all the possible conceptual schemata, ClassId* is the sort of names for conceptual schemata while ClassId is used for the rest of classes, DomDef is the sort of the primitive classes (ADT's) and EcList, CcList respectively denote collections of elementary and complex classes. We also introduce sorts Ec and Cc describing elementary and complex classes which are built in a similar way as Cs. While Ec has only one possible constructor (named "elementary class"), Cc has a

variety of them: one for each kind of class composition operation (specialization, aggregation,...) and also the one named "defined class" for user-defined operators.

"Protecting" is a module importation mode of OBJ3. CC-LIST is an instantiation of a parameterised module defining a collection of complex classes which is built as follows:

```
make CC-LIST is LIST[COMPLEX-CLASS]*(sort List to CcList) endm
```

Where COMPLEX-CLASS is the OBJ-module describing the OASIS complex classes. Analogously,

```
make EC-LIST is LIST[ELEMENTARY-CLASS]*(sort List to EcList) endm
```

ELEMENTARY-CLASS is the OBJ-module employed to define the signature and related equations describing the structure and behavior of the OASIS elementary classes. It contains, among other:

```
obj ELEMENTARY-CLASS is sort Ec .
      protecting EVENT-LIST .
      dfn CAttribList is ATTRIBCDEF-LIST .
      dfn VAttribList is ATTRIBVDEF-LIST .
      protecting CONSTRAINTS .
      protecting PRECOND-LIST .
      protecting TRIGGER-LIST .

sorts  Pc Class Attrib CAttrib  VAttrib Constraints Varsdecl Events
        Private Shared Preconditions Triggering .
subsorts    Pc Ec < Class .
*** CLASS HEAD
      op elementary class : ClassId Attrib Events -> Ec .
*** ATTRIBUTES OPERATION SYMBOLS
      op attributes : CAttrib Constraints VAttrib -> Attrib .
                  *** CONSTANT
      op constants : CAttribList CAttribList -> CAttrib .
      ................
                  *** VARIABLE
      op variables : VAttribList -> VAttrib .
      op vars : VardeclList -> Varsdecl .
*** EVENTS, PRECONDITIONS AND TRIGGERING
      op preconditions : VardeclList PrecondList -> Preconditions .
      op Error : -> PrecondList .
      op events : Private Shared Preconditions Triggering -> Events .
      op private _ ; new _ destroy _ ; _ : Varsdecl Event Event
            EventList -> Private .
      op shared _ ; _ : Varsdecl EventList -> Shared .
      op triggering : VardeclList TriggerList -> Triggering .
      op Error : -> TriggerList .          ..........
*** some non-constructor (also valid for complex class)
      op cclassid : Class -> ClassId . *** returns class identifier
      op ceventid : Event -> EventId .
      op ckey : Class -> CAttribList .
      op cnokey : Class -> CAttribList .
      op cnew : Class -> Event .
            ........ axioms ......
endo
```

Components of classes are similarly formalized in this constructive way: we have algebraic specifications formalizing constant (ATTRIBCDEF, ATTRIBCDEF-LIST) and variable (ATTRIBVDEF, ATTRIBVDEF-LIST) attributes, static constraints (CONSTRAINTS) private and shared events (EVENT, EVENT-LIST), preconditions (PRECOND, PRECOND-LIST) and triggering relations (TRIGGER, TRIGGER-LIST). Error messages are included, as constants of different sorts, in most of the modules for handling exceptions and errors. In addition, a variety of non-constructor operations are included to consult different properties of objects and classes (Looking at term *cs = library* below will help to understand these specifications). Thus, we can formally specify any conceptual schema as a ground term *cs*, $cs \in T_{OP,Cs} \subseteq T_{OP}$, where T_{OP} is the ground terms set and $T_{OP,Cs}$ is the ground term set of sort Cs. *cs* is the unique (up to congruence) canonical term representing the conceptual scheme described by an OASIS specification. The fact of representing the conceptual schema as a ground term (without variables) will allow efficient reduction using the OBJ3 operational semantics (conditional rewriting). The concrete ground term *cs* formally describes the structure and dynamic properties of the object system. For instance, the internal term *cs* corresponding to our example is, now changed to "library":

```
let library= conceptual schema('Library,domains('nat 'bool 'string
'time), Borrower Book,Loan)
```

with Borrower, Book $\in T_{OP,Ec} \subseteq T_{OP}$; Loan $\in T_{OP,Cc} \subseteq T_{OP}$

(we detail only subterm Borrower; Book and Loan would be similar subterms)

```
let Borrower = elementary class('Borrower,
      attributes(constants(key(cdef('id-number,'nat)),
                           nokey(cdef('name,'string) cdef('age,'nat))),
                           constraints('age < 100 , 'age > 16),
            variables(vattrib('books-on-loan['Borrower,'time] is 'nat,
            equations((('L is 'Book )  ('B is 'Borrower) ('T is 'time)),
      < 'books-on-loan['B,'T] + 1,ev('borrow,key('L),key('B),'T) >
      < 'books-on-loan['B,'T] - 1,ev('return,key('L),key('B),'T) >
      < 0,ev('join,key('B),'T) >)))),
      events(private vars(('B is 'Borrower) ('T is 'time));
      new ev('join,key('B), 'T)
      destroy ev('remove,key('B),'T) ; ev('notify,key('B), 'T) ; nil,
            shared vars(('B is 'Borrower) ('L is 'Book) ('T is 'time)) ;
                  ev('borrow,key('L),key('B),'T)
                  ev('return,key('L),key('B),'T),
      preconditions(('B is 'Borrower) ('L is 'Book) ('T is 'time)),
      < ev('remove,key('B),'T),'books-on-loan['B,'T] = 0 >
      < ev('borrow,key('L),key('B),'T),'books-on-loan['B,'T] < 5 >
      < ev('return,key('L),key('B),'T),'books-on-loan['B,'T] > 0 >),
      triggering( vars (('B is 'Borrower) ('T is 'time)),
       self  < ev('notify,key('B),'T),'books-on-loan['B,'T] = 4 >) .
```

Note that each well-formed term *cs* represents a different conceptual schema for a particular object system, thus having a general tool of description of information Systems (Fortunately, *cs* terms are not visible to users¡).

4.1 Built-in Modules and Order-Sorting

An advantage provided by the use of OBJ3 was the possibility of using built-in OBJ3 modules (INT,NAT,BOOL,STRING,...) as primitive classes, in addition to those

defined by ourselves. By subsorting on the related sorts we had the possibility of connecting them with the underlying arithmetic of Lisp so making the environment, not only formal and theoretical, but also operative for practical use.

4.2 Extensible Operators

A characteristic of our kernel is the facility given to the specification of defining new operators between classes that encourages reusability of primitive components: OASIS thus becomes an extensible language [1]. The objective is to allow developers to build new complex classes through the use of operators defined by themselves, starting from the standards operators provided by OASIS (generalization/ specialization, aggregation/ projection and classification/ instantiation) and offering the possibility of using the primitive operation symbols defined in AS-OASIS with its related stated semantics. Developers will give the syntactic definition for the operator together with its semantics using an equational theory.

For each new extensible operator defined it is required to add to AS-OASIS:

- The operation symbol (constructor) in mixfix form:
  op defined class _ <operation-name> _..._ : S1 S2...Sn → Cc .

where "defined class" is the keyword for extensible operators and "operation-name" is the name assigned by the developer to the new operator.

- A declaration of the variables to be used in the equations describing the semantics of the extensible operator.

- A set of equations describing the semantics of the extensible operator, which will be converted to rewriting rules by the OBJ3 rules generation system.

- A set of equations describing Error management, exception handling and trivial cases for that operator, which will also be converted to rewriting rules.

5 Formalization of the Information Base

The algebraic theory INFORMATION-BASE enriches AS-OASIS with new modules and operations for managing the information base:

```
obj INFORMATION-BASE is
      protecting  AS-OASIS .
      dfn ParallelTrace is PARALLELTRACE .
      dfn InfBase is INFBASE .

      .......new sorts, vars and operation symbols......

      .......axioms.......
endo

obj TRACE is
      sort Trace .
      protecting CLASSID .
```

```
      protecting EVENT .
      op nilTrace : -> Trace .
      op *ev : EventId ConsValList Trace -> Trace .
*** Error Handling in prototyping
      op Error. Bad input keys for _ : ClassId -> Trace .
      op Error : -> Trace .
endo

make PARALLELTRACE is |||{TERM-LIST,TRACE} * (sort Array to
                                              ParallelTrace) endm
make INFBASE is |||{CLASSID,PARALLELTRACE} endm
```

"|||[INDEX VALUE :: TRIV]" is a parameterized module handling pairs of structures and TRIV is the trivial theory which only requires existence of at least one sort. Operation symbols (*acall, *av) are provided to implement the observability function α.

This specification allows the formal description of the information base states as a ground term, *ib*, of its ground term algebra, $ib \in T_{OP,InfBase} \subseteq T_{OP}$. This term is the unique (up to congruence) canonical term representing the state of the information base as the parallel composition of the state of each one of the objects living (or destroyed) in the system. Persistence of the information base is ensured by keeping term *ib* updated. Objects life-cycles tracing is done by reductions of the corresponding operations and terms with the underlying rewriting system:

```
*** INFORMATION BASE
*** Borrower individual traces (should be read from right to left)
..............
let ptrace1 = *ev('return, key(2),key(1),*ev('borrow, key(2),key(1),
*ev('borrow, key(3),key(1), *ev('return, key(1),key(1),*ev('borrow,
key(1),key(1), *ev('join, key(1),nokey('Maria 34), nilTrace))))))) .
```

"key(n)" refers to the book with oid n. The number following the name in "join" represents the age, e.g.: 'Maria 34.

Each occurrence of an event has associated a time instant (which does not appear in the text for the sake of clarity). For shared event ocurrences this time instant will be exactly the same.

```
*** All Borrower traces together:

let Borrowertrace = //(1, ptrace1, //(2, ptrace2, //(3,ptrace3,nilArr)))

*** // is an associative and commutative operation standing for
"parallel composition" of traces; ptrace2, ptrace3 describe the traces
respectively for persons "2" and "3" in a similar way as ptrace1 does.
```

If we similarly use the terms Booktrace and Loantrace for all the object traces from classes Book and Loan, respectively, the term ib (now renamed infbase) for our example is:

```
let infbase= //('Loan,Loantrace, //('Book, Booktrace, //('Borrower,
Borrowertrace, nilArr)))
```

Together with the kernel described above, we have designed an integrated set of computer-aid software tools to support prototyping of complex software system from an object-oriented perspective (OOAP environment) which consists of an User Interface ,

the Algebraic Theory described here and the Execution Support System (the OBJ3 rewriting system).

6 Conclusions

The kind of algebraic formalization shown in this article may have practical consequences e.g. in the requirements engineering phase, construction of CASE tools to assist in transformations of specifications, as a basis for OO code generation in an adequate OOP or OODB, or simply as a unification of the OO model concepts presented here. A related work for unifying Structured models (JSD, Structured Analysis, Structured Design) has been investigated in [22] using algebraic categorial theory. Finally, the algebraic formalization procedure described here (which is different from the one used in CMSL [23] or in FOOPS [7]) is easily carried to other languages similar to OASIS, such as TROLL [12] or OBLOG [17] [18].

Acknowledgments

The authors wish to thank Dr. Jose Meseguer, from SRI, for his support and the facilities offered with OBJ3 and related theoretical issues.

References

[1] M. Atkinson; F. Bancilhon; D. DeWitt; K. Dittrich; D. Maier and S. Zdonik. "The object-oriented database system manifesto". In Proc. of the First Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan, December 1989

[2] G.Booch. "Object Oriented Design with applications". The Benjamin/Cummings Publishing Company Inc. (1991)

[3] A.M. Davis. "Operational prototyping: a new development approach". IEEE Software. September 1992, vol. 9, n.5

[4] H.-D.Ehrich;A.Sernadas; C.Sernadas. "Objects,Object Types and Object Identification" in Categorical Methods in Computer Science, H.Ehrig (ed.). Springer-Verlag 1988

[5] H.Ehrig;B.Mahr. "Fundamentals of Algebraic Specification 1". Springer-Verlag. 1985

[6] J.A.Goguen; J.J Jouannaud and J.Meseguer. "Operational Semantics of order-sorted algebra", in W.Brauer (ed.) Proc. 1985 Int. Conf. on Automata, Languages and Programming, Springer 1985, LNCS n° 194

[7] J.A.Goguen; J.Meseguer. "Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics". in Research Directions in Object-Oriented Programming, B.Shriver and P.Wegner (eds.). MIT Press (1987) pp.417-477

[8] J.Goguen; J.Meseguer, "Order-Sorted Algebra I", Technical Report, SRI International, Standford University, 1988

[9] J.A.Goguen; T.Winkler;J.Meseguer;K.Futatsugi and J.P.Jouannaud, "Introducing OBJ3" SRI-CSL Report (Draft of January 1992)

[10] IS-CORE: Information Systems - Correctness and Reusability Esprit-2 BRA WG 3023, Second Year Report, September 1991

[11] Udo W. Lipeck, G. Koschorreck (eds.). Proc. of the Int. Workshop on : Information Systems - Correctness and Reusability IS-CORE'93, Institut für Informatik, Hannover, September 1993

[12] R.Jungclaus, T.Hartmann, G.Saake, C.Sernadas, "Introduction to TROLL- A Language for Object-Oriented Specification of Information Systems" in . [10]

[13] B.Meyer, "Object-Oriented Software Construction". Prentice-Hall 1988

[14] Jesus G. Molina; Ambrosio Toval; Manuel González. "Sape: a structured analysis prototyping environment" stream: software development and maintenance (Poster Sessions). 12TH World Computer Congress. IFIP CONGRESS 92 Madrid (Spain). September 7-11,1992

[15] O.Pastor; Hayes,F.; S.Bear. "OASIS: An Object-Oriented Specification Language" in Proceedings of the CAISE-92 Conference, Spriger-Verlag (1992)

[16] I. Ramos; O. Pastor and J.H.Canós "On the use of algebras as a semantic domain for Objects Societies", Proc. of the Deductive Approach to Information Systems and Databases, Costa Brava (Catalonia, Spain), Sept. 92

[17] A.Sernadas; J.Fiadeiro; C.Sernadas; H.-D.Ehrich. "The Basic Building Blocks of information Systems" Proc. IFIP 8.1 Working Conference, Falkenberg, E.,Lindegreen,P. (eds.), North Holland, Amsterdam, 1989, pp.225-246

[18] Sernadas, A. ; Sernadas, C. Gouveia, P. ; Resende, P. "OBLOG: An Informal Introduction" Research Report INESC 1991.

[19] A. Toval. "EMTR-01: First Technical Report on the Euromethod Progress". Internal Report. E.U. Informática. Universidad de Murcia (Spain). October 1990.

[20] A. Toval. "EEC: EUROMETHOD IN PROGRESS". Short Note, IEEE Computer Society/TCSE Newsletter, April 1992

[21] A. Toval. "Formalización Algebraica de un Entorno de Producción Automática de Prototipos Orientados a Objetos". PH.D. dissertation. Departamento de Sistemas Informaticos y Computación. Universidad Politécnica de Valencia (Spain). Mayo 1994

[22] T.H.Tse. "A Unifying Framework for Structured Analysis and Design Models: an approach using initial algebra semantics and category theory". Cambridge University Press (1991)

[23] R.J.Wieringa. "Algebraic Foundations for Dynamic Conceptual Models". PhD Thesis, Vrije Universiteit, Amsterdam, 1990

IMPLEMENTATION OF A TERM REWRITING SYSTEM FOR SOLVING PROCESS QUERIES IN AN OBJECT-ORIENTED ENVIRONMENT.

Jaume Devesa, José Cuevas and Isidro Ramos
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n
46071 València (Spain)
Tel: +34 6 3877350
Fax: +34 6 3877359
E-mail: {jdevesa, jcuevas, iramos}@dsic.upv.es
April 1994

ABSTRACT

The definition of an object as an observable process [17] leads us to an approach to the formalization of the objects coming from the world of processes. According to [18], we have opted for process algebrae to describe the objects lives. An object is represented by a process query. A term rewriting system can be developed as the implementation of the operational semantics of an algebraic presentation. Thus, every process query can be solved by giving a sequence of events, which is a life of this object.

1. INTRODUCTION.

In the last years, many efforts have been developed to define what an object is. Most of this attempts are actually informal, based on the skill of the writter to describe such intuitive concept in a more or less convincing way. Others formalize the object concept using different methodological bases. This work approaches this problem in a formal way provided by the process theory.

An *object* is an entity, and some events occur along its existence. The sequence of events occurred is the object life. We call a *process* a description of all possible lives of an object. In other words, an object can be thought as a process from our point of view. As processes, objects may interact among each others in different ways: communicating, synchronizing,

Devesa, J., Cuevas, J., Ramos, I. Implementation of a Term Rewriting System for solving process queries in an object-oriented environment". Proc. Of the GULP-PRODE 94. 1994.

acting in parallel, and so on One of the main ways to describe processes and their interaction is *process algebra*, where the semantics of objects is defined axiomatically [1].

Our proposal fits in this algebraic approach Our process definitions are sorts of equations (*process queries*) of the language defined by a process algebra. But the aim of this paper goes beyond Term rewriting is an efficient mechanism to implement the operational semantics of equational specifications So, we show how a term rewriting system (shortly trs) can be built from this algebra, where a process is a query The trs rewrites the process query to a sequence of events, that is a possible life of the object represented by it

The next section reviews the work on processes we have recently developed In the section 3, we introduce the algebra of processes on which the specifications of objects are based. The section 4 shows the term rewriting system implemented from that algebra In the last section, we give some conclusions and suggest some improvements to our work. In appendix 1 an implementation in Prolog can be studied, and in the appendix 2 there is an example of execution

2. PREVIOUS WORK.

In [18], a process theory called ACP (Algebra of Communicating Processes) is presented. It is an equational specification of the process algebra, developed by Bergstra & Klop, which defines the sorts EVENT and PROCESS and operations like + (choice), · (sequence), ‖ (parallel composition of two processes) and | (synchronous execution of two events) The intended model of this algebra is the set of all possible processes that behave according to the axioms of the theory Thus, an object specification can contain a set of equations, each one is a query in the process model The meaning of the query (process variable instantiation) is that the object executes an instance of this process, i e it is a possible life.

Wieringa also shows how the ACP is a conservative extension of a BPA (Basic Process Algebra), which is the specification of processes using only the operations + and ·, and a constant of sort EVENT δ (dealing with the situation in which a process deadlocks) In other words, every closed ACP term can be rewritten to one containing only alternatives and sequences of processes

We have adapted these ideas to the deductive and object-oriented environment, represented by the language OASIS [12, 17, 13] An APO (Algebra of Process for Objects) has been built by adding the new sort TRANSACTION and operations dealing with it Thus, events and other transactions can be gathered in transactions, that actually are the execution units. We have also given a formalization of the concept of transaction, based on the Dynamic Logic [6] Every object has a specific state in every moment, and a Kripke structure can be built with such states, whose function of transition (i e the function that changes the object state) is defined for every event, transaction and process The succesful execution of a transaction makes the object reach the final state But if the transaction fails, the object does

not leave the initial state Afterwards, the effects of a transaction are not noticed until the end This is formalized by forbidding the process observation in the intermediate states

In OASIS, an object is said to be an observable process, because there are mechanisms to observe the objects states The state of any object is made out of different items, so-called attributes, each of them has a value belonging to a particular sort Attribute values change as the events in the process occur The APO is used to declare processes and transactions

Similarly as ACP terms, an APO term can also be rewritten to a BPA term An implementation of the BPA can be found in [10] Every process is divided into its possible alternatives A prototyping environment chooses one of them, depending on specified conditions

3.- PROCESS ALGEBRA

The APO which is going to be introduced is an order-sorted process specification that has three ordered sorts: EVENT≤TRANSACTION≤PROCESS, and the following set of operations:

_+_: process process → process (internal alternative)
_⊕_: process process → process (external alternative)
_._: process process → process (sequence)
_|_: process process → process (parallel composition)
__: process process → process (left merge)
_|_: event event → event (communication)
δmess: process → process (encapsulation)
α_β: process → process (transactions)
δ: event (deadlock)
ε: event (transaction failure)
nop: event (null event)

The specification is parameterized by:

- a set E of constants of sort event, called event names

- a set MESS ⊆ E of messages Their elements are events that cannot occur by themselves: they cannot occur except as part of a communication, as we will state clearly later

- The syncronization function γ: event event → event It states that two events can occur at the same time, giving a new name to this *shared* event occurrence

x + y is the process that has a choice between x and y, depending on internal conditions that the system evaluates x ⊕ y is similar, but in this case is the environment who decides what

to do x y is the process x followed by the procces y (the is often omitted when no ambiguity arises), and x || y is the process consisting of the paralell execution of x and y || is an auxiliary operator x \\ y is the parallel execution of x and y, whose first event is the first event of x Processes relate each other by means of communication (| operator) and encapsulation (δmess operator), as it will be explained later A transaction is a process, with a beginning event α and an ending event β (*commit* event) δ is a special constant interpreted as deadlock: when δ occurs nothing more can happen ε is an event that represents a failure in a transaction, whose effects must be undone (*rollback* event) *nop* is the absence of any event

The equations of the APO are given below, where x, y and z are variables of type PROCESS and e, e1 and e2 are variables of type EVENT First, let us see the axioms for the sequence operator:

$\delta\ \delta \rightarrow \delta$ (1)
$\delta\ x \rightarrow \delta$ (2)
$\varepsilon\ \varepsilon \rightarrow \varepsilon$ (3)
$\varepsilon\ x \rightarrow \varepsilon$ (4)
$x\ \varepsilon \rightarrow \varepsilon$ (5)
nop nop $\rightarrow$ *nop* (6)
nop $x \rightarrow$ *nop* (7)
x *nop* $\rightarrow$ *nop* (8)
$(x\ y)\ z \rightarrow x\ (y\ z)$ (9)
$(x+y)\ z \rightarrow x\ z + y\ z$ (10)

Now, the internal and external alternative:

$\delta+\delta \rightarrow \delta$ (11)
$x+\delta \rightarrow x$ (12)
$\varepsilon+\varepsilon \rightarrow \varepsilon$ (13)
$x+\varepsilon \rightarrow \varepsilon$ (14)
nop+*nop* $\rightarrow$ *nop* (15)
$x+$*nop* $\rightarrow$ *nop* (16)
$x+y \rightarrow y+x$ (17)
$x+(y+z) \rightarrow (x+y)+z$ (18)
$x+x \rightarrow x$ (19)

$\delta\oplus\delta \rightarrow \delta$ (20)
$x\ominus\delta \rightarrow x$ (21)
$\varepsilon\ominus\varepsilon \rightarrow \varepsilon$ (22)
$x\oplus\varepsilon \rightarrow \varepsilon$ (23)
nop$\oplus$*nop* $\rightarrow$ *nop* (24)
$x\oplus$*nop* $\rightarrow$ *nop* (25)

$x \oplus x \rightarrow x$ (26)
$(x+y) \oplus z \rightarrow (x \oplus z) + (y \oplus z)$ (27)
$x \oplus y \rightarrow y \oplus x$ (28)
$x \oplus (y \oplus z) \rightarrow (x \oplus y) \oplus z$ (29)
$x \oplus y \rightarrow x$ if choice(x,y)=x (30)
$x \oplus y \rightarrow y$ if choice(x,y)=y (31)

choice represents the election performed by the environment Next, we present the axioms for parallel composition, left merge and comunication $x \parallel y$ is the interleaving of x and y, having a choice between executing the first event of x, the first event of y, or a synchronous execution of a first event of both processes The rest of both processes is executed in parallel, in its turn

$x \parallel y \rightarrow (x \backslash\backslash y) + (y \backslash\backslash x) + (x \mid y)$ (32)

$e \backslash\backslash x \rightarrow e\ x$ (33)
$(e\ x) \backslash\backslash y \rightarrow e\ (x \parallel y)$ (34)
$(x+y) \backslash\backslash z \rightarrow (x \backslash\backslash z) + (y \backslash\backslash z)$ (35)

Communication is explained by marking out parallel composition from encapsulation of processes When we have a term e1|e2, according with the definition of γ, this term is rewritten as an element of the event sort; or as δ, if γ is undefined ($\perp$) for e1 and e2 The intuitive meaning of γ(e1,e2) is that e1 and e2 can occur syncronously

$\delta \cdot \delta \rightarrow \delta$ (36)
$e \cdot \delta \rightarrow \delta$ (37)
$nop \cdot nop \rightarrow nop$ (38)
$e \mid nop \rightarrow nop$ (39)
$e1 \mid e2 \rightarrow e2 \mid e1$ (40)
$(e1 \mid e2) \mid e3 \rightarrow e1 \mid (e2 \mid e3)$ (41)
$e1 \cdot e2 \rightarrow e3$ if γ (e1,e2)=e3 (42)
$e1 \mid e2 \rightarrow \delta$ if γ (e1,e2)=$\perp$ (43)
$(e1\ x) \cdot e2 \rightarrow (e1 \mid e2)\ x$ (44)
$(e1\ x) \cdot (e2\ y) \rightarrow (e1 \cdot e2)\ (x \parallel y)$ (45)
$x \cdot (y+z) \rightarrow (x \cdot y) + (x \cdot z)$ (46)

A transaction is a process delimited with the α and β events (transaction start and transaction ending) The main characteristic is that transactions represent *all-or-nothing* actions: if the transaction fails (represented by the ε event occurrence), the object remains in its initial state This is modeled substituting the transaction by the *nop* event The idea is basically the same as the transaction in relational databases: if any part of the transaction cannot occur, the whole transaction will not occur

$\alpha x \beta \rightarrow nop$ if $x = \varepsilon$ (47)

$\alpha x \beta \rightarrow x$ if $x \neq \varepsilon$ (48)

Encapsulation is achieved by using δmess operator If MESS = {e1, e2}, then δmess(x) is a process in which e1 and e2 cannot occur except as a part of a communication So γ defines which events can occur synchronously, and MESS defines which events cannot occur by themselves This operator is parameterized by a set MESS Its elements are those events that cannot occur by themselves, they must occur in a communication If one of these events appears in no communication the δ event is obtained

$\delta mess(e1) \rightarrow \delta$ if $e1 \in MESS$ (49)
$\delta mess(e1) \rightarrow e1$ if $e1 \notin MESS$ (50)
$\delta mess(x\ y) \rightarrow \delta mess(x)\ \delta mess(y)$ (51)
$\delta mess(x+y) \rightarrow \delta mess(x)+\delta mess(y)$ (52)

For external choice we have to add axioms so that the user will be asked for which of the options is chosen If the choice is internal, the system must have a set of conditions so that it can take an option at any moment

When we specify a system, we have to write the system events, the function γ and the set MESS This is needed to instantiate the trs

4. THE REWRITING SYSTEM

In this section, we explain the main differences between the APO, presented in the previous section, and the rewriting system In order to get a canonical trs (the rewriting ends, and its result is unique), some axioms of algebra have been adapted or eliminated

Axioms that indicate operation properties, have been removed, because they are not rewriting rules:

$(x\ y)\ z \rightarrow x\ (y\ z)$	(associative for)	(9)
$x+y \rightarrow y+x$	(commutative for +)	(17)
$x+(y+z) \rightarrow (x+y)+z$	(associative for +)	(18)
$x \oplus y \rightarrow y \oplus x$	(commutative for ⊕)	(28)
$x \oplus (y \oplus z) \rightarrow (x \oplus y) \oplus z$	(associative for ⊕)	(29)
$e1 \mid e2 \rightarrow e2 \mid e1$	(commutative for |)	(40)
$(e1 \mid e2) \mid e3 \rightarrow e1 \mid (e2 \mid e3)$	(associative for |)	(41)

The absence of the commutativity requieres that the following axioms have been added:

$\delta+x \rightarrow x$	(commutative for +)
$\varepsilon+x \rightarrow \varepsilon$	

$nop+x \rightarrow nop$

$\delta \oplus x \rightarrow \delta$ (commutative for $\oplus$)
$\varepsilon \oplus x \rightarrow \varepsilon$
$nop \oplus x \rightarrow nop$
$\delta | e \rightarrow \delta$ (commutative for |)
$nop | e \rightarrow nop$
$(x+y)|z \rightarrow (x|z)+(y|z)$

The absence of associativity force us to apply the process operators two by two In the presence of more than two alternatives, for example $x \oplus y \oplus z$, we must choose first between x and y We cannot select z until the previous has been done

External and internal choice differ in who decides the choice The axioms are basically the same, but we distribute the internal choice with regard to the external choice because we want to solve the internal one in the end If the choice involves a deadlock, there is no choice and the system selects the process, not the deadlock The choice appears as a query to the environment, who decides what to do

Other important remark is the non-conmutativity of the sequence operator. Thus, we formalize the unidirectionality of the pass of time Aditionally, the absence of the axiom x(y+z) formalizes the sensitivity of the equal sign for nondeterminism x(y+z) is not equal to xy+xz: the first process is deterministic, the second nondeterministic We call the choice x+y arbitrary because it is not specified which branch is choosen

Let us see as an example the information system of a library Readers borrow books Books are lent to readers Readers give the books back Other events are the following: a new book can be acquired; a person can ask to be a new reader; a reader can leave this condition; a reader can be notified that the loan time has finished The involved objects, reader and book, are specified as process queries The trs will animate the objects society, rewriting the parallel composition of both processes to a possible life of the system The trs implemetation in Prolog is given in Appendix 1 An execution of this example can be studied in Appendix 2

The definitions of the event set E, the event set MESS, and the function gamma are:

E = {new_book, new_reader, cease_reader, be_notified, borrow, lend, loan, give_back, be_returned, return}
MESS = {borrow, lend, give_back, be_returned}
γ(borrow,lend) = loan
γ(give_back,be_returned) = return

Borrowing and lending occur at the same time (loan) Afterwards, each one cannot occur separately It is just like when a reader gives a book back: the book is returned at this moment

If the lives of a reader and a book are respectively:

READER = new_reader borrow (give_back ⊕ be_notified)
BOOK = new_book lend be_returned

the parallel execution of both process can be rewritten in two different possibilities, depending on the choosen alternative If the reader gives the book back, we have a perfect synchronization between both objects:

READER || BOOK → new_reader new_book loan return

If the reader is notified that the loan time has finished, the process deadlocks because no synchronization is possible:

READER || BOOK → new_reader new_book be_notified δ

5. CONCLUDING REMARKS.

In this paper, we show the way a term rewriting system can be used to animate object societies Every object is represented by a process query, by using the operations defined by the algebra This process query is rewritten to a sequence of events, that is a possible life of the object Thus, $\forall$ t $\in$ APO, $\exists$ $t_0 \in$ BPA | t $\longrightarrow\!\!\!\!\!\rightarrow$ t_0, i.e the term t_0 is obtained from t after n≥0 rewriting steps t_0 is the ground term representing the life of the object, and is generated by the trs

This trs could also be used as correct trace analizer This improvement could be achieved by synchronizing the rewriting and the event ocurrence:

t	$\rightarrow t_1$	$\rightarrow t_2$	$\rightarrow$	$\rightarrow t_n = t_0$	(rewiting)
nop	$\rightarrow e_1$ nop	$\rightarrow e_2\ e_1$ nop	$\rightarrow$	$\rightarrow$ en $e_2\ e_1$ nop	(object life)

An event e_i would be relevant if $e_i \in$ inic(t_i), being inic(t_i) the first element of the process query $t_i \in$ APO The term t will be rewritten t $\longrightarrow\!\!\!\!\!\rightarrow$ t_i until inic(t_i) returns a single event, or a set of events performing a choice

In appendix 1, there is an implementation of the trs in Prolog The use of languages more suitable to deal with processes, like RAP [8] or LOTOS [2], would improve its eficiency and will be our next implementation work

In the very near future, we are interested in extending this work with the agent concept, following the ideas presented in [20] An agent is an object able to execute a transaction declared in other object The latter would serve the former with the execution of the transaction

REFERENCES.

[1] E.Artesiano, G Reggio, "Algebraic Specification of Concurrency", University of Genova (Italy), 1992.

[2] A Azcorra, J Quemada, S Pavón, "An Introduction to LOTOS", Technical Report, Technical University of Madrid, 1993

[3] J H Canos, O.Pastor, "Functional and Object Oriented Specification of Information Systems", Proceedings of DEXA, Berlin 1991

[4] D.Coleman, C Dollin, R Gallimore, P Arnold, I Rush, "An Introduction to the Axis Specification Language", Technical Report, Hewlett Packard Labs , Bristol (U K) 1988

[5] P Dauchy, M -C Gaucel, "Algebraic Specifications with Implicit State", University of Paris-Sud, 1993

[6] D Harel, "Dynamic Logic" in Handbook of Philosophical Logic II, Reider 1985

[7] C A R Hoare, "Communicating Sequential Processes", Prentice-Hall 1985

[8] H Hussmann, "Rapid Prototyping for Algebraic Specification RAP System, Users Manual", Technical Report, University of Passau, 1989

9] J P Jouannaud, P Lescanne, "La Réécriture", in Technique et Science Informatiques, 987

10] J Mingot, "Procesos y Transacciones en un Sistema de Especificación Orientado a)bjetos", Phd work, Valencia 1993

1] O Pastor, I Ramos, V Casado, "Object Oriented and Relational Specification of ıformation Systems", Proceedings of the 2nd Workshop on the Deductive Approach to formation System and Databases, Aiguablava 1991

2] O Pastor, "Diseño y Desarrollo de un Entorno de Producción Automática de Software sado en el Modelo Orientado a Objetos", PhD Thesis, Valencia 1992

3] O Pastor, I Ramos, J Cuevas, J Devesa, "Oasis: An Object Definition Language for)ject-Oriented Databases", Internal Report, Technical University of Valencia, 1994

l] I Ramos, "Logic and Object Oriented Data Bases: a Declarative Approach",)ceedings of DEXA, 1990

[15] I Ramos, O Pastor, V Casado, "Open and Active Formal Specification of Information Systems", Proceedings of DEXA, Berlin 1991

[16] I Ramos, O.Pastor, J H Canos, "On the Use of Algebras as Semantic Domain of Object Oriented Societies", Proceedings of the 3rd Workshop on the Deductive Approach to Information System and Databases, Roses 1992

[17] I Ramos, O Pastor, J Cuevas, J Devesa, "Objects as Observable Processes", Proceedings of the 3rd Workshop on the Deductive Approach to Information System and Databases, Lloret de Mar 1993

[18] R J Wieringa, "Algebraic Foundations for Dynamic Conceptual Models", PHD Thesis, Vrije Universiteit, Amsterdam 1990

[19] R J Wieringa, "A formalization of objects using equational dynamic logic", Lecture Notes in Computer Science 566, 2nd Intn'l Conference on Deductive and Object-Oriented Databases, Springer 1991

[20] R J Wieringa, J J Ch Meyer, "Actors, Actions, and Initiative in Normative System Specification", in Annals of Mathematics and Artificial Intelligence 1992

APPENDIX 1.
IRS IMPLEMENTATION IN PROLOG

```
/* OPERATIONS */

?-op(100,xfy,·).          /* X·Y      SEQUENCIAL COMPOSITION */
?-op(101,xfy,++).         /* X++Y     INTERNAL ALTERNATIVE */
?-op(101,xfy,+?).         /* X+?Y     EXTERNAL ALTERNATIVE */
?-op(102,xfy,|).          /* X|Y      COMMUNICATION */
?-op(103,xfy,\\).         /* X\\Y     LEFT-MERGE */
?-op(104,xfy,||).         /* X||Y     PARALLEL COMPOSITION */
?-op(105,fx,tr).          /* tr(X)    TRANSACTIONS */
?-op(106,fx,dM).          /* dM(M,X)  ENCAPSULATION */

/* EVENTS */

event(d) :- !.            /* DEADLOCK */
event(e) :- !.            /* TRANSACTION FAILURE */
event(nop) :- !.          /* NO OPERATION */

rewrite(E,E) :- event(E), !.

/* AXIOMS FOR · */

rewrite(d·d,d) :- !.
rewrite(d·X,d) :- !.
rewrite(e·e,e) :- !.
rewrite(e·X,e) :- !.
rewrite(X·e,e) :- !.
rewrite(nop·nop,nop) :- !.
rewrite(nop·X,nop) :- !.
rewrite(X·nop,nop) :- !.

rewrite((X++Y)·Z,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(Z,Z1),
                rewrite(X1·Z1,R1),
                rewrite(Y1·Z1,R2),
                rewrite(R1++R2,R), !.

rewrite(X·Y,X1·Y1) :-
                rewrite(X,X1),
                rewrite(Y,Y1), !.

/* AXIOMS FOR ++ */

rewrite(d++d,d) :- !.
rewrite(X++d,R) :- rewrite(X,R), !.
```

```
rewrite(d++X,R) :- rewrite(X,R), !.
rewrite(e++e,e) :- !.
rewrite(X++e,e) :- !.
rewrite(e++X,e) :- !.
rewrite(nop++nop,nop) :- !.
rewrite(X++nop,nop) :- !.
rewrite(nop++X,nop) :- !.

rewrite(X++X,R) :-
                rewrite(X,R), !.

rewrite(X++Y,X1++Y1) :-
                rewrite(X,X1),
                rewrite(Y,Y1), !.

/* AXIOMS FOR +? */

rewrite(d+?d,d) :- !.
rewrite(X+?d,R) :- rewrite(X,R), !.
rewrite(d+?X,R) :- rewrite(X,R), !.
rewrite(e+?e,e) :- !.
rewrite(X+?e,e) :- !.
rewrite(e+?X,e) :- !.
rewrite(nop+?nop,nop) :- !.
rewrite(X+?nop,nop) :- !.
rewrite(nop+?X,nop) :- !.

rewrite(X+?X,R) :-
                rewrite(X,R), !.

rewrite(X++Y+?Z,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(Z,Z1),
                rewrite(X1+?Z1,R1),
                rewrite(Y1+?Z1,R2),
                rewrite(R1++R2,R), !.

rewrite(X+?Y++Z,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(Z,Z1),
                rewrite(X1+?Y1,R1),
                rewrite(X1+?Z1,R2),
                rewrite(R1++R2,R), !.

rewrite(X+?Y,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                choice(X1,Y1,R1),
                rewrite(R1,R), !.
```

```
/* AXIOMS FOR || */

rewrite(X||Y,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(X1\\Y1,R1),
                rewrite(Y1\\X1,R2),
                rewrite(X1|Y1,R3),
                rewrite(R1++R2++R3,R), !.

/* AXIOMS FOR \\ */

rewrite(E\\X,R) :-
                event(E),
                rewrite(X,X1),
                rewrite(E·X1,R), !.

rewrite(E·X\\Y,R) :-
                event(E),
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(X1||Y1,R1),
                rewrite(E·R1,R), !.

rewrite((X++Y)\\Z,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(Z,Z1),
                rewrite(X1\\Z1,R1),
                rewrite(Y1\\Z1,R2),
                rewrite(R1++R2,R), !.

/* AXIOMS FOR | */

rewrite(d|d,d) :- !.
rewrite(E|d,d) :- event(E), !.
rewrite(d|E,d) :- event(E), !.
rewrite(nop|nop,nop) :- !.
rewrite(E|nop,nop) :- event(E), !.
rewrite(nop|E,nop) :- event(E), !.

rewrite(E1|E2,E3) :-
                event(E1),
                event(E2),
                gamma(E1,E2,E3), !.

rewrite(E1|E2,d) :-
                event(E1),
                event(E2), !.
```

```
rewrite(E1·X|E2,R) :-
                event(E1),
                event(E2),
                rewrite(E1|E2,R1),
                rewrite(X,X1),
                rewrite(R1·X1,R), !.

rewrite(E1|E2·X,R) :-
                event(E1),
                event(E2),
                rewrite(E1|E2,R1),
                rewrite(X,X1),
                rewrite(R1·X1,R), !.

rewrite(E1·X|E2·Y,R) :-
                event(E1),
                event(E2),
                rewrite(E1|E2,R1),
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(X1||Y1,R2),
                rewrite(R1·R2,R), !.

rewrite(X++Y|Z,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(Z,Z1),
                rewrite(X1|Z1,R1),
                rewrite(Y1|Z1,R2),
                rewrite(R1++R2,R), !.

rewrite(X|Y++Z,R) :-
                rewrite(X,X1),
                rewrite(Y,Y1),
                rewrite(Z,Z1),
                rewrite(X1|Y1,R1),
                rewrite(X1|Z1,R2),
                rewrite(R1++R2,R), !.

/* AXIOMS FOR tr */

rewrite(tr(X),X1) :-
                rewrite(X,X1),
                X1\=e, !.
rewrite(tr(X),nop) :- !.

/* AXIOMS FOR dM */

rewrite(dM(M,E),d) :-
                event(E),
                member(E,M), !.
```

```
rewrite(dM(M,E),E) :-
                event(E), !.

rewrite(dM(M,X·Y),R) :-
                rewrite(dM(M,X),X1),
                rewrite(dM(M,Y),Y1),
                rewrite(X1·Y1,R), !.

rewrite(dM(M,X++Y),R) :-
                rewrite(dM(M,X),X1),
                rewrite(dM(M,Y),Y1),
                condition(X1,Y1,R), !.

rewrite(dM(M,X),R) :-
                rewrite(X,X1),
                rewrite(dM(M,X1),R), !.

/* EXTERNAL CHOICE */

:hoice(X,Y,X) :-
                write('1. '), write(X), nl,
                write('2. '), write(Y), nl,
                read(C), C=1, !.

hoice(X,Y,Y) :- !.

' INTERNAL CONDITION */

ndition(d,d,d) :- !.
ndition(d,X,X) :- !.
ndition(X,d,X) :- !.
ndition(X,Y,X) :- !.

 SET OF EVENTS */

mber(X,[]) :- !, fail.
ber(X,[X|Z]) :- !.
ber(X,[Y|Z]) :- member(X,Z).

 REWRITING START */

 (X,R):- mess(M), rewrite(dM(M,X),R).
```

A Knowledge-Base Architecture for Object Societies[†]

José H. Canós, M.C. Penadés, I. Ramos and O. Pastor
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n
E-46071 València (Spain)
jhcanos@dsic.upv.es

Abstract

This paper introduces KAOS, a Deductive, Active and Object-Oriented Database Management System (DAOODBMS) built at DSIC over a Knowledge-Base Management System (KBMS). We have based its construction on the advantages of such a system, so we have definitely made what we could call an object layer on top of the knowledge-based systems. An operational prototype that is a first version of a DAOODBMS compliant with almost all characteristics collected in documents such as the OODBMS Manifesto and the standard ODMG-93 is currently available on HP-UX workstations.

1. Introduction.

Despite the great efforts that the Information Management Community has made in the last few months to establish a standard model for ODBMS at every level (Object Definition Language (ODL), Object Management Language (OML), Object Query Language (OQL), etc.), the main ideas of the object paradigm diminish as the abstraction level decreases to the physical level, where we can find records in the purest relational style. This represents a serious impedance mismatch against the object model we are trying to implement.

This paper introduces KAOS, a DAOODBMS built at DSIC with the aim of reducing the impedance mismatch described above. In fact, it is a system where only one language is used as ODL, OQL and OML (database programming language). The KAOS system starts from the object model underlying OASIS, a formal specification language ([1]) which uses dynamic logics to describe (the state and) the behaviour of objects through well-formed formulae on these logics. A declarative semantics based on Kripke structures ([2]) and a Plotkin-style operational semantics ([3]) have converted OASIS into a very suitable tool for rapid prototyping as well as for becoming the unified language we mentioned earlier.

We will first introduce KBMS1 ([4]), the system on which KAOS was built. Next, we will write about the implementation of the OASIS model over KBMS1, and will finish showing the system architecture and describing its basic features. We are not dealing here about how KAOS is (almost) full Manifesto ([5]) and ODMG-93 ([6]) compliant, neither with how the OASIS language has become the only language to interact with the system. For more information about these topics, please see [7,8].

2. The KBMS1 system

KBMS1 is a Knowledge-Base Management System developed at the Hewlett-Packard Laboratories in Bristol, UK. It is a logic programming-based tool that allows for

[†] Published in the *DEXA-95 Workshop Proceedings*, London, September 1995.

Canós, J. H., Penadés, M.C., Ramos, I., Pastor, O. A Knowledge-Base Architecture for Object Societies. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995.

the storage, retrieval and update of large amounts of knowledge. By knowledge we mean sets of (Horn) clauses, written in kbProlog, a Prolog language extended with theories and primitives to handle these theories.

The theories provide a partitioning of the knowledge base; a *theory* is an unordered set of procedures (a procedure being an ordered set of Horn clauses with the same functor and arity of the atom on their heads). They are first class objects, and their updates are treated in a declarative manner, so that a program is not able to produce side-effecting updates on itself, as happens with assert and retract in Prolog. We may then make deductions over theories that are not updated at deduction time.

There are two ways to refer to theories: through their name and through their theory identifier. A special kind of theories, the so-called *system theories*, contains all the other theories defined in the knowledge system. They provide the logical organisation of theories and their metatheories, and store the system's state, represented by the contents of the theories it contains at a given moment.

Each theory has three meta-theories associated to it: the *interpreter* theory, the *assimilator* theory and the *attribute* theory.

The interpreter theories are the query interface to theories (every query to a theory is handled by its interpreter). To query a theory, we have to use the kbProlog predicate *prove(Thry, Qry, S)*, by which we are asking the theory *Thry* the query *Qry* in the state of the system represented by the system theory *S*. *Qry* may be any Prolog goal we wish to demonstrate in *Thry*.

Similarly, assimilator theories are the updating interface for theories. To make changes in a theory, we must use the predicate *update(Thry, Upd,S, NewS)*, by which we produce the update *Upd* in *Thry*, switching from a system state *S* to a new one *NewS* in which *Thry* has already been updated (we can consider that NewS=Upd(S)).

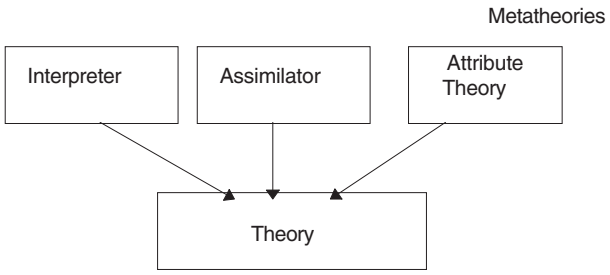


Figure 1: Internal representation of a kbProlog theory

2.1 Persistence

KBMS1 was built over the relational DB manager ALLBASE/SQL in order to be able to deal with persistent theories. This way knowledge bases can be stored and retrieved in a DB-like style as well as stored and retrieved efficiently (which is more important).

3. Objects in KBMS1.

The notion of theory yields a set of characteristics which are nicely close to the object paradigm concepts: each theory has its own *identifier* that distinguishes it from others, and its contents are *encapsulated* through its interpreter, assimilator and attribute metatheories. It is possible to demonstrate goals in theories which are different from those that generate there goals (in the most genuine *message/method*-style interaction).

Therefore, it seems to be worthwhile to study the ablility of kbProlog to act as the OASIS object language. We could do that simply by defining an appropiate representation for objects as theories; It must be remembered that, in the OASIS object model, objects are erified as formal theories in dynamic logic.

We will represent *an object in a given state* as a kbProlog theory. This theory will contain predicates indicating what the current value of all the object's attributes is. Hence, two different states of an object will be represented by two theories that differ in the value of at least one attribute of the object. The associated interpreter and assimilator metatheories will contain the observation and update methods resp., while the attribute theory will store some object properties such as integrity constraints, triggering relationships, etc. The set of metatheories compose what in the OO approach is called a *type* (i.e. a set of properties).

The actual contents of the theory that represents an object state as well as that of the metatheories depends on the inference strategy chosen for the runtime system. In the following paragraphs we will discuss each of the two main tendencies. To illustrate our discussion, we will take as a target the OASIS specification of the books in a library which appears in figure 2. It can be seen how, following a clausal specification style, we made up an OASIS spec. An object has static properties (attributes) whose value (*not the signature*) may vary over time, and behaviour ones (events) that are directly related to the change of state of the object as shown in the valuations paragraph. The valuations are composed of a set of dynamic formulae. A dynamic formula is of the form $\varphi[e]\psi$, where φ and ψ are first-order formulae and *e* is an event, and has the following intuitive meaning: "in a state in which φ holds, after the occurrence of the event *e* a

new state will be reached in which ψ must necessarily hold".

Associated with one event of the object's signature, the event responsible for the creation of an object is in fact a service offered by the class to which the object belongs to. The same can be said about object destruction, although we could have non destroyable objects, if a problem requires them. Finally, every object's state must satisfy some integrity constraints, stated as 1st order or modal temporal formulae that may produce an event rejection if the state to which this event leads to is not constraint-compliant.

```
class book
    constant_attributes
        key  code(nat);
             title(string).

        constraints      N:nat.
        static
         code(N) & N>1 & N<1000.
    variable_attributes
    available(bool)
      valuation
        [new_book]  available(true).
        [loan]      available(false).
        [return]    available(true).
      end_valuation
    private_events
        new  new_book;
        destroy destr_book.
    shared_events
        loan with reader;
        return with reader.
    preconditions
        destr_book <- available(true).
        loan      <- available(true).
        return    <- available(false).
    processes
        book  <- new_book & loan & book0.
        book0 <- (loan or return) & book0 or book1.
        book1 <- destr_book.
end_class
```

Figure 2. OASIS specification of the class book.

In the processes section the correct lives of the objects are described using a process description formalism. In the class of the figure 2, we have used a Prolog-like syntax where the sequence operator is represented by &, the choice by or and the iteration is implemented by tail recursion (see [9,10] for a discussion about the process nature of objects).

Forward Inference. In query-oriented systems where the number of updates is less than the number of queries, it is more convenient to have the object states explicitly stored in the database. We may do it through an object representation like that in figure 3; how we define the current value of an object's attribute (without distinction between constant and variable) by means of a Prolog ground fact can be seen.

Any query or update to an object will be handled by the interpreter and assimilator theories resp. that, along with the attribute theory, constitute the type associated to the class book. *Every class instance shares the same metatheories*.

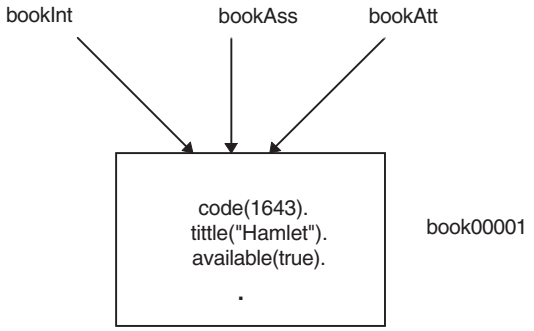


Figure 3. Representation of an object *called* book00001.

Following this approach, the kbProlog code for the interpreter is trivial, as can be seen in figure 4:

```
% bookInt
%%%%%%%%%%%%%%%%%%

interpret(_, T, _, Goal) <-  solve(T, Goal).
% COMPLEX QUERIES
solve(_, true) <- !.
solve(T, (A & B) ) <-  ! & solve(T, A) & solve(T, B).
solve(T, (A or B) ) <- ! & (solve(T, A) or solve(T, B)).
solve(T, not(Goal) ) <- ! & not(solve(T, Goal)).

% CONSTANT  AND VARIABLE ATTRIBUTES
solve(T, Goal) <- ! & call(T, Goal).
  nil
```

Figure 4. Interpreter for the class book

Any observation to the object is just the demonstration of the corresponding goal in the theory that represents the object's state (T in the previous listing). On the other hand, the assimilator contains the

implementation of the methods associated to the events of the object's signature (see figure 5).

Note how simple the way we handle the state changes is: we change the theory that represents the object's snapshot every time the value of an attribute changes. That along with the fact that every kbProlog theory can be made persistent, makes it possible to have a historical database in which to define some rollback and/or version control model.

```
% bookAss
%%%%%%%%%%%%%%%%%%%%%%
% loan/0
% EVENT FOR LOANING A BOOK
assimilate(loan, Thy, NewThy, _, S, S) <-
  ! & call(Thy, available(true)) &
% PRECONDITION
  replace(Thy, NewThy, available(_)<-true, available(false)<-
true).
% return/0
% EVENT FOR RETURNING A BOOK
assimilate(return, Thy, NewThy, _, S, S) <-
  ! & call(Thy, available(false)) &
% PRECONDITION
  replace(Thy, NewThy, available(_)<-true, available(true)<-
true).
% new_book/0
% EVENT TO CREATE BOOK OBJECTS
assimilate(new_book, Thy, NewThy, _, S, S) <-
  ! & assert(Thy, NewThy, available(true)<-true).
nil
```

Figure 5. Assimilator for the class book.

Backward inference. When our system is update-oriented, it's worthwhile to store implicitly the objects' state in a trace, so that an event occurrence is not immediately processed, but is deferred until we need the state to give an answer to a query. Under this policy, object representation may be slightly different than in the former. Each theory now stores a trace, i.e., the sequence of events occurred in the life of the object it represents. If one wants to know the value of any object's attribute, it must be calculated using the code now contained in the interpreter metatheory. However, the assimilator code is now quite simple: every time an (admissible) event occurs in the life of an object, the event is added to the object's trace.

3.1. Animation of the KAOS objects

Once a KAOS schema has been defined, it is possible to (create and) use the Database defined by that schema. Users may navigate through the object society that represents the DB because they belong to another class of objects into the system, and might use the message passing mechanism for the interaction with every object in the society:

```
dest <op> service(args)
```

where `dest` is the oid of the object to which the message is addressed, `service` is one of the characteristics belonging to the object's signature (attributes or events) and `args` is an optional list of arguments for the service.

KAOS distinguishes two kinds of messages or <op>s: observations (or queries) and updates. *Observations* are messages denoted by `??` that do not alter the receiver's state, but only obtain the value of one of its attributes. On the other hand, *updates* are denoted by `::` and have as their effect the triggering of an event in the receiver's life.

Examples:

- If we want to know the value of an object's attribute, we send it an observation with an uninstantiated argument:

```
book00001 ?? available(A)
```

Note that when this observation is handled by the object's interpreter, it is transformed in a kbProlog goal to be satisfied in the theory that represents the `book00001`'s state. This, according to figure 4, will make `A` be instantiated to `true`.

- If we want to update an object, we may send it a trigger:

```
book00001 :: loan(reader00023)
```

When this update is handled by the assimilator of figure 5, it triggers a change in the object's state consisting of the update of its attribute `available`.

- *Classes are also objects*. They are characterized, among others, by their extent, and they are responsible for the notions of object factory and object store. For each classe in the OODB schema, there is an object in the society whose oid is the class name and which can respond to messages such as "create a new instance":

```
book :: new_book(NewOid, [1,"Hamlet"])
```

where the 1st argument will be instantiated to the new instance's oid after the execution of the corresponding method.

4. The KAOS OODDBMS

The basic operation features of the system (which is depicted in figure 6) are the following:

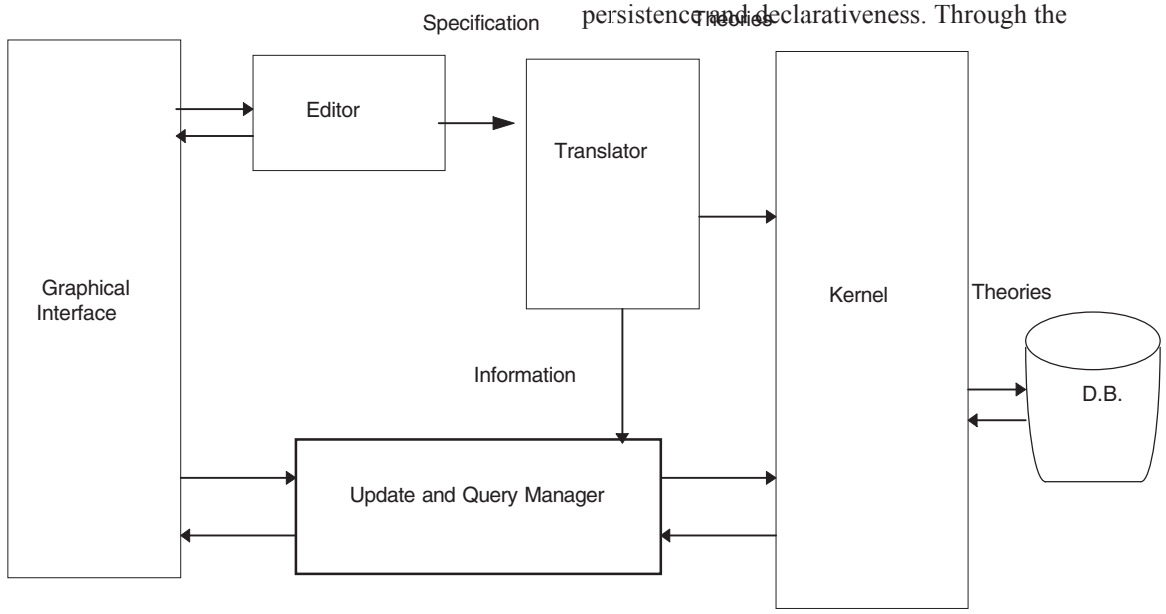


Figure 6: Architecture of the KAOS svstem.

1. With a graphical or text editor, users build the OASIS specifications that define the OODB schema.
2. A specialised translator produces a kbProlog model of the schema, following the rules derived from the implementation of the OASIS model.
3. Once the schema has been generated in the form of kbProlog (meta) theories, the Kernel builds a persistent version of the schema.
4. The interaction with the system relies on the following components:

- A Graphical Interface converts the mouse and keyboard events into strings that are sent to the Update and Query Manager (UQM).
- The UQM receives these strings and decomposes them into single messages (`??` or `::`)that are both syntactically and semantically checked, and sent to the Kernel afterwards.

The system Kernel handles every message generated in the system, makes the objects persistent, recovers the system after crashes, and has the control over the transactions, the system clock, and the parallel composition of the alive objects, etc. It is fully described in [11].

6. Conclusions and future work

The KAOS system is a DAOODBMS built over a KBMS from which we have borrowed characteristics related to persistence and declarativeness. Through the representation of OASIS objects as theories, we have transformed that language from a specification tool to a unique language to work in a OODB environment. The system functionality is quite close to the one described by the OODB Manifesto, and recent versions of both the language and the system are converging to the ODMG-93 standard proposals.

Further developments include full convergence, as well as the design of a general graphical OODB environment that will replace the current one, which in our opinion is too user-centered. We assert that users should become just one more class in the system, being another node in the class hierarchy to which our system belongs ([7]).

References

[1] Pastor, O., *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos*, Ph.D. Thesis, U.P. Valencia, 1992.

[2] Wieringa, R., *A Formalization of Objects using Equational Dynamic Logic*, in Proc. of the DOOD 91, LNCS, Springer-Verlag, 1991.

[3] Spruit, P.A. et. al., *Axiomatization, Declarative Semantics and Operational Semantics of Passive and Active Updates in Logic Databases*, Tech. Report, Vrije University, Amsterdam, 1992.

[4] Manley, John et. al.., *KBMS1: a User Manual (Version 3)*, HP-Labs. Internal Report HPL-91-178, 1991.

[5] Atkinson, M. et. al., *The Object-Oriented Database Management System Manifesto*, in Proc. of the DOOD-89, LNCS, Springer-Verlag, 1989.

[6] Cattell, R. (ed.), *The Object Database Standard: ODMG-93, Release 1.1*, Morgan Kaufmann Publishers, 1994.

[7] Canós, J.H. et. al., *Is a DOOD a Complex Object?*, Tech. Report, DSIC, Universitat Politècnica de València, 1995.

[8] Canós, J.H. et. al., *A Unified language for Deductive and Object-Oriented Databases*, Tech. Report, DSIC, Universitat Politècnica de València,1995.

[9] Ramos, I. et. al., *Objects as observable processes*, in Proc. of the 4th International Workshop on the Deductive Approach to Information Systems and Databases, Tech. Report, DLSI, UPC, 1993.

[10] Pastor, Oscar, *OASIS 2.1: An Object Definition Language for Object Oriented Databases*, Draft, DSIC (UPV), 1994.

[11] Carsí, J. *Diseño e Implementación del Núcleo de un SGBDDAOO Basado en OASIS*, M. Sc. Thesis, Facultat d'Informàtica, UPV*, 1994.*

OASIS v2: A Class Definition Language

Oscar Pastor López, Isidro Ramos Salavert, José H.Canós Cerdá
Departamento de Sistemas Informáticos y Computación.
Universidad Politécnica de Valencia.
Camino de Vera, S/N , 46071 Valencia.
Tel: 3877350; Fax: 3877357; e-mail: opastor@dsic.upv.es

Abstract. In this paper we present OASIS 2.1, the last version of OASIS. OASIS is a formal, OO Specification Language developed in DSIC ([1] [4] [6] [8] [7] [12] [11]). In its initial versions, the Software Development Environment associated to the Language was formalized using first order theories that evolve over time. This lead us to a Logic Programming environment with the complement of a for dealing with dynamics. But as will be shown later, it has a natural, sound and complete formal basis within the context of Dynamic Logic. A new formalization of OASIS according to this view is presented in this paper, in order to properly characterize objects as observable processes. The language expressiveness has been thus enriched, trying to balance ease for specification use and a precise formal framework. Additionally, language implementations have been built for rapid prototyping purposes by generating logic prototypes (implemented in logic programming languages) functionally equivalent to any given OASIS specification.

1 Introduction

Research on the integration of the OO and the deductive approach for designing Information Systems and Databases has been a major topic of interest in the last few years. The OO and deductive environment represented by **OASIS 1.0** ([1] [4] [6] [8] [7] [12] [11]) as OO Specification Language, and **OO-METHOD** ([9] [10]) as a methodological extension covering the Analysis and Design Software Production phases within an Automated Programming Paradigm, is the result of a research effort tackled during the last five years in the DSIC-UPV.

This OASIS environment was formalized using first order theories that evolve over time, within a Logic Programming environment complemented with a monitor foro dealing with dynamics. But it has a more natural, sound and complete formal basis within the context of Dynamic Logic.

A new formalization of OASIS according to this view is presented in this paper, which allows us to introduce a process algebra to properly characterize objects as observable processes.

The transition to the new language version (the so-called OASIS 2.1, referred to as OASIS from now on) is accomplished by giving a new characterization to the different kinds of formulas that, within a Dynamic Logic environment, are provided by OASIS. Additionally, the previous language version is also enriched with:

Pastor, O., Ramos, I., Canós J.H. OASIS v2: A Class Definition Language. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995.

- the introduction of the agent notion to specify which agents are allowed to activate which events, going in consequence from events to actions (which are events labeled by their valid agents)
- the introduction of a temporal logic to allow for the specification of dynamic integrity constraints
- the specification of transactions, for grouping actions in execution units.
- the definition of class objects as processes
- the possibility of defining interfaces or virtual classes to establish a protection model, fixing which view of a given object will have objects of other classes

Some basic notions coming from the Dynamic Logic theory are given in order to apply them in an OO environment. Later, we will present the formal environment associated to OASIS. We introduce a class signature, the OASIS Dynamic Logic formulas and the view of objects as observable processes. The paper will end with the presentation of an example coming from a real world application domain, to demonstrate how the Language is used in the Specification of Information Systems.

2 Dynamic Logic in an OO context

As Harel affirms in [2], the intended role of dynamic logic is to provide a natural formalism for studying certain assertions about programs in a natural and direct way.
The traditional modal logic is interpreted on a collection W of possible worlds, with a relation ρ describing accessibility between certain pairs of worlds. Typically, a sentence of the form $\Box\Phi$, read "necessarily Φ", is used to mean that the assertion Φ is necessarily true in a given world s in W, i.e. Φ is true in all worlds accessible via ρ from s. Dually, $\Diamond\Phi$ means "possibly Φ", and it is true in s if Φ is true in at least one world accessible from s (KRIPKE structure or an "allowed world" semantics). Furthermore, W can be taken to be the set of all possible worlds relevant to the execution of a program p. Thus, the accessibility relation $\rho(p)$ over W is such that (s,t) $\in \rho(p)$ iff t is a possible final state of p, with s being its initial state. In other words, there is a computation of p starting in s and terminating in t. To qualify occurrences of $\Box$,$\Diamond$ to a particular p, we will use [] and <>, resp. We will place the p inside these modal operators.
Syntactically, dynamic logics are obtained from classical ones by adding the modal operators $[p]$ and $< p >$ to formulas. Thus, $[p]\Phi$ means that Φ is true in every final state t in p, although t may not exist. The formula $< p > \Phi$ states that at least one final state t in p exists that satisfies Φ.

In our OO view, the execution of a program p is the process which represents the life of an object. Thus, every object is defined as a process composed of atomic actions called events. The state of an object is represented by valued object properties, so-called attributes. Every attribute takes a value in a particular domain. The set of attribute values in a given instant is the object state in that instant. Object evolution is

characterized in terms of changes of state. In other words, the occurrence of an event causes values of some attributes to change.

Dynamic logic is based upon two languages: a set F of first-order formulas as the abstraction of properties of states, and a set P of processes as an abstraction of changes of state. We use Φ, ... for elements of F, and p, q,... for elements of P. There is also an infinite set V of variables for which we use x,y,...

The declarative semantics of Dynamic Logic is defined relative to a given structure or model U = (W,τ,ρ), where W is a set, the elements of which are called worlds, and τ and ρ are functions.

$\tau : F \rightarrow 2^W$
$\rho : P \rightarrow 2^{W \times W}$

Informally, the mapping τ assigns the set of states in which it is true to each formula, whereas ρ assigns some binary relation between states to each process (the relation or function —in the deterministic case— which is the declarative semantics of the language) . The intended meaning is: (s,t) $\in$ ρ(p) iff execution of p can lead from state s to state t. The pair (s,t) is called a transition.

In accordance with to the previous concepts, the new ideas about the formal background of OASIS are presented in the following section.

3 Formalization of OASIS

A class in OASIS is made up of a class name, an identification function for instances (objects) of the class and a type or template that all the instances share.

The identification function characterizes the naming mechanism used by objects. It gives a set of surrogates belonging to a predefined sort or to a sort defined by the user (the so-called domains in OASIS). They are imported in the class definition. The most usual are predefined as int, nat, real, bool, char, string and date. They represent numbers, boolean values, characters, strings and dates in a particular format. New domains can be introduced in a specification by defining the corresponding Abstract Data Type.

A type is the template that collects all the properties (structure and behaviour) shared by all the potential objects of the class being considered. Syntactically, it can be formalized as a **signature**, which contains sorts, functions, attributes and events to be used; **a set of axioms**, which are formulas in a dynamic logic; and a **process query** as a set of equations with variables of sort process that are solved in a given process algebra. These variables are instantiated to the ground terms that represent possible lives of instances (objects). [1] Its semantics is given in terms of the (W, τ, ρ) structure (KRIPKE semantics).

[1] It is then a presentation of a theory in the modal logic outlined.

3.1 A signature of objects

A class signature contains:

- A set of sorts with a partial order relation. Among this set of sorts, we have in particular the sort of interest (the class name) associated with the class being defined.
- A set of functions including:
  1. those functions included in the definition of the (predefined) sorts
  2. the identification function whose sort is the ADT for identities implicitly provided with a class specification. It provides us with values of a given sort to identify objects, to assure that any object of a given class has a unique identity. [2]
  3. a set of (constant, variable and derived) attributes. They all have as domain the sort of the class, and as codomain the given sort associated to the attribute being considered.
  4. A set of events, with the sort of the class as domain (plus any additional sort representing event information) and with the sort of the class (sort of interest) as codomain. It is important to remark that this so-called sort of interest can be seen as a subsort (in the partial order relation) of a general sort *process* when objects are viewed as processes.

     As presented in the OO Analysis models of OO-METHOD [9], every event occurrence will be labelled by the agent that is allowed to inititate it. Following the approach taken by [14] when dealing with this agent notion, we will write **i:e** if the agent i initiates event e, and we will call i:e an action. i could be the environment or any object of a system class. This way we can represent the notion of set of object services as the interface that allows other objects to access the state. They can be events (server view) or actions (client view) depending on whether these services are offered or requested. Actions become services requested by an object, by which it can consult or modify other object's state (or its own state).

     In OASIS, the ideas of initiative of the event execution and visibility of events are separated by introducing the agent and view notions respectively. With the declaration of the OASIS interfaces we represent which events are visible to which agents. Generally speaking, agents represent those who execute events, while interfaces express which events are visible to which agents.

[2] For specification purposes, we will introduce an identification mechanism consisting of the declaration of one or more key maps used as aliases for identifying objects. They are similar to the candidate key notion of the relational model. These maps, from a given key value, return its associated object identity. Key maps will be declared as (tuples of) constant attributes. This approach is the same taken by Troll to identify objects in its recent revised version ([3])

3.2 OASIS Dynamic Logic Formulas

In an OASIS class template we have the following kinds of dynamic formulas:

- **Evaluations**; they are formulas of the form $\Phi\,[a]\,\Phi'$ whose semantics is given by defining a ρ function that, from a ground action a returns a function between possible worlds.
  $$\rho(a) \in W \rightarrow W$$
  In other terms, every possible world for an object is a valid state. The ρ function determines which transitions between objects states are valid after the execution of action a.
  It is important to realize that, within this dynamic logic environment, the formula Φ is evaluated in s $\in$ W, and Φ' is evaluated in $\rho(s)$, with $\rho(s)$ the world after the execution of the action being considered.
- **Derivations** are formulas of the type $\Phi \rightarrow \Phi'$. They allow us to define derived attributes (Φ') in terms of the given derivation condition (stated in Φ).
  They basically differ from the evaluation formulas in the fact that this derived evaluation is done in a unique state.
- **Integrity Constraints**; they are formulas that must be satisfied in every world. We distinguish between static and dynamic.
  Static integrity constraints are those defined for every possible world. They must always hold.
  On the other hand, dynamic integrity constraints are those relating different worlds. They require the use of a temporal logic, with the corresponding temporal logic operators.
- **Preconditions** are formulas with the template $\neg\Phi\,[a]\,false$, where Φ is a formula that must hold in the world previous to the execution of action a. Only in the worlds where Φ holds, is a allowed to occur. If $\neg\Phi$ holds, the occurence of a gives no state as successor.
- **Triggers**. They are formulas of the form $\Phi\,[\neg a]\,false$. $\neg a$ is the action negation. If Φ holds and another action other than a occurs then there is no successor state. This forces a to occur or the system remains in a blocked state.
  Thus, triggers are actions activated when the condition stated in Φ holds.

In any of these dynamic formulae, Φ, Φ' are well-formed formulas in a first order logic (explained in [5]). These first order logic wff's usually refer to a given system state characterized by the set of values attached to attributes of objects in the world considered.

3.3 Objects as Observable Processes

In OASIS, an object is defined as an observable process. The process specification in a class allows us to specify object dynamics and determines the access relationship between the states of instances. Processes are constructed by using **events** as atomic actions. But the designer also has the choice of grouping events in execution units, so-called **transactions**.

The *molecular* units that are the transactions follows an *all-or-nothing* policy with respect to behaviour as events occur: when a failure happens during a transaction execution, the resultant state will be the initial one. The execution of processes are represented by terms in a well-defined algebra of processes, which is based on the approach presented by [13]. It allows us to declare possible object lives as terms whose elements are transactions and events. As shown in [5], every process can be rewritten to a term in a Basic Process Algebra $BPA_{\delta\epsilon}$, with the $\bullet$ (sequence) and $+$ (alternative) process operations. This provides an implementation of concurrency based on arbitrary interleaving.

3.4 Semantics

The semantics of OASIS is given in terms of the (W, τ, ρ) structure. W is the set of all possible states for an object to achieve. Every object state is a set of pairs (a, v), where a is an attribute and v is its value in such a state.
For every kind of first order formula representing any state property (object attribute values or static integrity constraints), the set of states in which it is true fix the mapping τ.
Finally, the transition function ρ determines the valid transitions between object states (worlds). In this context, the evaluations state what the valid transitions are; preconditions forbid certain transitions be relevant (those where the given condition does not hold in the initial state); and triggers introduce obligation by forcing an action (and its induced transition) to be activated when a condition is satisfied or another event occurs.
As an object is an observable process, the introduction of a mechanism to specify class objects as processes is a basic topic. This is what is done in the process section of a class specification.
Next, to finalize this section, we are going to present a formal definition for the notion of class type (class template).

3.5 Formal definition of class template

According to the previous concepts, we define a class template as follows:

Definition 3.1 *We call* class-template *to* { *A, E, Φ, Process*}*, where*

- *A is the attribute alphabet*

$$\{\ a_i : naming - sort \rightarrow valuation - sort\}$$

- *E is the generic event alphabet. For each* $e \in E$ *we can obtain* $\underline{e} = \sigma e$*, with* σ *being a ground substitution.*
- *Φ is a set of dynamic formulas in a dynamic logic, that includes those presented in this report:*
  1. *Static and Dynamic Integrity Constraints.*
  2. *Preconditions.*
  3. *Triggers*

4. *Valuations.*
5. *Derivations*

– *Process is the chosen process algebra (implicit) + process queries (seen as a set of equations between process terms).*

4 An example

To complete this general view of OASIS 2.0, we give an example taken from an accounting application. An organization controls its funds, dividing the money between several items, so-called **programs** (not computer software programs!). An initial amount of money is assigned to each program. Every expense is always charged to one program's **available** money, which is then reduced. The program can spend more than it has available, so it can have a **debit balance.** Periodically, money spent is replaced, so that more expenses could be made. Before an expense is formalized, money can be paid to suppliers in advance. Consequently, granted advance payments must be controlled, although they do not reduce the program's available money.

A larger version of this example is developed step by step in [5]. Within the scope of this paper, a part of the specification dealing with the PROGRAM class is presented in the Appendix.

We will briefly discuss some relevant aspects:

– Let us see how the available amount of money and debit balances are updated when an expense is incurred. If the program is in the red, only the debit balance is increased by the amount of the expense. Otherwise, if there is enough available money to meet the expense, the available money is decreased. But if the available money is not enough, it is completely used and the debit balance is set to the difference.

  ADM and ACC are administrator oids and accountant oids respectively, and act as event agents in this case. USR is a generalization of both of them.

```
valuation A, N : nat;
[ADM:new'program] available=initial'money and debit=0 and
                  advanced=0.
available=A [ACC:expend(N)] available=A-N.
red=false and available=A [ADM:expend(N)]
                          if A¿=N then available=A-N
                          else  available=0 and debit=N-A.
red=true and debit=D [ADM:expend(N)] debit=D+N.

red=false and available=A [USR:restore(N)] available=A+N.
red=true and debit=D  and D¿=N [USR:restore(N)]
                          if D¿=N then debit=D-N
                          else  debit=0 and available=N-D.
[new'program] advanced=0.
advanced=A [advance(N)] advanced=A+N.
advanced=A [return(N)] advanced=A-N.
```

– An integrity constraint for programs is stated as the prohibition of having an available amount of money higher than that initially assigned. This formula must hold in every state.

```
available¡=initial'money.
```

- A precondition for the event advance would be stating that the program is not in the red and the advanced amount of money is not greater than that available. We write:
  advance(N) if red=false and advanced¡=available
- Conditions may serve to trigger events when they are satisfied. In programs, when red state is reached and the advanced money has surpassed a given upper limit, money must be replaced. We will declare the following trigger:
  self::restore(N) if red=true and advanced ¿ limit

5 Conclusions

An enriched version of OASIS has been presented, including the capability of specifying transactions and processes. Objects are viewed as observable processes within a logic dynamic framework, keeping the deductive properties attached to the OASIS expressiveness and, at the sane time, formalizing the procedural view of objects by using an Algebra of Processes for Objects.
The tools and methods of the OO-METHOD methodology developed at DSIC-UPV hav been updated to include this added expresiveness, keeping the methodology faithful to the Automated Programming Paradigm within an OO model. From the specification perspective, the other relevant improvements are:

- introduction of actions
- introduction in OASIS of a Temporal Logic to deal with dynamic integrity constraints
- views to fix the user-system interface, and to allow these views to:
  1. hide some class properties.
  2. modify structure and behaviour of objects according to specific needs.
  3. view an object from different perspectives.

Further research work will focus on the generation of code in industrial and graphical software development environments, the use of the language as Object Definition Language for OO Data Bases, the verification of the language expressiveness in real complex Software Systems and real time extensions to adapt the language to the characteristics of real time systems.

APPENDIX

```
class program
identification
     by_program_code: (program_code);
constant_attributes
     program_code:nat
     program_desc : string;   # program description
```

```
     initial_money : nat;     # program's initial
     limit: nat;  # upper limit of money that can be advanced when the program is in red
variable_attributes
     available : nat;    # money available at a given time
     debit : nat;        # debit balance of program
derived_attributes
     red : bool;         # the program is in the red
private_events N : nat;
     new_program    new;
     destr_program  destroy;
     expend(N);     # money is expended
     restore(N);    # money previously expended is restored
shared_events
     advance(N) with supplier; # money is paid in advance
     return(N) with supplier;  # money previously paid in advance is returned
     hire with supplier;        # a supplier is formally hired
     fire with supplier;        # a supplier is fired
     justify with supplier;     # a supplier justifies an advanced payment
valuation A, D : nat; ADM : administrator; USR : user; SUP : supplier;
     [ADM:new_program] available=initial_money and debit=0 and advanced=0.
     available=A [ACC:expend(N)] available=A-N.
     red=false and available=A [ADM:expend(N)]
                                if A>=N then available=A-N
                                else  available=0 and debit=N-A.
     red=true and debit=D [ADM:expend(N)] debit=D+N.
     red=false and available=A [USR:restore(N)] available=A+N.
     red=true and debit=D and D>=N [USR:restore(N)]
                                     if D>=N then debit=D-N
                                     else  debit=0 and available=N-D.
derivation
     red = (debit>0).
constraints
  static
     available <= initial_money.
  dynamic
     sometimes (red=false) since (red=true).
preconditions  ACC : accountant;
     ACC:expend(N) if available>=N.
     advance(N) if red=false and advanced<=available.
     destr_program if available=0 and advanced=0.
triggers
     self :: restore(N) if red=true and advanced>limit.
transactions
    JUSTIFICATION= USR:justify(N) USR:expend(N)
processes
     PROGRAM = ADM:new_program PROGRAM0.
     PROGRAM0 = (ADM:advance(N) + SUP:return(N) +  [ADM,ACC]:expend(N) + USR:restore(N) +
                    USR:JUSTIFICATION + ADM:hire + ADM:fire) PROGRAM1.
     PROGRAM1 = PROGRAM0 + ADM:destr_program.
end_class
```

References

1. Canós,J.H.:Pastor,O. *Object Oriented and Functional Specification of Information Systems* in Proc. of DEXA-91 Conference, Berlin,1991 351-386, Sept.1981.
2. Harel,D. *Dynamic Logic* in Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pags. 497-694. Reidel 1984
3. Hartmann,T.; Kusch,J.; Saake,G.; Hartel,P. *Revised Version of the Conceptual Modeling and Design Language Troll* Proc. of the International IS-CORE Workshop, pags.89-102; Amsterdam-1994.
4. Pastor,O.:Hayes,F.:Bear,S. *OASIS: An Object Oriented Specification Language* In the Proceedings of the CAiSE-92 Conference, Springer-Verlag, pag.348-363; Mayo 1992, Manchester (UK)
5. Pastor,O.;Ramos,I.;Cuevas,J.;Devesa,J.;Canos,J.H. *OASIS 2.0: An Object Definition Language for Object Oriented Databases* Technical Report, DSIC-UPV, Febrero 1994
6. Pastor O. *Diseño y Desarrollo de un Entorno de Producción Automática de Sotfware basado en el modelo OO* PhD thesis, DSIC-UPV, May 1992.
7. Pastor,O.:Ramos,I:,Canos,J.H. *Object Oriented and Relational Specification Of Information Systems* In the Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS, Aiguablava (Catalonia), 1991
8. Pastor,O;Casamayor,J.C. *A Deductive and Object Oriented Environment for Software Production* In the Proc. of the Workshop in Expert Systems, AI and Software Engineering Application, ILPS-91, San Diego (California).
9. Pastor,O.;González, A. *OO-METHOD: An Object-Oriented Methodology for Software Production* in the Proceedings of the DEXA-92 Conference, Valencia (Spain)
10. Pastor,O.;Ramos,I.;Canos,J. *From Analysis to Design in an OO and deductive environment* in the Proceedings of the DEXA-93 Conference, Prague (Chec Republic).
11. Ramos,I. *Logic and OO Databases: a Declarative Approach* Proc. of the DEXA 90, Springer-Verlag 1990
12. Ramos,I:,Pastor,O.:Canos,J.H. *On the use of Algebras as Semantic Domains of Object Societies* In the Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS, Roses(Catalonia), 1992
13. Wieringa R.J. *Algebraic Foundations for Dynamic Conceptual Models* PhD thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1990.
14. Wieringa R.J.;Meyer J.-J.Ch. *Actors, Actions and Initiative in Normative System Specification* Annals of Mathematics and Artificial Intelligence, 1992.

This article was processed using the LaTeX macro package with LLNCS style

Entorno general de producción automática de prototipos de software orientado a objetos: OOAP (Object-Oriented Assistant Prototyper)[1]

PROYECTO PASO (Plan de acción software)

Grupo OASIS[2]

Isidro Ramos
Dpto. de Sistemas Informáticos y Computación
Camino de Vera, s/n
46071 Valencia
e-mail: iramos@dsic.upv.es

Ambrosio Toval
Dpto. de Informática y Sistemas
Campus de Espinardo
37001 Espinardo - Murcia
e-mail: atoval@dif.um.es

Resumen

El trabajo que presentamos describe un proyecto de investigación aplicada, actualmente en realización, en el ámbito de la Ingeniería del Software. El objetivo es conseguir un entorno de producción automática de prototipos, a partir de especificaciones realizadas en el lenguaje de especificación orientado a objetos OASIS, para ayudar en la fase de Ingeniería de Requisitos, basándose en técnicas formales de especificación y en estándares gráficos de interfaz de usuario (X-Window). El proyecto se enmarca dentro del programa PASO (financiado con fondos mixtos nacionales, del Mº de Industria, y europeos, de la DGXIII de la CEE) y se realiza en colaboración con empresas (PYME's) de desarrollo de software. En el trabajo se hace una descripción técnica del proyecto y también se describen las experiencias y lecciones aprendidas en la preparación y puesta en marcha de proyectos de este tipo

1.INTRODUCCIÓN.

El trabajo que describimos a continuación corresponde a un proyecto enmarcado dentro del programa PASO (Plan de acción software) del CDTI (Centro para el Desarrollo Tecnológico Industrial, Ministerio de Industria) que gestiona fondos europeos y nacionales para financiar este tipo de investigaciones.

El consorcio formado para abordar el proyecto consiste de tres empresas (PYME's de desarrollo de software) y dos departamentos universitarios:

Empresas: BASE, de Murcia; INFORGES, de Murcia; MACRO, de Alicante

[1]Proyecto parcialmente subvencionado por el programa PASO (Plan de acción software) del CDTI, con fondos mixtos de la CEE (DGIII) y del Mº de Industria.

[2]El grupo OASIS está constituído por los dos equipos siguientes:

OASIS Valencia: Isidro Ramos, Oscar Pastor, José Hilario Canós, Vicente Pelechano, M.Carmen Penadés, Begoña Bonet. Dpto. Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

OASIS Murcia: Ambrosio Toval, Jesús G. Molina, Mariano Flores, Manuel González, José Luis Fernández, Joaquín Nicolás. Dpto. Informática y Sistemas. Universidad de Murcia.

Ramos, I., Toval, A., García-Molina, J., Flores, M., González, M., Fernández, J. L., Nicolás, J., Pastor, O., Canós, J. H., Pelechano, V., Penadés, M.C., Bonet, B. Entorno General de Produccion automática de prototipos de software orientado a objetos: OOAP (Object-Oriented Assistant Prototyper). Actas de las I Jornadas de Informática. Asociación Española de Informática y Automática (AEIA). 1995.

Departamentos: Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia; Departamento de Informática y Sistemas. Universidad de Murcia.

El proyecto tiene una duración estimada de 21 meses a partir del 1 de Febrero de 1995, según la memoria inicial del mismo.

El objetivo de nuestra propuesta consiste en diseñar e implementar una herramienta CASE (Computer-Aided Software Engineering) de "muy alto nivel" dirigida a soportar fundamentalmente la fase de especificación de requisitos en proyectos de desarrollo de software. Junto con este entorno formal de modelización conceptual llamado OOAP ("Object-Oriented Assistant Prototyper"), se ofrecerá una estrategia para conseguir prototipos funcionales [Case 86] orientados a objetos de los sistemas de información a construir [Luqi 89 92b] [Davis 92], y se experimentará todo ello sobre casos reales aportados por las PYMES.

El entorno propuesto, OOAP, permitirá, en concreto:

1- la especificación en OASIS de las componentes, y sus relaciones, del sistema de información, basada en el paradigma orientado a objetos (OO), mediante una interfaz gráfica adecuada.

2- la simulación (ejecución de las especificaciones) de las características operacionales del sistema de información en desarrollo mediante prototipado interactivo conjuntamente con los usuarios, ayudándoles a expresar, definir y validar sus requisitos. La necesidad de desarrollar urgentemente este tipo de herramientas ha sido reconocida recientemente por importantes autores en Ingeniería del Software [Brooks 87] [Harel 92] [Hsia et al. 93].

3- la elaboración de unas especificaciones claras, validadas y verificadas de los requisitos de los usuarios, como documento final de salida de la etapa conocida como "ingeniería de requisitos" necesarias para producir software de calidad, integradas con diferentes metodologías de desarrollo de software OO.

Además de esta introducción, en la siguiente sección se detallan los objetivos marcados por el proyecto, mientras que la sección 3 proporciona una descripción técnica de sus componentes; en la sección 4 se revisan las decisiones adoptadas hasta ahora, y su justificación. Finalmente en la sección 5 se resumen las lecciones aprendidas (no sólo técnicas) en relación con la preparación y puesta en marcha del proyecto.

2.OBJETIVOS DEL PROYECTO

Distinguimos los siguientes objetivos básicos:

Objetivos básicos:

1- Diseñar una herramienta general CASE de "muy alto nivel" que permita la definición, prototipado y validación de especificaciones de sistemas de información en un ambiente gráfico y dentro de un paradigma orientado a objetos.

2- Introducir la tecnología orientada a objetos en PYMES de desarrollo de software, en forma realista y crítica, que permita una valoración sobre su incidencia en el proceso de desarrollo de software, costes asociados y beneficios obtenidos en caso de utilización masiva de las mismas.

3- Experimentar el modelo formal OO procedente del proyecto ESPRIT (IS-CORE ESPRIT BRA WG 3023 [IS-CORE 91 93]) y el lenguaje OASIS, como técnica general de especificación de requisitos desde una perspectiva de orientación a objetos. Otros lenguajes basados en este modelo son OBLOG [Sernadas et al. 91], TROLL [Saake Jungclaus 91] y TESORO [Torres et al. 94]. OASIS, para sistemas abiertos y activos, ha sido desarrollado, y cuenta con prototipos ya existentes, en el Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia [Pastor 92] [Pastor et al. 92] [Toval 94] [Toval et al. 94] en el ámbito de dos proyectos de investigación de la CICYT (SÍNTESIS: TIC 88-0572, 88/92 y PRÓTESIS: TIC 91-0425, 92/94).

4- Construir un prototipo evolutivo de OOAP sobre workstation en entornos abiertos (UNIX) que sea accesible desde PC, junto con la documentación asociada y una estrategia de utilización en el ciclo de vida OO.

Identificación de los resultados del proyecto

Productos:

Construcción de un prototipo de interés comercial de OOAP: Software y documentación asociada (manual de usuario, tutorial sobre la utilización de OOAP, documentación técnica).

Elementos Modulares:

1) Núcleo: formado por el Gestor de Esquemas y el gestor de la Base de Información.

2) Interfaz de usuario: comprende un editor gráfico y otro textual de OASIS, un traductor de OASIS a la representación interna del núcleo y un monitor de comportamiento para realizar acciones y consultas sobre el núcleo.

3) Sistema de Soporte a la Ejecución: es básicamente el sistema de deducción del lenguaje de especificación ejecutable (LE) utilizado como base

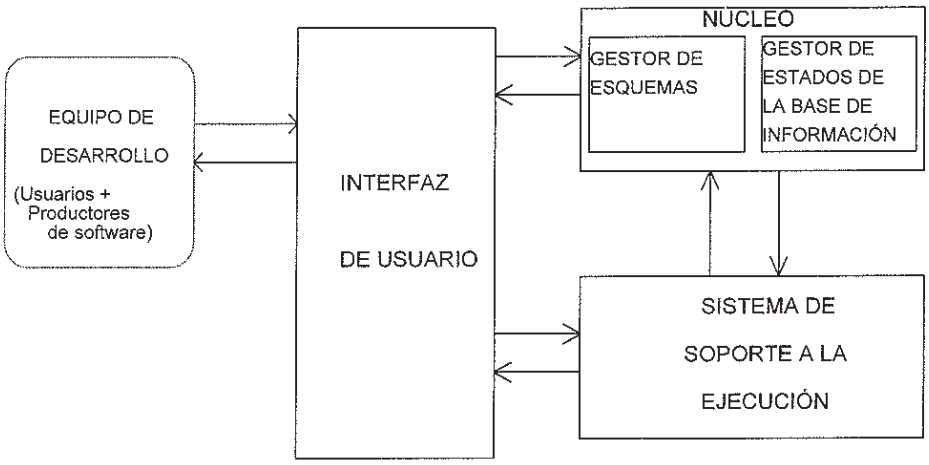


Figura 1. Principales componentes de OOAP

Servicios: Instalación, formación y mantenimiento asociados al producto, que serán ofertados a los futuros clientes de OOAP.

Metodología: Guía sobre cómo utilizar OASIS y OOAP en el desarrollo de sistemas de información y software asociado.

3.DESCRIPCIÓN TÉCNICA DEL PROYECTO

Innovaciones y ventajas: Este proyecto contiene varios aspectos innovadores desde un punto de vista práctico. Como producto-herramienta CASE de utilización comercial, cabe decir que el presente trabajo aparece en un momento en el que por parte de las empresas y Administraciones Públicas se empieza a hacer un uso, aún tímido, de metodologías estructuradas y herramientas CASE de soporte a las mismas ya asentadas en otros países desde hace más de 15 años.

Es de esperar que esta misma tendencia se repita con las metodologías y herramientas CASE basadas en el paradigma de orientación a objetos, por lo que el proyecto que hemos concebido

goza de la ventaja de la **oportunidad** ya que es el momento de producir herramientas y soporte a un paradigma que, según todas las previsiones (congresos, revistas especializadas,etc.) se impondrá en los próximos años frente (o de forma complementaria) al estructurado. Hay que decir a este respecto que en la actualidad existen muchas herramientas CASE para dar soporte a metodologías orientadas a objetos (por supuesto procedentes de empresas extranjeras) pero éstas básicamente se limitan a dar soporte gráfico y fundamentalmente sintáctico sobre las especificaciones de los sistemas de información y software asociado. OOAP incorpora una **innovación básica:** la de **permitir la especificación y prototipado del comportamiento del sistema de información además de su estructura.** Solo conocemos un producto similar, aún en fase de experimentación, de la empresa OBLOG Software, de Portugal, denominado OBLOG CASE, que fue presentado en la Conferencia Internacional "TOOLS'94" en París.

La utilización de las técnicas formales subyacentes permitirán la comprobación automática y verificación de la corrección de las especificaciones, detectando automáticamente tanto errores sintácticos como cierto tipo de errores semánticos introducidos por los especificadores del sistema. El uso de técnicas formales facilitará además la evolución de la herramienta hacia otras etapas del ciclo de vida OO, en particular y de forma más importante hacia la implementación de bases de datos y generación de código, dentro del conocido paradigma de programación automática [Balzer et al. 83] [Agresti 86].

Tecnología a aplicar: La característica más destacable del proyecto presentado es sin duda la utilización y puesta en práctica de **tecnologías de la información innovadoras y vanguardistas** en el ámbito de la Ingeniería del Software aplicada. Nuestro objetivo es el de realizar una utilización conjunta de todas ellas para ayudar a consolidar estas tecnologías en nuestro país, ahora que aún se está a tiempo para contribuir a disminuir las (actualmente) grandes dependencias del exterior, tradicionales desafortunadamente en tecnologías clásicas de software y sistemas de información.

La tecnología propuesta tiene que ver con las siguientes líneas (algunas de ellas establecidas expresamente por el programa PASO, como prioritarias):

a– Metodologías orientadas a objetos
b– Ingeniería del software y herramientas de desarrollo avanzadas
c– Especificaciones formales
d– Ingeniería de Requisitos y Prototipado

a-Metodologías orientadas a objetos

Es la metodología a la que da soporte el producto a construir, OOAP, y a la vez una de las metodologías a utilizar en su construcción. Este último hecho contribuirá a aumentar el conocimiento de las mismas y podrán ser transferidas como valor añadido al "know-how" de las empresas participantes en el consorcio que las podrán incorporar a sus métodos habituales de trabajo. Se ha realizado ya un primer conjunto de actividades formativas dirigidas a las empresas, en Análisis y Diseño OO.

b-Ingeniería del software y herramientas de desarrollo avanzadas

Tanto el producto a construir como los métodos utilizados en el proceso de construcción del mismo forma parte íntegramente de la disciplina de "Ingeniería del Software" (y en particular de la subdisciplina de "Ingeniería de Requisitos"). OOAP es una herramienta de desarrollo avanzada de "muy alto nivel", en el sentido de dar soporte a las primeras etapas del ciclo de vida orientado a objetos de desarrollo de software de propósito general. Simultáneamente, en el proceso de construcción de OOAP y junto con las metodologías de Orientación a Objetos, ya citadas anteriormente, se utilizarán herramientas CASE avanzadas (OBLOG Case) lo que aportará un doble beneficio: incrementar la productividad en el desarrollo de OOAP y a la vez servir de punto de referencia para incrementar las características funcionales del mismo.

c- Especificaciones formales

La utilización de técnicas formales aportan rigor y carácter de verdadera disciplina científica a la Ingeniería del Software. Se utilizarán como fundamentos formales, además de la lógica soportada por el LE finalmente adoptado, los modelos formales orientados a objetos propuestos por el grupo IS-CORE ESPRIT BRA WG 3023 en el desarrollo de sistemas de información. El uso de técnicas formales en Ingeniería del Software es defendido y recomendado en la actualidad por autores, y revistas,prestigiosos en Ingeniería del Software [Halang Krämer 94] [Hsia et al. 93][Gaudel 94] y que empiezan ya a tener una cierta presencia industrial [Hall 95].

Las técnicas formales son utilizadas siempre de forma "transparente" a los usuarios: tanto a los usuarios directos de OOAP (clientes tipo, empresas desarrolladoras de software), como a los usuarios de los prototipos de software de propósito general creados mediante OOAP (clientes habituales de las empresas desarrolladoras de software) Esto se consigue mediante las "interfaces de usuario" que permiten una interacción agradable con iconos y distintos tipos de gráficos, entre OOAP y sus usuarios, que son así ajenos al uso de

formalismos, interno a la herramienta. En particular, se trabajará con sistemas basados en X-Window.

d-Ingeniería de Requisitos y Prototipado

OOAP se establece como una herramienta CASE de soporte básicamente a las tareas de Ingeniería de Requisitos, mediante prototipado operativo y funcional de los requisitos de los usuarios realizado de forma interactiva conjuntamente entre productores de software y los propios usuarios. La necesidad de desarrollar urgentemente herramientas CASE de este estilo (en especial para el paradigma de orientación a objetos), ha sido también reconocida recientemente por numerosos autores en Ingeniería del Software [Brooks 87] [Harel 92] [Hsia et al. 93][Andriole 94]. La figura 2 muestra un esquema del proceso de prototipado:

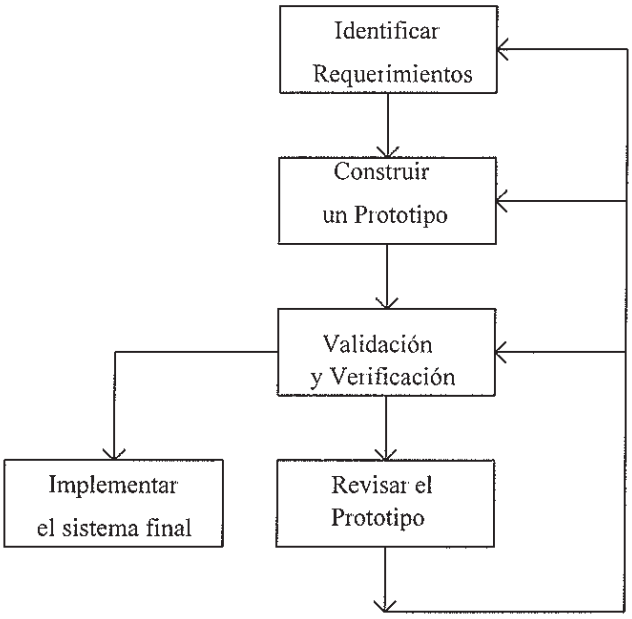


Figura 2. La estrategia de prototipado

Adecuación a los estándares internacionales: No existe en este momento una normativa ampliamente aceptada sobre el uso del paradigma orientado a objetos, o sobre la estrategia de prototipado en el desarrollo de software. Adoptaremos, sin embargo, las ideas procedentes del grupo OMG (Object Management Group) y de sus estándares de objetos y de BDOO, ODMG [Atwood et al 94]. Nuestro proyecto puede, sin embargo, contribuir a la consecución de estándares del modelo OO al experimentar y utilizar una de las primeras propuestas formales procedentes del ya mencionado proyecto ESPRIT IS-CORE

4. CRITERIOS DE SELECCIÓN DE ALTERNATIVAS

El tipo de decisiones que se han tomado hasta ahora corresponden a alguno de los dos grupos siguientes: sobre la tecnología a utilizar (hardware, lenguajes de desarrollo) y sobre el aspecto metodológico.

Respecto a la **tecnología** se han tomado decisiones sobre:

- El tipo de máquina y sistema operativo utilizados como plataforma de desarrollo
- El lenguaje de especificación ejecutable a utilizar
- Software de comunicaciones
- El estándar gráfico utilizado para la interfaz de usuario

Los criterios que se han seguido han sido:

- Seleccionar en primer lugar los requisitos de software que son los que condicionan el hardware necesario.
- Utilizar tecnología normalizada o, al menos, la más extendida dentro de cada grupo
- Posibilitar el uso de la herramienta producida no solo por grandes empresas de desarrollo de software, sino también para PYMES a un coste razonable.

Algunas de las decisiones más importantes, y su justificación, han sido las siguientes:

- En relación con la selección de candidatos a LE, considerar OBJ3 Y KB-Prolog como lenguajes de especificación ejecutables frente a otros candidatos como AXIS, ACT ONE, etc. en el caso funcional, debido a que la última versión (release 2.0, OBJ3) tiene un rendimiento aceptable, y permite interconexiones con LISP y X-window; en relación con el caso lógico, por la expresividad de KB-Prolog y su comunicación con el lenguaje de BDOO KBMS, de HP. No se descarta, en cualquier caso, la migración a otro lenguaje análogo con mayores prestaciones, si ésto supone un mejor desarrollo del proyecto.

- Utilizar estaciones de trabajo como sistemas abiertos, con sistema operativo UNIX y entornos gráficos basados en el estándar X-window debido a su normalización y al alto rendimiento de ejecución.

- Posibilitar la comunicación entre las estaciones de trabajo y PC's ordinarios con el fin de abaratar costes

La utilización de workstations (UNIX), como plataformas de desarrollo, frente a PC´s se debe básicamente, además del criterio de rendimiento, a que los lenguajes de especificación ejecutables considerados, no están disponibles sobre PC. Se ofrece la solución de posibilitar el acceso desde PC´s con software tipo "PCXware" o bien mediante "LYNUX". No hay que olvidar que el objetivo de este proyecto es construir un buen prototipo, no un producto final. En una siguiente fase de este proyecto (posiblemente a través de una petición a ESPRIT) se planteará el transporte a ambientes PC´s de todo el entorno.

Con respecto a los **aspectos metodológicos**, se han tomado decisiones sobre:

- La metodología de desarrollo a soportar por parte de la herramienta producida
- La fase o fases del ciclo de vida que soportará la herramienta
- La metodología seguida para desarrollar la propia herramienta

siguiendo los siguientes criterios:

- Adoptar las directrices más actuales en Ingeniería del Software para la producción de software de calidad, que garantice la supervivencia del producto a largo plazo.
- Adoptar las recomendaciones más recientes sobre producción de herramientas CASE, por parte de autores de reconocido prestigio en Ingeniería del Software.
- Seguir las recomendaciones y experiencias de otros grupos de investigación, en particular IS-CORE de ESPRIT, sobre la utilización de técnicas formales (matemáticas) en el desarrollo de sistemas de información.

En relación con este grupo, algunas de las decisiones más importantes, y su justificación, han sido:

- Utilizar la aproximación OO como paradigma metodológico al que dar soporte, debido a que se ha erigido como una de las aproximaciones con más futuro y con mayor actividad en el ámbito de la Ingeniería del Software actual.
- Utilizar el lenguaje de especificación OO, OASIS, como herramienta de descripción inicial de los sistemas a prototipar y desarrollar.
- Utilizar Análisis Estructurado (sólo en las fases iniciales) y OASIS para el desarrollo del producto OOAP y CASE de soporte disponible para la primera.
- Abordar fundamentalmente las fases iniciales del ciclo de vida ("very upper CASE").

5. LECCIONES APRENDIDAS.

La experiencia adquirida en la elaboración y puesta en marcha de esta propuesta para el programa PASO del CDTI nos ha dado la oportunidad de aprender unas cuantas lecciones, no sólo desde el punto de vista técnico, sino desde una perspectiva socio-técnica. Por la posible utilidad que pudiera tener para futuras peticiones nacionales o europeas (PASO es muy similar a ESPRIT en cuanto a las formas y procedimientos de solicitud y aprobación), resumimos estas lecciones en los siguientes puntos:

De las personas:

Hay una cierta tendencia a infravalorar el trabajo de los profesionales de la Universidad que se traduce en algunos casos en la recomendación de no incluir en los gastos el concepto de *retribuciones para profesorado*. En las mismas propuestas y para el mismo proyecto, sin embargo, sí se admiten retribuciones para profesionales de empresa. Ésto parece injusto, ya que a diferencia de otros proyectos (como los de la CICYT) éste tipo de actuaciones no se contempla como apoyo a la actividad investigadora, sino como impulso a la internacionalización de las empresas en el área de I+D (Órden 12435 29/4/93, BOE nº 113 de 12/5/93). En resumidas cuentas, defendemos que una actividad "extraordinaria" del profesorado en este tipo de actividades, adicional a su trabajo habitual, debería ser reconocida al menos en la misma cuantía en que se reconoce a los profesionales de empresas.

De las relaciones empresa/Universidad, en materia de Informática.

Nuestro caso particular confirma el "gap" existente en nuestro país entre el trabajo en las empresas y en la Universidad en materia de Ingeniería del Software. Se hace imprescindible un proceso previo de formación y transferencia tecnológica desde la Universidad hacia las empresas (uno de los objetivos del proyecto). A la inversa, el conocimiento de esta realidad nos ha conducido a planteamientos más "prácticos" de los trabajos realizados en la Universidad, lo que no está reñido con el rigor en las investigaciones y los modelos teóricos utilizados: creemos que en algún sentido ésto nos puede hacer "más ingenieros". Por otro lado la experiencia adquirida en el proyecto y en la relación con las empresas, es válida para futuras peticiones nacionales y/o europeas.

De la gestión administrativa de los proyectos.

Durante el desarrollo del proyecto, los participantes contratarán con la Fundación Universidad-Empresa de Murcia o con sus Oficinas de Transferencia de Tecnología (OTRI).

la gestión contable y administrativa del consorcio. Esta labor de la Fundación, comenzó con la elaboración de la propuesta resumida ante el CDTI, haciendo labores de intermediación e interlocución entre participantes del consorcio y entre éstos y el CDTI. En el caso particular de los departamentos universitarios españoles tradicionalmente las tareas administrativas (preparación de memorias, gestión contable de los proyectos, etc...) ha recaído sobre los propios investigadores, lo que en nada beneficia a los proyectos desde el punto de vista de su desarrollo técnico. El poder contar con apoyo humano en estas labores, a través de las oficinas OTRI o de las fundaciones Universidad-Empresa (cuando este apoyo es "real") es un factor muy importante a tener en cuenta, por el tiempo que consumen estas actividades, que no es nada desdeñable.

Se tiene previsto que este funcionamiento siga en marcha, y que a través de una comunicación contínua y directa entre Fundación y miembros, se coordine la colaboración entre los participantes, apoyando con su infraestructura la realización de tareas y la entrega de informes.

Existe, finalmente, otro problema que, de no contar con recursos propios suficientes, puede ser serio para un departamento universitario: éste se refiere al hecho de que PASO ingresa el dinero por semestres, a hito realizado, lo que significa que en teoría el departamento (que suele ser financiado al 100%) se tiene que gastar en ese primer semestre un dinero que en realidad no tiene, salvo que consiga encontrar algún "mecenas" que se lo adelante. Como dijimos antes, puede ser un serio problema.

6. BIBLIOGRAFÍA Y REFERENCIAS.

[Agresti 86] W.W. Agresti "New Paradigms for Software Engineering" IEEE Computer Society Press 1986

[Andriole 94] "Fast, Cheap Requirements: Prototype, or else!", S.J Andriole. IEEE Software. March 1994

[Atwood et al. 94] T.Atwood; J.Duhl; G.Ferran; M.Loomis; D.Wach "The Object Database Standard: ODMG-93". R.G.G. Cattell (ed.) Morgan Kauffman Publisher, Inc. 1994.

[Balzer et al. 83] Balzer,R.; T.E. Cheatham; C.Green "Software Technology in the 1990s: A New Paradigm" IEEE Computer, vol. 6, n° 11, November 1983, pp. 39-45

[Brooks 87] F.Brooks "No Silver Bullet: Essence and Accidents of Software Engineering". Computer, vol.20,n° 4 April 1987, pp 10-19

[Case 86] A.F Case,Jr. "Information Systems Development: Principles of Computer-Aided Software Engineering" Prentice-Hall 1986

[Davis 92] A.M. Davis "Operational prototyping: a new development approach", IEEE Software. September 1992, vol. 9, n 5

[Gaudel 94] "Formal Specification Techniques", M Claude Gaudel. Proc. of the 16 Int Conf on Software Engineering-ICSE'16 IEEE 1994

[Halang Krämer 94] Wolfgang Halang; Bernd Krämer "Safety Assurance in Process Control", IEEE Software, January 1994, pp.61-68

[Hall 95] "Formal Methods can deliver provable software", Nina Hall. Scientific Computing World March 1995.

[Harel 92] D Harel "Biting the Silver Bullet", COMPUTER, vol. 25, nº 1, January 1992, pp.8-20

[Hsia et al 93] Pei Hsia, Alan Davis, David Kung. "Status Report: Requirements Engineering", IEEE Software, Nov 1993

[Luqi 89] Luqi Naval Postgraduate School "Software Evolution Through Rapid Prototyping", IEEE COMPUTER vol.22, n 5, May 1989 pp.13-25

[Luqi 92a] Luqi Naval Postgraduate School "Computer-Aided Prototyping for a command-and-control system using CAPS", IEEE Software vol 9, n.1,January 1992 pp 56-68

[Luqi 92b] Luqi. Naval Postgraduate School "Status Report: Computer-Aided Prototyping", IEEE Software vol 9, n 6, November 1992 pp.77-81

[Pastor 92] O Pastor "Diseño y Desarrollo de un entorno de producción automática de software basado en el modelo orientado a objetos", PH.D. dissertation. Departament de Sistemes Informatics i Computació. Universitat Politecnica de Valencia(Spain). April 1992

[Pastor et al. 92] O Pastor; Hayes.F.; S Bear "OASIS: An Object-Oriented Specification Language" in Proceedings of the CAISE-92 Conference, Spriger-Verlag (1992)

[Saake Jungclaus 91] G. Saake; R. Jungclaus. "Specification of Database Applications in the TROLL language". En Harper, D. (ed.) Proc. Int. Workshop on the Specification of Database Systems, Glasglow 1991 Springer-Verlag London 1991

[Sernadas et al. 91] Sernadas, A. ; Sernadas, C. Gouveia, P. ; Resende, P "OBLOG: An Informal Introduction" Research Report INESC 1991

[Torres et al. 94] "Desde el Lenguaje de Especificación Orientado a Objetos TESORO a LOTOS", J. Torres, J.A Troyano, M.Toro. Revista de Informática y Automática de la AEIA, Vol. 27, Nº2

[Toval 94] A. Toval "Formalización algebráica de un entorno de producción automática de prototipos orientados a objetos", Tesis Doctoral. Departamento de Sistemas Informáticos y Computación Universidad Politécnica de Valencia. Mayo 1994

[Toval et al 94] "Prototyping Object Oriented Specifications in an Algebraic Environment". A.Toval, I Ramos, O.Pastor. En "Database and Expert Systems Applications" (D.Karagianis, Ed.). LNCS nº 856 Springer-Verlag 1994

[Wieringa Meyer 93] R.J. Wieringa; J.J. Ch. Meyer "Actors, Actions, and Initiative in Normative System Specification" to be published in Annals of Mathematics and Artificial Intelligence.

An Object Oriented Methodological Approach for Making Automated Prototyping Feasible*

Oscar Pastor,Vicente Pelechano, Begoña Bonet,Isidro Ramos
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Camí de Vera, S/N, 46071 València (Spain)
Tel: 3877350; Fax: 3877357; e-mail: { opastor-pele}@dsic.upv.es

Abstract

We present the OO-Method, an Object-Oriented Methodology that complements the use of formal specification languages with the background of the classical informal OO Analysis and Design proposals. This methodology eliminates the ambiguity of such methods and puts into practice the appropiate properties of formal specification languages in a practical way. It uses a formal OO Specification Language called Oasis as a design tool. The most relevant contribution of the OO-Method is that it generates the specification in an automated way from the graphical models obtained in the Analysis step and the quoted specification is a robust documentation of the System. Finally, code generators make possible to obtain implementations of the specification in different Software Development Environments, including declarative (Prolog-based) and imperative environments.

1.- Introduction

In the last few years an important activity has been developed in the area of Object-Oriented (OO) Analysis and Formal OO Specification Languages. Several OO Methodologies based on the practice acquired during the Software Development Process have been proposed (OMT [Rum91], OOSE [Jac92], Wirfs-Brock [Wir90], Booch[Boo94], Fusion [Col94]), and OO Specification Languages have been elaborated with a solid mathematical basis (OASIS [Pas95], Oblog [Ser87], Troll [Har94).

The research work developed at the DSIC (Technical University of Valencia) over the last five years has led to a methodological proposal called the OO-Method ([Pas92-3,Pas93]) that covers the classical Software Production phases of Analysis, Design and Implementation and includes the conclusions reached within the context of the Object-Oriented Model and its formalization ([Ram90,Ram92,Pas92-2]).
The OO-Method provides an advanced graphical environment

- including graphic models for dealing with System Analysis
- using Oasis as an OO and a Formal Specification Language
- allowing automated prototyping in both declarative and imperative Software Development Environments ([Ram90,Pas92-2])

Next, after giving a brief overview of the Object-Oriented basic notions attached to OO-Method, the Methodology is presented focussing on its Analysis Models and in the way in which a specification in Oasis is generated from them.

* This work has been supported by the Spanish Industry Ministry and the DGXIII of the EEC under the PASO PC-310 Project and also by the CICYT under OASIS TIC 94-0557 Project.

Pastor O., Pelechano V., Bonet B., Ramos I. An object Oriented Approach for making Automated Prototyping Feasible. Proc. of the International Conference on Database and Expert System Applications (DEXA 96). 1996.

2.- The OO basic notions in OO-Method/Oasis

From an intuitive point of view, an object can be viewed as a cell or capsule with a state and a set of services. The state is a local memory that changes throughout the object's life. It has several individual components called attributes. Every attribute has a value at every instant, and the set of attribute values characterizes the object state. The state is hidden to other objects and can be handled only by means of services. The set of services is the interface which allows other objects to access the state. They can be methods or actions, depending on whether they are offered or requested.

Methods are services offered by an object, through which other objects (or itself) can consult or modify its state. In this case these other objects act as agents. Actions are services requested by an object which acting as an agent, can consult or modify another object's state.

We have, in fact, two views for system objects: a client's view, dealing with those actions that a given object can activate by itself; and a server's view, dealing with those methods provided by a given object to the system.

To modify an object state, methods and actions are represented as events and transactions. Object evolution is characterized in terms of state changes. Events represent atomic of state changes, in the sense that they occur in an instant of time and change the attribute values involved. Events are grouped into transactions, which are molecular execution units. If we label an event with (a list of) its potential actors, we can say that we have an action.

When we want to build a system specification, we need to specify classes, which represent a collection of objects, and which all share the same properties. Every existing object in the class is said to be an instance of the class. Classes have a naming mechanism that manages the way in which instances are named and which provides new objects with identifiers. Object identifiers (oid) are provided by data types.

If Oasis is used as an OO specification language, a class has a template or set of properties shared by every instance. Basically this template must allow for the declaration of an identification mechanism, the signature of the class including attributes (static view), methods and actions (dynamic view), and finally a set of formulae of different kinds to cover the rest of the class properties:

- integrity constraints (static and dynamic) which state.
- conditions that must be satisfied.
- valuations which to explain how attributes are changed by events.
- derivations which relate some attribute values to others.
- preconditions which determine when an event can be activated.
- triggers which introduce internal system activity.

Finally, as an object can be defined as an observable process, a class definition should be enriched with the specification of the process attached to the class. This process will allow us to declare possible object lives as terms whose elements are events and transactions, combined with the well-known alternative and sequence process operators. A complete description of the Oasis language can be found in [Pas95]. The OO-Method has been created on this formal basis, as we will show in the following sections.

3. The OO-Method :An Object Oriented Methodolgy

The *OO-Method* is an Object-Oriented Software Production Methodology whose objective is to join the appropiate properties of Formal Specification Languages with the pragmatism and productive spirit present in the usual informal Methodologies. The phases of the OO-Method are shown in Figure 1.

We start with an Analysis step where three models are generated: the Object Model, the Dynamic Model and the Functional Model. They describe the Object Society from three complementary points of view within a well-defined OO framework. For these models we have preserved the names used in many other well-known and wide-used OO methodologies (especially OMT), even if the similarities are purely syntactic as can be seen throughout this paper.

When the Analysis phase is completed and we have the three quoted models, the formal and OO Oasis specification (the OO-Method design tool) is generated in an automated way. This is done through a translation process. The resultant specification acts as a complete system repository, where all the relevant properties of the component classes are included.

A prototype which is functionally equivalent to the specification is built in an automated way, from this specification,. This may be done both in declarative environments (Prolog-based) and imperative environments (specially those visual OO programming environments that are widely used nowadays). The main contributions of OO-Method are the two described as follows:

1. The facility of automatic code generation from the Models obtained during the Analysis step by applying two processes of translation (from the graphical models to Oasis, and from Oasis to the selected programming environment.
2. The blending of classical OO methods with formal specification languages, with their different backgrounds: software development practice on the one hand, and a mathematical theory background on the other hand.

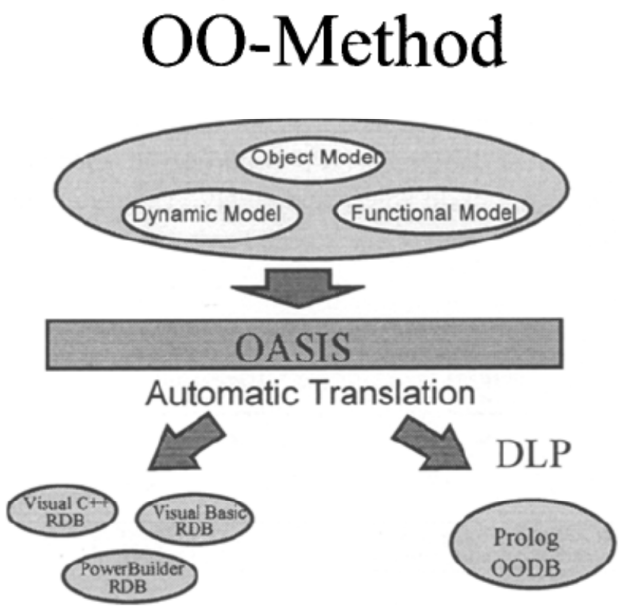


Fig. 1. Phases of OO-Method

3.1 Analysis

Object Model.

The Object Model is represented by means of a Class Configuration Diagram (CCD), a graphic model where system classes are declared, including their attributes and services. Aggregation and inheritance hierarchies are also graphically depicted representing class relationships. Additionally, agents are introduced to specify who can

activate each class service. Classes are the basic modelling units. A class is represented by a rectangle with three areas:

- a header with the class name.
- a static component where attributes are declared.
- a dynamic area where services are introduced, distinguishing among new and destroy events, and among private and shared events.

Shared events are connected by solid lines in the CCD. Client classes (agents) of a given service are represented by dotted lines joining every potential client class with the corresponding server class.

The OO-Method deals with complexity by introducing aggregation and inheritance hierarchies. In a CCD, we represent the aggregation relationship between two classes including its cardinality (minimum and maximum), to determine how many components can be attached to a given container and how many containers a component class can be associated with. The graphic representation is shown in Figure 2.

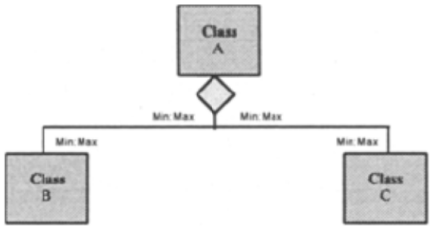


Fig. 2. Aggregation in a DCC.

Inheritance is graphically depicted as an arrow from a given subclass to its superclass. This arrow can be labeled with a condition of specialization, or with the events that activate/cancel the child role respectively. This graphical representation is introduced in Figure 3[1].

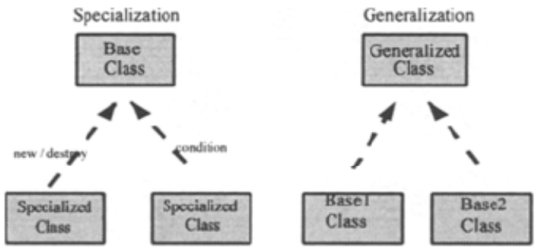


Fig. 3. Inheritance in a CCD.

Next, the CCD corresponding to a Library Information System (fully developed in [Pas96-3]) is shown. As a basic explanation (for reasons of brevity), we assume that as usual in such a System, there are readers, books and loans relating a

[1] This is how inheritance is dealt with in Oasis (Pas95), distinguishing between permanent and temporal specialization. The permanent case refers to child instances created when the ancestor instance is created, and they need a condition built on constant attributes. Temporal specialization (role) appears when a superclass event makes a child object play the role corresponding to the specialized class.

book to the reader who orders it. Readers can 'play the role' of unreliable readers, if their return dates expire.

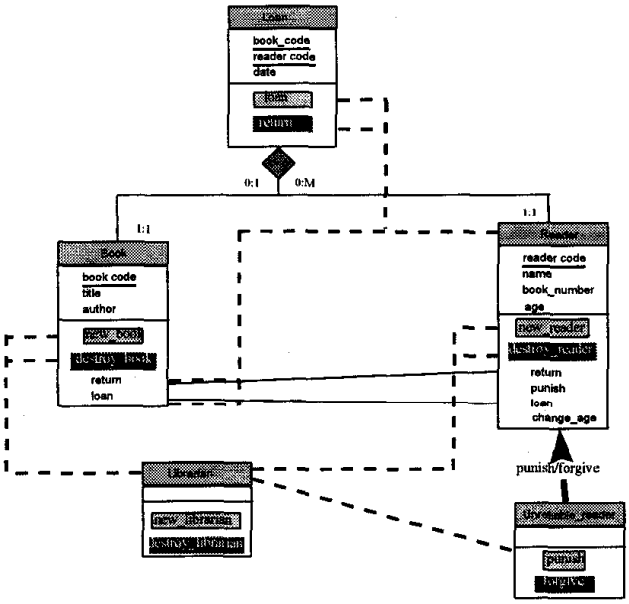


Fig. 4. CCD that represents the Object Model of the Library Information System.

Dynamic Model.

The Dynamic Model is used to specify valid object lives and interobject interaction. To describe valid object lives, we use State Transition Diagrams (STDs, one for each class). To deal with object interaction, we introduce an Object Interaction Diagram (OID, one for the whole System).

State Transition Diagram

STDs are used to describe correct behaviour by establishing valid object lives. By valid life, we mean a right sequence of states that characterizes the correct behaviour of the objects for every class. In this context states denote the different available situations for class objects, and are depicted using a circle labeled with the state name.

When an object does not exist, a black circle represents this "state" of non existence, and will be the source of initial transition labeled by the corresponding *new* event. A bull's eye is used to represent the post-mortem state.

Transitions represent valid changes of state that can be constrained by introducing conditions. They follow the syntax shown below:

event | action | transaction **[if** *precondition*] [**when** *control condition*]

where *precondition* and *control condition* are conditions defined on the object attributes that must hold for a service to occur in *precondition*, and to avoid the possible non-determinism for a given action in *control condition*. An example of STD can be seen in Figure 8.

Object Interaction Diagram

The object interactions are represented by diagrams of this kind. We declare two basic interactions:

- triggers, which are services of a given class activated in an automated way when a condition is satisfied by an object of the same or another class.

- global interactions, which are transactions involving services of different classes.

Basically, we represent classes in the OID as boxes with a header including the class name. Class services are declared as smaller boxes inside the corresponding class box. The class service boxes are connected when one of the previous types of interactions is defined. Triggers are introduced by starting the corresponding solid line in the header of the class and ending it in the triggered action. The general model for an OID can be seen in Figure 10.

Functional Model.

After declaring object attributes and services in the Object Model and valid life cycles and object interactions in the Dynamic Model, the aim of the Functional Model is to capture semantics attached to any change of state in an easy and an intuitive way. The main idea is to describe how event activation in a given state changes the attribute values. This model specifies the effect of an event on its relevant attributes through an interactive dialogue. The value of every attribute is modified depending on the action that has been activated, the involved event arguments and the present object state.

The specification of an action effect should be done declaratively, as proposed in Oasis ([Pas95]). However, a good specification requires a solid formal basis for any Analyst. To resolve this situation, the OO-Method provides a model where the Analyst only has to categorize every attribute among a predefined set of three categories and introduce the relevant information depending on the corresponding selected category.

This classification of attributes ([Pas92-2]) is a contribution of this method and gives a clear and simple strategy for dealing with the task of generating the Functional Model. At the same time it opens the door to include this information in a specification in Oasis (the Design tool in OO-Method) in an automated way.

There are three types of attributes: *push-pop*, *state independent* and *discrete-domain based* attributes.

Push-pop attributes are those whose events increment or decrement their value by a given quantity. Events that reset the attribute to a given value can also exist.

An example of this category is the book_number of the reader class, with REA:loan as incremental action and REA:return as decremental one (REA is assumed to be a variable of type reader).

Attribute : book_number		Category : push-pop	
Incr.Acc.	Decr.Acc.	Reset Acc.	Effect
	REA : return		- 1
REA :loan			+ 1

Fig. 5. Push-pop attribute *book_number* of the *reader* class

The **state-independent** attributes have a value that only depends on the latest action that has occurred. Once a relevant action is activated, the new attribute value of the object involved is independent of the previous one. In such a case we consider that the attribute remains in a common state, having a certain value for the corresponding attribute. We can introduce the attribute bookshelf of the book class as an example. A book has a bookshelf assigned when the event locate(B) is activated. When this event occurs, bookshelf takes the argument value independently of any previous value.

Attribute : bookshelf	Category : state-independent.
Carrier Acc	Effect
LIB :locate(B)	= B

Fig. 6. State-independent attribute *bookshelf* of the *book* class

Discrete-domain valued attributes take their values from a limited domain. The different values of this domain model the valid situations that are possible for objects of the class. Through the activation of carrier actions (that assign a given domain value to the attribute) the object reaches a specific situation. The object abandons this situation when another event occurs (a "liberator" event). As an example, let's consider the available attribute of the book class. The available value tells us what the present book situation is. The carrier event (loan) lets the object into a situation where available has the value false. The situation is abandoned when the event return is activated.

Attribute : available	Category : discrete-domain valued	
Present value	Carrier Act.	New value
true	REA:loan	false
false	REA:return	true

Fig. 7. Discrete-valued attribute *available* in the *book* class.

3.2 Design : OASIS generation from the Analysis Models

The design process in OO-Method obtains a formal OO specification from the resultant Analysis models. The main property of this process is that the specification is generated in an automated way by using a translator that converts graphic information into a textual OO specification. This specification, which is written in Oasis, constitutes a complete, formal System Repository.

Object Model Information Management

The whole system classes are obtained from the Object Model. Elementary classes in the CCD are translated into elementary Oasis classes, where we declare attributes (constant, variable and derived), static integrity constraints, derivation formulas for derived attributes and (private and shared) events. All this information is included in the source Object Model. Complex classes of the resultant Oasis specification come from every class relationship defined in the CCD with aggregation and inheritance (specialization and generalization).

Dynamic Model Information Management.

Trigger relationships, event preconditions, process definition attached to any class and global interactions are obtained from the Dynamic Model with its two diagrams: the State Transition Diagram (STD) and the Object Interaction Diagram (OID).

Given a class and its corresponding STD, the information obtained includes *preconditions* and *processes*. Preconditions appear as transition labels, and process definition is achieved from the paths that appear in the diagram. Specification paragraphs that are obtained from the STD in Figure 8 for the class book can be seen in Figure 9.

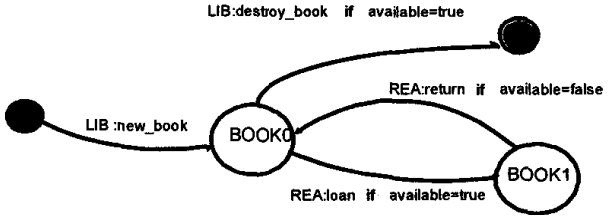


Fig. 8. STD corresponding to the class *book*

As an example, we obtain the following precondition and process sections of the corresponding Oasis specification from the previous diagram.

preconditions	**processes**
REA :return **if** available=false	BOOK=LIB:new_book • BOOK
REA :loan **if** available=true	BOOK0=REA :loan • BOOK1 + LIB :destroy_book
LIB : destroy_book **if** available=true	BOOK1=REA :return • BOOK0

Fig. 9. The precondition and process sections for the *book class.*

The trigger and global interaction sections of the resultant specification are also obtained in an automated way from the OID, where all relevant information is included. In Figure 10 we see the OID corresponding to the library example, from where we obtain a trigger declaration for the book class or a global interaction (see Figure 11) representing an execution involving an event of the reader class followed by an event of the librarian class.

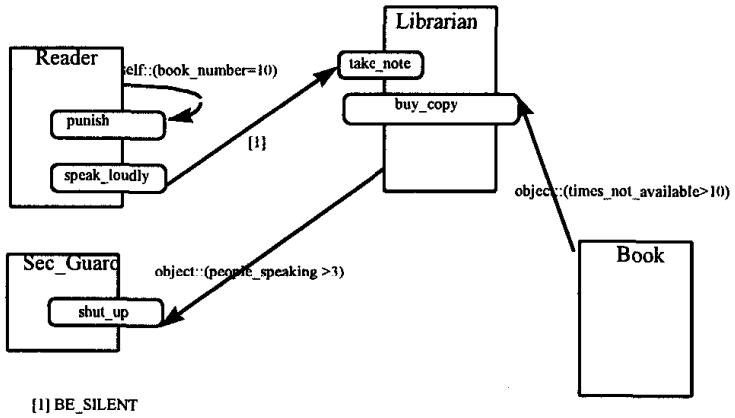


Fig. 10. (simplified) Object Interaction Diagram for the Library

triggers

object : :buy_copy if times_not_available>10

global interactions

BE_SILENT = REA :speak_loudly • LIB :take_note

Fig. 11. *Book* class trigger obtained from the OID.

Functional Model Information Management

Valuation formulas of the specification in Oasis are directly obtained from the information extracted during the dialogue with the Analyst while generating the Functional Model. Let's see how to deal with every attribute category.

On one hand, the problem is easy to solve when dealing with *discrete-domain valued* attributes. As attribute values denoting valid situations,as well as carrier actions,

have been introduced during the functional modelling process we can directly build valuation formulas. In Fig. 12 a), we show valuation formulas related to Figure 7 which were generated in the previous Analysis step.

On the other hand, when dealing with *push-pop* attributes, we know the incremental and decremental actions together with the function that determines the particular modification to be applied to the object when the involved event is activated. We also know all potential reset actions as well as the corresponding arguments which determine the resultant attribute value. For instance, for attribute *book_number* of class *book* we have the valuations shown in Fig. 12 b).

Finally, in the case of having state-independent attributes, the information stored during the functional modelling process contains those actions which assign a new value to the involved attribute. This value is determined by an event argument. The formula in Fig. 12 c) refers to this situation and corresponds to the information introduced in Figure 6.

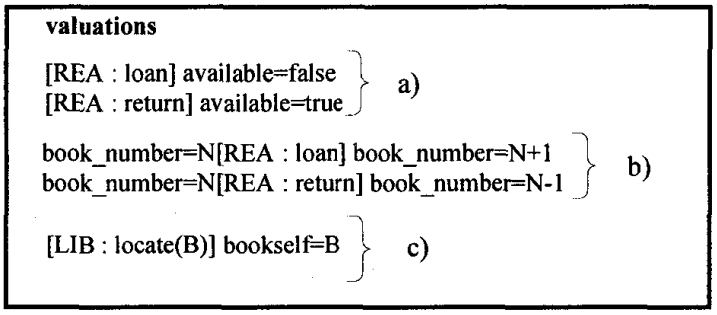


Fig. 12. Valuation formulas for all the attribute caterories.
a) the discrete-domain *available* attribute of the class *book*.
b) the state-independent *bookself* attribute of the class *book*.
c) the push-pop *book_number* attribute of the class *reader*.

4.- Conclusions

We have presented the main features of the OO-Method proposal in keeping with the strict space limitations, focussing on its basic contributions: the blending of well-known OO methods with formal specification languages and the automatic design and implementation from graphic models obtained in the Analysis step. In this way, the OO-Method comes close to the ideal of a rapid prototyping Software Development environment, for creating complex, industrial applications which are portable across many platforms.

At present, graphic tools have been developed to support the Methodology in both declarative and imperative environments ([Pas96-2]) using Relational and Object-Oriented servers. A Windows-based CASE tool is about to be delivered in the context of a University-Enterprise Contract managed by the Computing Technology Institute of Valencia. The objective of this project is to generate an industrial CASE tool which is fully OO-Method compliant, using well-known visual programming environments (Delphi, PowerBuilder,...) for code generation.

References

[Boo94]Booch,G. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company,Inc. 1994

[Har94] Hartmann T.;Saake,G.;Jungclaus,R.;Hartel,P.;Kusch,J. *Revised Version of the Modelling Language Troll (Troll version 2.0)* Technische Universitat Braunschweig, Informatik-Berichte, 94-03 April 1994.

[Jac92] Jacobson I.,Christerson M.,Jonsson P.,Overgaard G., *Object Oriented Software Engineering , a Use Case Driven Approach.* Reading, Massachusetts. Addison - Wesley

[Pas92-1]Pastor,O.,Hayes,F.,Bear,S. *OASIS: An Object Oriented Specification Language,* Proceedings of the CAiSE92, LCNS 593, pp.: 348-363, Springer Verlag

[Pas92-2]Pastor, O. *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos.* Phd Thesis, managed by Isidro Ramos, 1992, DSIC-UPV (Valencia).

[Pas92-3]Pastor,O. *OO-METHOD: An Object Oriented Methodology for Software Production.* Proceedings of the DEXA 92, Springer-Verlag, pp. 121-127. ISBN: 3-211-82400-6

[Pas93] Pastor,O.,Ramos,I.,Canos J.H. *From Analysis to Design in an Object Oriented and Deductive Environment.* Proceedings of the DEXA 93, LCNS 720, pp. :667-673, Springer-Verlag,

[Pas95] Pastor,O.: Ramos, I. *Oasis 2.1.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach,* February 94 (1st ed.), March 95 (2nd ed.), October 95 (3th ed.).

[Pas96-1]Pastor,O.;Pelechano,V.; Bonet,B. ; Ramos, I. *OO-Method 2.0 : una metodología de Análisis y Diseño Orientada a Objetos.* Technical Report, DSIC-UPV, 1996.

[Pas96-2]Pastor,O.;Merseguer,J.;Romero,J.;Barberá,J.M. *Descripción del entorno gráfico de la herramienta CASE OO-METHOD.* Technical Report , DSIC-UPV, 1996.

[Pas96-3]Pastor,O.;Pelechano,V.; Bonet,B. ; Ramos, I. *Using OO-Method : a practical guide to software modelling.* Technical Report , DSIC-UPV, 1996.

[Ser87] Sernadas,A.;Sernadas,C.; Ehrich,H.D. *Object-Oriented Specification of Databases: An Algebraic Approach* In P.M.Stocker, W.Kent eds., Proceedings of VLDB 87, pp.-107-116. Morgan Kaufmann, 1987

[Ram90] Ramos, I. *Logic and OO Databases: a Declarative Approach* Porceedings of the Dexa 90, Springer-Verlag, 1990

[Ram92] Ramos,I.:Pastor,O.:Canos,J.H. *On the use of Algebras as Semantic Domains of Object Societies* In the Proceedings of the 3rd International Workshop of the deductive Approach for DB an IS, Roses (Costa Brava), 1992.

[Rum91] Rumbaugh J.,Blaha M., Permerlani W., Eddy F.,Lorensen W. *Object Oriented Modeling and Design.* Englewood Cliffs, Nj. Prentice-Hall

[Wir90] Wirfs-Brock R.,Wilkerson B.,Wiener L., *Designing Object Oriented Software.* Englewood Cliffs NJ. Prentice- Hall.

Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic.

Jose A. Carsí, Isidro Ramos, Mª Carmen Penadés, Vicente Pelechano
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Camí de Vera, s/n, 46071 València
☎ 3877350; Fax: 3877357
e_mail { pcarsi | iramos }@dsic.upv.es

Resumen

En la búsqueda de formalismos que permitan definir las características de los modelos orientados a objetos, la Transaction Frame Logic (TF-Logic) destaca por su adecuación a la hora de reflejar dichas características. En este artículo se presenta una primera aproximación a la descripción del modelo subyacente al lenguaje OASIS a través de la implementación del concepto de clase OASIS en TF-Logic.

1 Introducción

Ultimamente una de las preocupaciones de los investigadores ha sido la búsqueda de los fundamentos teóricos que dan soporte a los modelos orientados a objetos.

El modelo orientado a objetos subyacente al lenguaje de especificación formal OASIS ([PR96]) ha sido, desde hace unos años, motivo de investigación en este sentido. Algunas aproximaciones seguidas han sido: programación lógica concurrente [Let93], especificaciones algebraicas [Tov94], lógica clausal dinámica [Can96] y redes de petri [San96]. Dichas aproximaciones han tenido, normalmente, un elevado coste implementacional, debido a la lejanía entre los conceptos objetuales y los formalismos usados para describirlos.

Uno de los últimos formalismos aparecidos es la Transaction Frame Logic propuesta por Michael Kifer en [Kif95]. La TF-Logic es una lógica de orden superior que se está revelando como una herramienta adecuada para expresar las características de los modelos objetuales.

Del estudio de los fundamentos de la orientación a objetos subyacentes en el modelo propuesto por el grupo OASIS, surgió la idea de la metaclase ([RPP95a, RPP95b]): una clase que describe el propio modelo orientado a objetos y que permite, al ser usada, definir las características de los sistemas de información.

En el presente artículo damos un esbozo de las posibilidades de la TF-Logic para describir las características que incorpora el modelo orientado a objetos subyacente a OASIS a través de la implementación de la metaclase.

El documento está estructurado de la siguiente manera: trás la introducción, en el segundo apartado se da un repaso al concepto de metaclase y las ventajas que aporta. En el tercer apartado se realiza una pequeña introducción de la Transaction Frame Logic. Por

Carsí, J.A., Ramos, I., Penadés, M.C., Pelechano, V., Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic. Actas de las I Jornadas de Trabajo en Ingeniería del Software, 1996.

último, en el cuarto apartado se aborda la especificación de algunos aspectos importantes de la metaclase.

2 Metaclase

El concepto de metaclase en OASIS se define como una clase compuesta que recoge todas las características de dicho modelo orientado a objetos. Esquemáticamente, la metaclase será un editor abstracto de clases OASIS, a través de cuyos servicios (crear_clase, añadir_atributo, ...) se crean las clases OASIS.

OASIS proporciona tres unidades de diseño que son: las clases primitivas, las clases elementales y las clases complejas. La metaclase permite al usuario construir clases de cualquiera de estas tres categorias. Por lo tanto, la metaclase ofrece los servicios de un editor de clases primitivas, un editor de clases elementales y un editor de clases complejas. Ver figura 1.

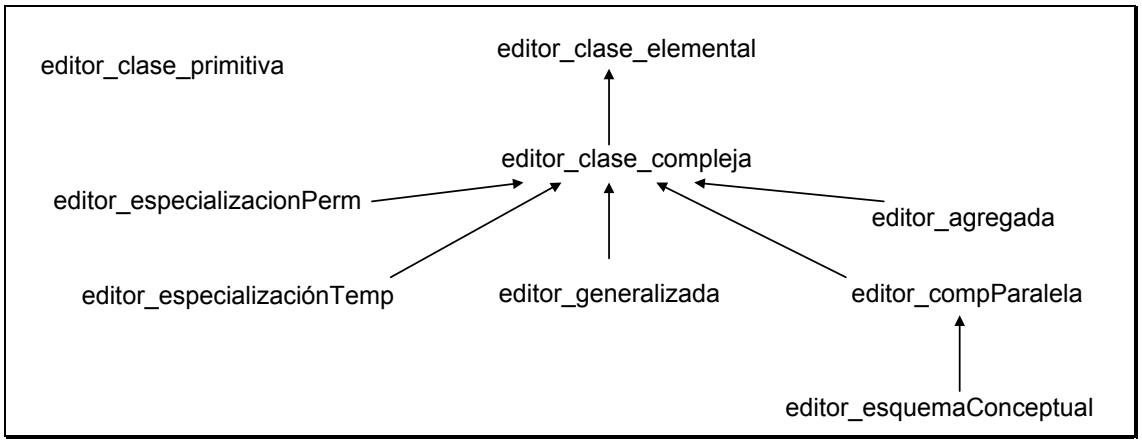


Figura 1: Representación jerárquica de la metaclase.

Una de las principales características de la metaclase consiste en que sus instancias no son sólo objetos sino que, además, son clases de objetos. En la figura 2 podemos ver una instancia de la metaclase con su doble vertiente clase/instancia.

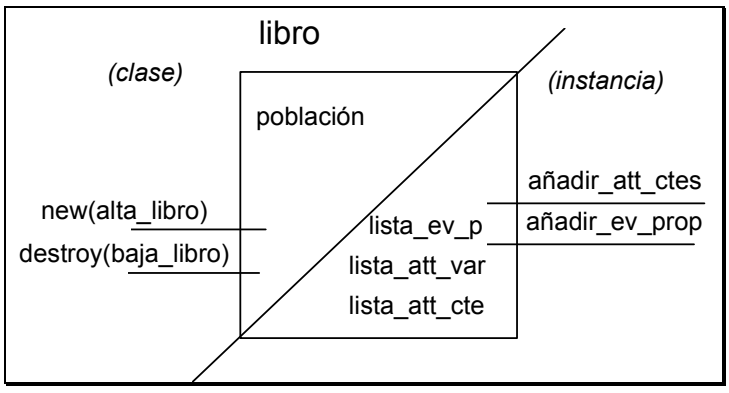


Figura 2: Doble visión de las instancias de la metaclase.

Debido a su doble visión, otra característica importante de la metaclase estriba en que los aspectos de evolución del esquema están totalmente integrados en la misma[i]. Esto es así, porque el disparo del, por ejemplo, evento que añade un atributo a una clase, puede provocar la adecuación de todos los objetos involucrados (instancias de la clase modificada) para ser consistentes con el nuevo estado (migración de objetos). Ver figura 3.

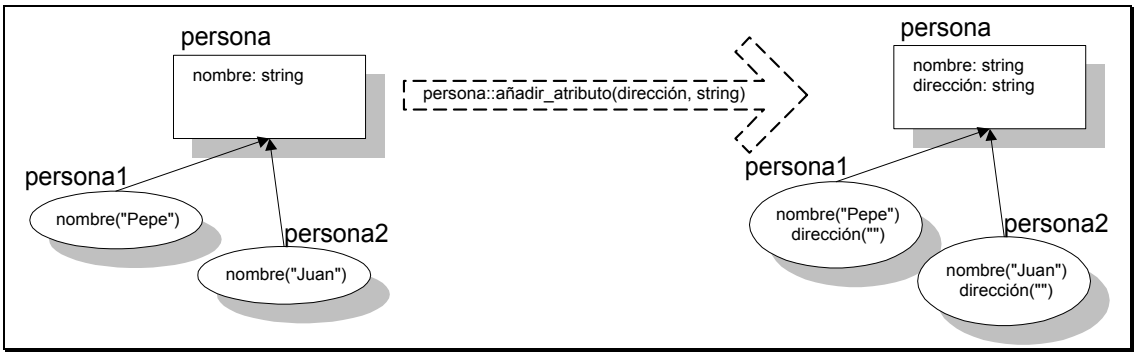


Figura 3: Ejemplo de evolución del esquema y migración de objetos.

La metaclase es el corazón de un entorno CASE-OO que permite realizar el análisis y el diseño de los sistemas de información orientados a objetos, siguiendo la visión aportada por OASIS.

El entorno resultante tiene una doble perspectiva. Por un lado, es un entorno de producción de software orientado a objetos y, por lo tanto, soporta entre otras características la facilidad de crear nuevas clases. Por otro lado, está basado en el paradigma de la programación automática y permite el prototipado rápido de las instancias de las clases que se crean como instancias de dicha metaclase.

3 Transaction Frame Logic

Aunque no es el objetivo del presente artículo dar una descripción exhaustiva de la TF-Logic, haremos una pequeña introducción para entender el resto del artículo.

La TF-Logic es una lógica de orden superior resultante de la unión de dos lógicas, la *Frame Logic* ([KLW95]) y la *Transaction Logic* ([KB95]), que permite describir sociedades de objetos de una forma declarativa, sencilla y formal.

En TF-Logic tenemos, por un lado, la *base de objetos* que contiene el estado actual de la sociedad de objetos. El estado es manipulable y evoluciona a lo largo del tiempo conforme se van activando transacciones en el sistema. Por otro lado, existe un *conjunto de transacciones* que manipulan dicho estado. Dicho conjunto de transacciones no es susceptible de ser manipulado y permanece inalterado con el paso del tiempo.

El estado se representa mediante hechos y reglas en F-Logic y las transacciones se describen usando las características de la T-Logic.

3.1 Características de F-Logic.

Se usan **oid** lógicos para referirse a los objetos. Un oid es un término de primer orden construido a partir de funciones y variables. Algunos identificadores de objetos son:

```
13, _256, pepe32, padre(maria), ...
```

Un oid puede identificar tanto a un objeto como a una clase, lo que permite manipular los datos (objetos) y el esquema (las clases) con el mismo lenguaje.

Los **atributos** son cuasi-funciones que almacenan el estado de los objetos. En el siguiente ejemplo vemos el objeto `pepe` con los atributos `nombre`, `salario` e `hijos`.

```
pepe[ nombre->"Pepe Carsí" ; salario-> 400000 ; hijos->{}]
```

Los **métodos** son cuasi-funciones que, dado un objeto y unos argumentos, devuelven unos resultados. En el siguiente ejemplo se puede observar la regla que define el atributo `salarioIdeal` de los objetos `persona`.

```
X : persona[salarioIdeal@N-> Z] :- X[salario->Y] & Z = Y*N.
```

En F-Logic podemos tener las siguientes expresiones de métodos:

1. expresión escalar k-aria no heredable (k>=0)
   `scalarMethod@Q0, ..., Qk -> T`
2. expresión evaluada a conjunto j-aria no heredable (j, n>=0)
   `setMethod@R0, ..., Rj ->> { S0, ..Sn }`
3. expresión escalar k-aria heredable (k>=0)
   `scalarMethod@Q0, ..., Qk *-> T`
4. expresión evaluada a un conjunto j-aria heredable (j, n>=0)
   `setMethod@R0, ..., Rj *->> { S0, ..Sn }`
5. expresión de signatura[ii] escalar (n>=0)
   `scalarMethod@V0, ..., Vn => A`
6. expresión de signatura evaluada a un conjunto (s, t>=0)
   `setMethod@W0, ..., Ws =>> (B1, ..., Bt)`

dónde `Ri, Qi, A, B`, ... son oid.

Disponemos de dos tipos de relaciones de **herencia** entre objetos :

is_instance_of:	`pepe : estudiante`	(pepe instancia de estudiante)
is_a:	`estudiante :: persona`	(estudiante subclase de persona)

Las relaciones de herencia cumplen una serie de propiedades interesantes:

reflexiva: p :: p
transitiva: si p :: q, q :: r entonces p :: r
aciclicidad: si p :: q, q :: p entonces p = q
inclusión de subclases:
si p : q, q :: r entonces p : r

Una **fbf** en F-Logic es:

- hecho is-a P : Q, P :: Q
- F-molécula P[expresiónMetodo ; expresiónMetodo ; ...]
- ecuación P = Q

3.2 Características de la T-Logic

La T-Logic proporciona una forma declarativa de realizar manipulaciones sobre el estado de los objetos.

Se fundamenta entorno a tres ideas:

1. **conjunción serial** (es la forma de decir que después de una transacción[iii] va otra)
   `A :- B sc`[iv] `C.`

   para realizar A primero hay que realizar B y después, en el estado alcanzado tras ejecutar B, se ejecuta C.

2. **transacciones de estado elemental** (son operaciones elementales que nos permiten cambiar el estado de los objetos)

Hemos dicho con anterioridad que el estado en TF-Logic se representa a través de hechos y reglas. Las transiciones de estado elementales son un conjunto de operaciones que nos permiten manipular cualquier aspecto del estado. En concreto nos permiten:

- insertar/destruir objetos:
  ```
  Oid[].ins/del
  ```
- insertar/eliminar atributos de los objetos:
  ```
  Oid[ ... ; (atr => _).ins/del ;
             (atr -> _).ins/del ; ... ]
  ```
- insertar/eliminar relaciones de herencia entre objetos:
  ```
  ( A : B ).ins/del
  ( C :: D ).ins/del
  ```

3. **uniformidad de representación de las consultas y las actualizaciones** (no existe diferencia notacional entre realizar una consulta de un atributo y disparar un evento).

```
?- socio1[ nombre -> N ]        (consulta de un atributo)
?- socio1[ casarse ]            (disparo de un evento)
```

Ahora que hemos dado un repaso a los principales conceptos del formalismo que vamos a utilizar, ya podemos empezar a revisar las ideas básicas sobre la implementación de la metaclase.

4 Metaclase en TF-Logic

Los identificadores de los objetos que pueblan las sociedades son términos generados por un objeto especializado (fuente). Cada objeto tiene su identificador propio asignado por la fuente en el instante de creación del objeto. El oid nunca sufre modificaciones ni siquiera cuando el objeto es destruido, de forma que siempre se puede referenciar un objeto tanto si existe como si existió. Ejemplos de oid son:

```
fuente, ece, oid(234001), ...
```

La parte estática de un **objeto** se representa a través de una F-molécula que contiene el valor de los **atributos constantes** y **variables** que determinan el estado de dicho objeto. Por ejemplo, supongamos que las personas tienen como atributos constantes el dni, y el nombre y como atributo variable la edad, una persona concreta cuyo oid fuera `persona1` podría tener un estado como el siguiente:

```
persona1[dni->"20987427" ; nombre->"Francisco Perez" ; edad->26]
```

Los **atributos derivados** serán reglas que definen el cálculo necesario para obtener el resultado. Por ejemplo, podemos ver la definición del atributo derivado `viejo` en función de la `edad` de la persona.

```
P[ viejo-> V ] :- P : persona & P[ edad->E ] & V=(E>90).
```

Para representar las jerarquía de herencia existente en la metaclase y sus relaciones con las instacias que se vayan creando utilizaremos los hechos *is-a* e *is-instance-of*. En la figura 4 podemos ver la jerarquía de herencia existente entre la metaclase y el esquema conceptual de la gestión de una biblioteca en un instante dado.

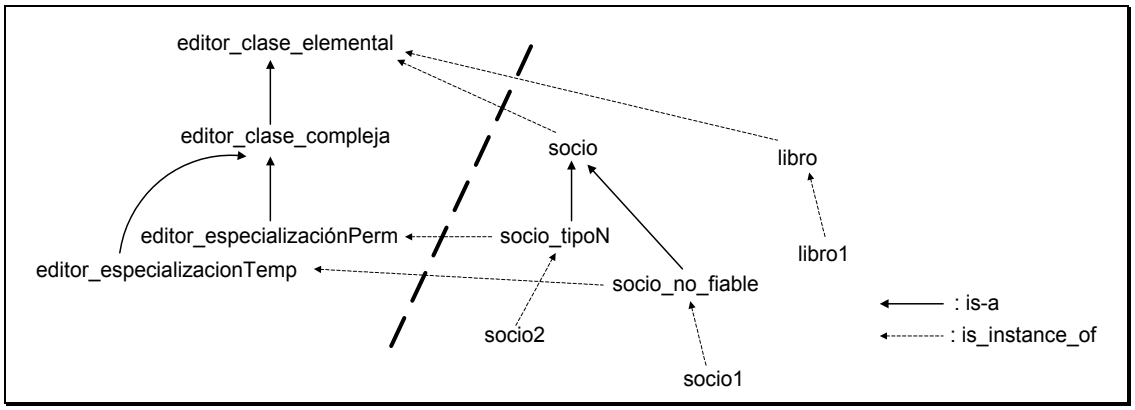


Figura 4: Estado del esquema conceptual de la gestión de una biblioteca en un instante dado.

La jerarquía de la figura 4 tiene una representación en F-Logic que se puede ver en la figura 5.

```
ecc^v :: ece^vi
eeP^vii :: ecc
eeT^viii :: ecc
socio : ece
libro : ece
socio_tipoN : eeP
socio_no_fiable : eeT

socio_tipoN :: socio
socio_no_fiable :: socio
libro1 : libro
socio2 : socio_tipoN
socio1 : socio_no_fiable
```

Figura 5: Hechos que representan una jerarquía de herencia.

Las **clases** tienen la doble visión que les aporta ser instancias de la metaclase: por un lado son objetos, con una serie de atributos consultables y manipulables a través de los eventos definidos en la clase de la cual son instancia (la metaclase), y por otro lado son clases de objetos que contienen la plantilla, la factoría y el almacén de los objetos que son instancia de ellas.

En la figura 6 podemos observar la instancia de la metaclase socio con sus atributos (nombre, new, lista_atr_ctes, ...), el almacén de los objetos instancia de ella (atributo población), así como la plantilla de sus objetos (num_socio, nombre, tipoN, num_libros, ...). Más adelante daremos la definición de la clase ece.

```
socio[ ........................
       nombre -> "socio" ;
       new -> alta_socio ;
       destroy -> baja_socio ;
       lista_atr_ctes  ->> { (num_socio, nat),
                             (nombre, string),
                             (tipoN, string} } ;
       lista_atr_vbles ->> { (num_libros, nat, 0),
                             (avisado, bool, false) } ;
       ............................



       ........................
       poblacion ->> { oid(5) } ;  * oid(5) es el objeto socio5

       num_socio => nat ;
       nombre => string ;
       tipoN => string ;
       num_libros => nat ;
       num_libros *-> 0 ;
       avisado => bool ;
       avisado *-> false ].
```

Figura 6: Parte del estado de la clase socio, instancia de ece.

Para definir el valor por defecto de los atributos variables utilizamos el mecanismo de los métodos heredables que nos proporciona la F-Logic[ix].

4.1 Fórmulas de consulta sobre el estado

En OASIS se utilizan fórmulas que se evalúan sobre el estado de los objetos para definir las precondiciones. También se utilizan fórmulas para definir las restricciones de integridad que deben de satisfacer en todo momento los objetos. Las fórmulas las representamos como objetos con un atributo que contendrá la propia fórmula a evaluar. Una fórmula es una expresión evaluable sobre el objeto. Las expresiones que son fórmulas se pueden deducir facilmente a través del código que las interpreta. Los objetos fórmulas tienen un método, llamado **interpretar**, que, como su nombre indica, los interpreta sobre los objetos instancia de la clase en la que se hayan definido las fórmulas.

Veamos cómo representar las restricciones de integridad de cualquier clase. La clase ece tiene un atributo, llamado `rie`, cuyo codominio es una fórmula:

```
ece[ ... ; rie => formula ; ... ]
```

Una clase concreta puede tener una restricción como la siguiente:

```
socio[ ... ; rie->formula(7)[exp->&(num_socio->N,&(N>1, N<10000)) ] ; ... ]
```

usando la notación prefija común a la hora de definir funciones, o bien, usando una notación infija equivalente:

```
socio[ ... ; rie->formula(7)[exp->(num_socio->N & N>1 & N<10000) ] ;
       ... ]
```

Por último sólo nos resta dar el método que interpreta las fórmulas:

```
F : formula[ interpretar@ Oid ] :-
       F[ exp -> Exp ] &
       formula[ solve@(Exp, Oid) ].
formula[ solve@( A & B, Oid ) ] :-
       formula[ solve@(A, Oid) ] &
       formula[ solve@(B, Oid) ].
formula[ solve@( A or B, Oid ) ] :-
       (formula[ solve@(A, Oid) ] or
        formula[ solve@(B, Oid) ]).
formula[ solve@( not(A), Oid ) ] :-
       not(formula[ solve@( A, Oid ) ]).
formula[ solve@(Query, Oid) ] :-
       Oid[ Query ] & !.
formula[ solve@(Calculo, _) ] :-
       Calculo.
```

que tendrá éxito si la fórmula se evalua a cierto, y que fallará en caso contrario.

4.2 Procesos

Para expresar las evaluaciones asociadas a la ocurrencia de los eventos en los objetos, utilizaremos una codificación similar a la usada para expresar las fórmulas que se evalúan sobre el estado.

Un proceso es un objeto que contiene un atributo con la secuencia de consultas, cálculos y *acciones elementales* a aplicar en el objeto sobre el que se haya provocado el evento asociado a dicho proceso. Las expresiones que son procesos se pueden deducir fácilmente a través del código que las ejecuta. Los objetos proceso tienen un método, llamado **ejecutar**, que los ejecuta sobre los objetos instancia de la clase en la que se hayan definido los eventos.

Veamos cómo representar las evaluaciones asociadas a los eventos de cualquier clase. La clase ece tiene un atributo, llamado `lista_eval`, cuyo codominio es proceso:

```
ece[ ... ; lista_eval =>> (ev(nomEv, largs), proceso) ; ... ]
```

Como ejemplo veamos el proceso asociado al evento prestar de la clase socio:

```
socio[ ... ; lista_eval ->> { (ev(prestar, nil),
                  proceso(9)[proc->sc( (num_libros->N).del,
                                       sc(M=N+1, (num_libros->M).ins ))]), ...
                    } ; ... ]
```

usando la notación prefija común a la hora de definir funciones, o bien, usando una notación infija equivalente:

```
socio[ ... ; lista_eval ->> { (ev(prestar, nil),
                  proceso(9)[proc->
(num_socio->N).del sc M=N+1 sc (num_socio->M).ins]),... } ; ... ]
```

El conjunto de transacciones en TF-Logic que ejecutan los procesos, y que es el encargado de hacer evolucionar a los objetos al provocar los cambios de estado asociados al disparo de los eventos, es el siguiente:

```
X : proceso[ ejecutar@ Oid ] :-
       X[ proc -> Proc ] sc
       proceso[ ejecutar@(Proc, Oid) ].
proceso[ ejecutar@( A sc B, Oid ) ] :-
       proceso[ ejecutar@(A, Oid) ] sc
       proceso[ ejecutar@(B, Oid) ].
proceso[ ejecutar@( AccionElemental, Oid ) ] :-
       Oid[ AccionElemental ].
proceso[ ejecutar@( Calculo, _ ) ] :-
       Calculo.
```

que tendrá éxito si es posible ejecutar el proceso, y fallará en caso contrario.

4.3 La Metaclase OASIS

Hemos dicho que las clases son instancias de la metaclase, por lo tanto, debemos definir qué es la metaclase. En la siguiente figura se puede ver la definición de la clase editor de clase elemental, parte fundamental de la metaclase, en F-Logic.

```
ece[   nombre => string ; nombre *-> "" ;
       new => nomEv ;
       destroy => nomEv ;
       lista_atr_ctes =>> (atributo, tipo) ; lista_atr_ctes*->>{} ;
       lista_atr_vbles =>> (atributo, tipo, valor)  ;
       lista_atr_vbles*->> {} ;
       lista_atr_deriv =>> (atributo, tipo) ; lista_atr_deriv*->> {} ;
       lista_ev_prop =>> ev(nomEv, largs^x) ;
       lista_ev_prop *->> {} ;
       lista_precond =>> (ev(nomEv, largs), formula) ;
       lista_precond *->> {} ;
       rie => formula ; rie *-> true ;
       lista_eval =>> (ev(nomEv, largs), proceso) ;
       lista_eval *->> {} ;
       poblacion =>> oid ;              poblacion *->> {} ;
       nextOid  => oid ;                * atributo derivado
       numObj => nat ;
       lista_atributos =>> atributo ;
       lista_eventos =>> nomEv ].
```

En el ece se puede observar la plantilla que seguirán las instancias de la metaclase. Contiene atributos para almacenar cada una de las características de la metaclase.

En el siguiente código se puede ver la regla que define el atributo derivado `nextOid`.

```
C[ nextOid -> oid(NID) ] :-
```

```
        C : editor_clase_elemental &
        fuente[nextOid-> NID].
```

Por supuesto, si decimos que todas las clases son instancias de la metaclase, la metaclase es instancia de sí misma.

```
editor_clase_elemental : editor_clase_elemental
```

Por lo tanto, tiene todos los atributos, eventos, restricciones, ... que puedan tener sus instancias.

```
ece[    nombre -> "editor_clase_elemental" ;
        poblacion ->> { } ;
        new -> crear_clase ;
        destroy -> destruir_clase ;
        lista_atr_ctes ->> { ... } ;
        lista_atr_vbles ->> { ... } ; ... ].
```

Y, por lo tanto, podemos hacerla evolucionar de la misma forma que hacemos evolucionar el resto de clases.

4.4 Animación de la metaclase

Hasta ahora hemos hecho referencia al conjunto de estructuras que van a dar soporte a las clases y objetos que pueblen nuestros sistemas sin embargo, no hemos hablado de cómo se va a animar la sociedad de objetos[xi].

Por un lado se encuentran todos los eventos definidos en la metaclase, que nos permitirán manipular las clases que se definan interaccionando con la metaclase. En la definición de estos eventos es donde se refleja la potencia de la metaclase.

Cuando se dispare un evento que modifique un atributo de una clase, habrá una **evolución** de la población afectada para adecuarse al nuevo estado del esquema.

Cuando se dispare un evento que implique un cambio en la condición de especialización de una clase, se producirá la **migración** de los objetos que pasen a / dejen de cumplir la condición de especialización.

Además, necesitamos un conjunto de transacciones que nos permitan disparar los eventos definidos para las clases: tanto los que provocan los cambio de estado fuerte (creación y destrucción de objetos), como los que provocan cambios de estado débiles.

El código en TF-Logic que resuelve el disparo de eventos que crean objetos es el siguiente:

```
C [ EvNew@ Largs -> Oid ] :-
        C : ece sc
        C[ new -> EvNew ] sc
        C[ lista_atr_ctes -> LAC ] sc
        C[ nextOid -> Oid ] sc
        Oid[].ins sc
        rellenaAtributos( Oid, LAC, Largs ) sc
        (Oid : C).ins sc     * objeto con atr. vbles valores defecto
        fuente[ incrementarOid ] sc
        C[ (poblacion->>Oid).ins ] sc
        rietriggers( C ) sc
        rietriggers( Oid ).

 * rellena un objeto con los valores de los atributos constantes
rellenaAtributos(_, nil, nil).
rellenaAtributos(Oid, (Atr, _).RestA, Valor.RestV) :-
        Oid[ (Atr->Valor).ins ] sc
        rellenaAtributos(Oid, RestA, RestV).
rellenaAtributos(_, _, _) :- error.
```

Dada una clase `C`, un evento `EvNew`, una lista (`Largs`) conteniendo el valor de los atributos constantes, crea un objeto y devuelve el `Oid` del objeto creado. Para ello comprueba que la clase es una instancia del ece (sólo las instancias de la metaclase pueden crear, a su vez, instancias). Después, se comprueba que el evento es realmente el evento de creación. Se calcula una lista con los nombres de los atributos que se hayan definido como constantes. Se obtiene el oid del objeto que se va a crear. Se añade un nuevo objeto en la base de objetos. Se 'rellenan' los atributos que se hayan definido como constantes con los valores pasados a través del argumento `Largs`. Se añade la información de que el objeto recién creado es una instancia de la clase en la que se generó el disparo del evento. Se actualiza el objeto fuente (generador de identificadores). Se mantiene el atributo población que es el encargado de almacenar la población de la clase. Y por último, se comprueba que no se han violado las restricciones de integridad de la clase ni las del objeto recién creado.

Después de ejecutar correctamente la transacción tenemos en el sistema un nuevo objeto, instancia de la clase en la que se disparó el evento de creación, con los valores adecuados en los atributos.

El código que resuelve el disparo de eventos que destruyen objetos es el siguiente:

```
C[ EvDestroy@ Oid ] :-
       C : ece sc
       C[ destroy -> EvDestroy ] sc
       C[ lista_precond ->> (ev(EveDestroy, nil), Precond) ] sc
       Precond[ interpretar@ Oid ] sc
       C[ (poblacion  ->> Oid).del ] sc
       (Oid : C).del sc
       Oid[].del .
```

Dada una clase `C`, un evento `EvDestroy`, y el `Oid` del objeto que se va a destruir, destruye dicho objeto. Para ello comprueba que la clase es una instancia del ece (sólo las instancias de la metaclase pueden destruir objetos). Luego se comprueba que el evento es realmente el evento de destrucción. Se obtienen y verifican las precondiciones de activación del evento de destrucción. Se elimina el objeto de la poblacion de la clase. Se elimina el hecho que relacionaba el objeto con la clase, y finalmente se elimina el objeto.

El código en TF-Logic que resuelve el disparo del resto de los eventos es:

```
Oid[ Eve@ Larg ] :-
       Oid : Clase sc
       Clase[ lista_precond ->> (ev(Eve, Largs), Precond) ] sc
       Precond[ interpretar@ Oid ] sc
       Clase[ lista_eval ->> (ev(Eve, Largs), Proceso) ] sc
       Proceso[ ejecutar@ Oid ] sc
       rietriggers( Oid ).
```

Dado un objeto `Oid`, un evento `Eve`, y una lista con los valores de los argumentos, se ejecuta el evento. Para ello se busca la clase a la que pertenece el objeto en el que se va a disparar el evento. Se comprueba que el objeto cumple la precondición. Se ejecuta el proceso asociado con el evento, y finalmente, se comprueba que se mantienen las restricciones de integridad y se disparan los triggers que se hayan podido activar.

5 Conclusiones y Trabajos Futuros

Partiendo de la TF-Logic y usando los mecanismos proporcionados por ella, hemos implementado parte de la metaclase OASIS ; metaclase que describe el modelo orientado a objetos de OASIS. Para ello, hemos descrito las estructuras que contienen el estado, tanto de la metaclase como de sus instancias, y hemos descrito la forma de animar los objetos que pueblen nuestros sistemas.

Este artículo es una primera aproximación, por lo que se ha dejado de lado aspectos tan importantes como son, por ejemplo, los eventos que manipulan las clases (y que incluyen la evolución), el tratamiento de los disparos, las transacciones, las clases complejas, ... que están siendo objeto de estudio en la actualidad.

Bibliografía

[BK95] Bonner A.J., Kifer M., *Transaction Logic Programming*, Technical Report CSRI-323, November 1995.

[Can96] Canós J.Hilario, *OASIS: Un lenguaje único para Bases de Datos Orientadas a Objetos*, Tesis Doctoral, Octubre 1996, DSIC-UPV (Valencia).

[Hung96] Hung S.Y.H., *Implementation and Performance of Transaction Logic in Prolog*, Master Thesis 1996, University of Toronto.

[Kif95] Kifer M. *Deductive and Object Data Languages : A Quest for Integration*, Dood 1996.

[KLW95] Kifer M., Lausen G., Wu J. *Logical Foundations of Object-Oriented and Frame-Based Languages*, Journal of the Association for Computing Machinery, May, 1995.

[Let93] Letelier P. ,Saavedra O., *Clases, Objetos y Comunicación en Programación Lógica Concurrente*, Technical Report, DSIC, Universitat Politècnica de València, 1993.

[PR96] Pastor, O. and Ramos, I., *OASIS version 2 (2.2): A Class-Definition Language to Model Information Systems using an Object-Oriented Approach*, SPUPV-95.788, Universitat Politècnica de València, 1996.

[RPP95a] Ramos I., Pelechano V., Penadés MC., Bonet B., Canós J.H., Pastor O., *Análisis y Diseño Orientado a Objetos de un Entorno de Prototipación Automática basado en OASIS*, Informe Técnico nº II-DSIC-9/95, Julio 1995, DSIC-UPV (Valencia).

[RPP95b] Ramos I., Pelechano V., Penadés MC., Bonet B., Canós J.H., Pastor O., *Especificación en R-OASIS de un Entorno de Prototipación Automática*, Informe Técnico nº II-DSIC-10/95, Julio 1995, DSIC-UPV (Valencia).

[San96] Sánchez P., Ramos I., *Object Petri Nets: Modelo de Implementación para OASIS*, Informe Técnico nº II-DSIC-17/96, Junio 1996, DSIC-UPV (Valencia).

[Tov94] Toval J.A., *Formalización de un Entorno de Producción Automática de Prototipos Orientados a Objetos*, Tesis Doctoral, Marzo 1994, DSIC-UPV (Valencia).

---

[i] Dado que un esquema se representa como el estado de una instancia de ella.

[ii] Nos permitirán definir los tipos de los argumentos usados en los métodos.

[iii] Una transacción en T-Logic es una acción o una secuencia de acciones.

[iv] ***serial conjunction***

[v] **ecc** es la abreviatura de editor_clase_compleja.

[vi] **ece** es la abreviatura de editor_clase_elemental.

[vii] **eeP** es la abreviatura de editor_especialización_Permanente.

[viii] **eeT** es la abreviatura de editor_especilización_Temporal.

[ix] Que también podríamos usar para definir atributos de clase, visibles por las instancias.

[x] largs identifica listas de variables. Ejemplo: A.B.C.Nombre.nil

[xi] Con excepción de las transacciones que ejecutan los procesos.

Análisis, Diseño e Implementación OO de un Entorno de Prototipación Automática

Penadés, M.C.; Ramos, I.; Canós, J.H.; Tolsá, J.; Ferrer, J
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Camino de Vera, s/n 46071 Valencia
Tel: 3877350 ; Fax: 3877357

Resumen

En este documento se presenta el análisis y diseño orientado a objetos de un entorno de prototipación automática y su posterior implementación. La descripción de este entorno introduce una visión totalmente homogénea y objetual de una herramienta CASE que nos permite producir y utilizar software orientado a objetos en un ambiente de programación automática, tomando el modelo OASIS como marco básico de referencia a través del cual se aborda con toda naturalidad el concepto de Metaclase. Se ha utilizado OO-Method como extensión metodológica, que cubre las fases de análisis y diseño dentro del Paradigma de Programación Automática y KAOS, un SGBDDAOO a través del cual se permite el prototipado rápido de las especificaciones realizadas en la fase de diseño, y al que se le ha incorporado el concepto de metaclase. Existe un primer prototipo operativo de esta herramienta llamado Meta-KAOS.

1. Introducción.

El Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia está desde hace varios años dedicado al estudio de los fundamentos teóricos y metodológicos de los Sistemas de Información (SI), con el fin de diseñar y desarrollar sistemas fiables, correctos y con un alto grado de reusabilidad, todo ello, siguiendo el Paradigma de Programación Automática ([Bal83]) y el Modelo Orientado a Objetos.

Siguiendo esta línea de investigación se ha desarrollado el lenguaje de especificación formal y orientado a objetos OASIS ([Pas95]), que ofrece un entorno deductivo y orientado a objetos en el que, usando OO-METHOD ([Pas96]) como extensión metodológica, se cubren hasta ahora las fases de análisis y diseño dentro del paradigma de programación automática. El objetivo inicial de OASIS fue el prototipado rápido de SI dentro del paradigma de programación automática; recientes trabajos, sin embargo, lo postulan como lenguaje único en una base de datos orientada a objetos: el sistema KAOS ([Can95]) es un prototipo de SGBDDAOO que implementa el modelo OO subyacente a OASIS sobre KBMS1 ([Man90]) una arquitectura de sistema de gestión de bases de conocimiento, ofreciendo una versión persistente del lenguaje OASIS, el cual no sólo se utiliza como lenguaje de definición de objetos, sino también como lenguaje de consulta y especificación de aplicaciones. La expresividad de la lógica empleada en la construcción de las especificaciones junto con el modelo OO empleado hacen de OASIS un lenguaje capaz de dar cuenta, entre otros, del modelo OQL planteado en el estándar ODMG-93 ([Cat94]), sin excesiva dificultad.

El siguiente paso ha sido abordar la generación de un Entorno de Prototipación Automática, una herramienta CASE que nos permita producir y utilizar software orientado a objetos en un ambiente de programación automática, partiendo de especificaciones realizadas

Penadés, M.C., Ramos, I., Canós, J.H., Tolsá, J., Ferrer, J. Análisis, diseño e implementación de un Entorno de Prototipación Automática. Actas de las I Jornadas de Trabajo en Ingeniería del Software,1996.

en el lenguaje de especificación formal OASIS.

El desarrollo de dicha herramienta se ha realizado totalmente dentro de un marco objetual, tomando el modelo de objetos subyacente al lenguaje OASIS como marco de referencia a través del cual se aborda, con toda naturalidad el concepto de Metaclase. Todo ello nos permite introducir una visión totalmente objetual de esta herramienta CASE soportada metodológicamente en la que *todo*, incluso los usuarios, son objetos que interactúan entre sí dentro de la sociedad, siguiendo una representación cercana a la visión que el usuario tiene de la herramienta.

En este artículo se esboza el análisis, diseño y posterior implementación de un Entorno de Prototipación Automática OO, presentando el primer prototipo operativo obtenido, Meta-KAOS. Para ello, en la sección 2 se presentan los conceptos básicos del modelo OO subyacente a OASIS. En la sección 3 se ofrece una descripción del Entorno de Prototipación Automática objeto de nuestro estudio. En el sección 4 se muestran los principales resultados obtenidos tras la fase de análisis y diseño del entorno, centrando nuestra atención en la noción de metaclase. En la sección 5 se presenta Meta-KAOS, que supone el primer prototipo en funcionamiento del entorno de prototipación propuesto. Finalmente, en la sección 6 damos paso a las conclusiones extraídas del presente trabajo así como los futuros puntos de investigación que han derivado de éste.

2. El Modelo Orientado a Objetos subyacente a OASIS.

Los bloques básicos de construcción en un Sistema de Información son los *objetos*. En el modelo OASIS, un **objeto** es un proceso observable ([Ram93]) que encapsula estado y comportamiento. Cada objeto tiene un identificador (**oid**)[1] que será único y lo distingue de cualquier otro objeto que haya existido, exista o pueda existir. Los objetos poseen propiedades, representadas en el modelo a través de la noción de *atributo*, distinguiéndose entre atributos constantes (si su valor no cambia durante toda la existencia del objeto), atributos variables (si su valor puede cambiar) y atributos derivados (si su valor se obtiene a partir del valor de otros atributos tras la aplicación de una serie de reglas de derivación).

El estado de un objeto viene dado por el conjunto de valores de sus atributos, de forma que un cambio en alguno de dichos valores representará un cambio de estado. Básicamente se distinguen entre cambios de estado fuertes (aquellos que suponen la creación/destrucción del objeto) y cambios de estado débiles (cambia el valor de los atributos del objeto sin que esto implique su destrucción). La causa de que se produzca un cambio de estado en un objeto se denomina *evento*. Un evento se define como la abstracción de un cambio de estado; es discreto, no tiene duración y ocurre en un cierto instante de tiempo. En el modelo OASIS, distinguimos entre eventos propios y eventos compartidos, según que participen en la vida de uno o más objetos.

La ocurrencia de un evento en la vida de un objeto sólo tendrá éxito si el objeto se encuentra en un estado que verifica la *precondición* asociada a dicho evento. Además no todos los estados de los objetos van a ser estados permitidos; en este sentido, nuestro modelo permite la definición de *restricciones de integridad* que definen precisamente qué estados serán aceptables en la vida de los objetos, en términos de valores permitidos para los atributos.

Los objetos no están aislados. La interacción entre objetos se modeliza en OASIS de dos formas: mediante compartición de eventos y relaciones de disparo. Las relaciones de disparo,

[1] En inglés, object identifier

representadas en el modelo OASIS por *triggers*, introducen actividad en la sociedad de objetos, permitiendo que un objeto actúe como agente de un evento de otro objeto cuando se satisfagan en él determinadas condiciones.

En el modelo OASIS, podemos definir *transacciones*, de manera que un conjunto de eventos funcionen como una unidad de ejecución de mayor granularidad. Siguen los dos principios básicos de toda transacción, a saber: todo o nada y no observabilidad de estados intermedios. Además también podemos definir *procesos* usando los eventos y transacciones como acciones atómicas, permitiéndonos especificar las posibles vidas correctas de los objetos.

Los objetos no están aislados, sino que tendemos a agrupar aquellos objetos que tienen las mismas propiedades en *clases*. Un objeto individual es una *instancia* o ejemplar de una clase. En este sentido, la sociedad de objetos de nuestro modelo OASIS está compuesta de las siguientes clases de objetos:

Las **clases primitivas** o dominios están constituidas por "objetos" que tienen un único estado y que siempre existen. En general, abarcan el dominio de los tipos abstractos de datos (TAD). La sociedad de objetos se construirá tomándolos como nivel estructural básico sobre el que se declaran las demás clases de objetos. En toda especificación OASIS habrá varios dominios predefinidos: los números naturales, el tipo *booleano* y las cadenas de caracteres. Además, pueden definirse explícitamente otros, a través de sus correspondientes TAD.

A partir de los dominios se construyen las **clases elementales**. Estas vienen caracterizadas por el tipo que una colección de objetos comparte. En el tipo se definen los atributos, los eventos, las restricciones de integridad, las precondiciones, las relaciones de disparo, las transacciones y la parte proceso que caracterizan a los objetos de la clase.

Mediante un mecanismo constructivo, se definen las **clases complejas** a partir de las clases elementales o de otras clases complejas definidas de antemano, aplicando sobre ellas los operadores de clases. Los operadores más utilizados son la especialización y la agregación ; aunque existen otros como la generalización (operador inverso a la especialización), la asociación (un tipo particular de agregación que abarca la noción de colección) y la composición paralela (un operador que viene de la Teoría de Procesos y nos permite definir un sistema organizacional como el resultante de la reunión concurrente de las clases definidas con anterioridad). Todos los operadores de clases mencionados anteriormente son *ortogonales*, es decir, son independientes unos de otros y es posible cualquier combinación entre ellos.

El modelo de objetos subyacente a OASIS también soporta la noción de **vista**. Una vista se define como una clase virtual que permite observar objetos imaginarios, aunque tengan un soporte real (una o más instancias). La definición de vistas establece un modelo de protección que permite al diseñador especificar qué visión tienen las instancias de una clase sobre los objetos de las demás clases del sistema. OASIS introduce cuatro tipo de vistas o interfaces: de proyección, de restricción de la población, personalizada y derivada ([Rai92]).

OASIS es un lenguaje formal de especificación para sistemas abiertos y activos. Está basado en la lógica y permite la definición de esquemas conceptuales según el modelo objetual visto. Existen diferentes versiones del lenguaje dependiendo de la expresividad lógica que se utilice. Las más destacadas son R-OASIS (expresividad clausal), F-OASIS (expresividad funcional) y L-OASIS (expresividad lógico-ecuacional), todas ellas definidas en

[Pas92]. En este artículo, utilizaremos la versión de Oasis basada en la lógica clausal dinámica (R-OASIS), pudiéndose encontrar una definición más completa del mismo en [Can96].

3. Descripción del Entorno de Prototipación Automática

Desde un punto de vista funcional, podemos describir este Entorno como una herramienta CASE que permite producir y utilizar software orientado a objetos en un ambiente de programación automática, a partir de especificaciones realizadas en OASIS y dando soporte a OO-METHOD como extensión metodológica. Todo esto, a través de entornos de trabajo gráficos que permitan especificar y ejecutar nuestros SI.

Dicho entorno debe ser visto desde una doble perspectiva. Por un lado, ha de ser un entorno de producción de software orientado a objetos y, por tanto, debe soportar, entre otras, facilidades para la creación de nuevas clases. Por otro, ha de estar basado en el paradigma de programación automática, lo cual significa que debe permitir el prototipado rápido de las instancias de las clases que se creen en dicho entorno.

Esta dualidad, de difícil solución en los modelos OO más tradicionales (en los que a menudo se cae sin remedio en la oscura circularidad de las **metaclases** entendidas como clases de clases de clases...), es desde el modelo OASIS tratada con toda naturalidad, como se verá en el siguiente apartado. La descripción de este entorno introduce una visión totalmente objetual y homogénea de una herramienta CASE soportada metodológicamente en el que *todo*, incluso los usuarios, son objetos.

4. Análisis y Diseño OO de un Entorno de Prototipación Automática.

En este apartado presentamos brevemente los principales resultados del análisis y diseño OO del Entorno de Prototipación Automática, cuyas características más relevantes acabamos de comentar.

En OO-METHOD la fase de análisis comienza con un proceso de clasificación para identificar las clases existentes en nuestro sistema y sus relaciones. De acuerdo con la descripción del entorno, identificamos la necesidad de distinguir un nuevo nivel de abstracción para poder dar cuenta de esa dualidad, lo que nos conduce a la introducción del concepto de **metaclase.** La *metaclase* es una clase con unas características que la diferencian de las demás, ya que sus instancias no sólo son objetos, sino que a su vez son nuevas clases que poblarán nuestros sistemas con sus instancias (los objetos).

La *metaclase* nos servirá para poder definir la estructura y el comportamiento de los objetos clase, de la misma forma que utilizamos las clases para definir la estructura y comportamiento de los objetos instancia. Por lo tanto, la metaclase debe proporcionar al usuario los servicios necesarios para poder construir nuevas clases, y en este sentido podríamos decir que la *metaclase* es un *editor abstracto de clases oasis*. Desde esta perspectiva, la metaclase es una representación abstracta del modelo OO subyacente al lenguaje OASIS.

Las instancias de la *metaclase* pueden ser vistas desde una doble perspectiva: **traza** y **estado.** Si analizamos su representación gráfica (Figura 1) vemos que hay dos partes: la parte derecha representa el nivel *meta* (traza) mientras que la parte izquierda representa el nivel *clase* (estado).

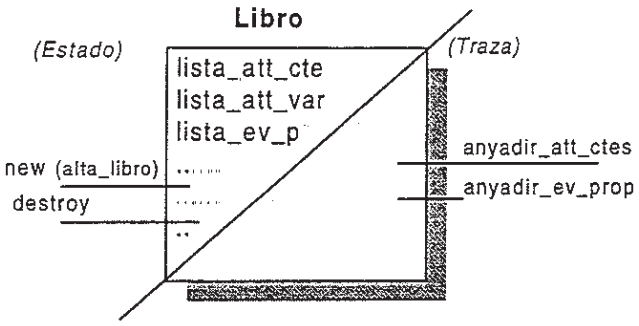


Figura 1. Representación gráfica de la instancia *libro* de la metaclase.

Las instancias de la *metaclase* vistas como traza, serán la secuencia de eventos que un *usuario en edición* habrá activado en la *metaclase*; la activación de estos eventos permitirán al usuario la construcción y refinamiento incremental de las distintas clases que forman los sistemas de información. Cada instancias de la *metaclase* vistas como estado es la representación abstracta de una clase, que coincidirá con una clase determinada en un instante determinado. El paso de traza a estado viene dado por las evaluaciones de los eventos de la *metaclase*, este cálculo del estado de la *metaclase* representa el paso intermedio entre edición y ejecución, es decir entre el nivel meta y el nivel clase. La semántica de la evaluación asociada al evento **new** que aparece en la parte correspondiente al estado de la instancia de la *metaclase,* nos marca el paso del nivel meta al nivel clase. Este paso es dependiente de la implementación elegida para representar a los objetos[1].

Vamos a ilustrar esta doble perspectiva de la instancias de la *metaclase*, tomando como ejemplo la clase *libro* del esquema conceptual que modeliza el funcionamiento de una biblioteca. La traza que aparece en la figura 2 es una posible vida de la clase *libro* vista como instancia de la *metaclase*, es decir, la sucesión de eventos de la *metaclase* que le han acontecido. Dichos eventos así como las evaluaciones asociadas a los mismos aparecen en la signatura del *editor de clases OASIS* (*metaclase*) cuya especificación la podemos encontrar en [Ram95-2] y [Bon95].

```
Traza_de_la_clase_libro =
crear_clase & anyadir_nom_clase("libro") & anyadir_atr_cte("cod_libro","nat") &
anyadir_atr_cte("titulo","string") & anyadir_atr_cte("tema","string") & anyadir_identificador("cod_libro") &
anyadir_atr_vble(atr("disponible","bool")) & anyadir_ev_new("alta_libro") &
anyadir_ev_destroy("baja_libro") & anyadir_ev_comp("prestar",vaciaL,almaL(vaciaL,"socio")) &
anyadir_ev_comp("devolver",vaciaL,almaL(vaciaL,"socio")) &
anyadir_evaluacion(true,"alta_libro",disponible(true)) &
anyadir_evaluacion(true,"prestar",disponible(false)) & anyadir_evaluacion(true,"devolver",disponible(true))
& anyadir_precond("baja_libro",disponible(true))&anyadir_precond("prestar",disponible(true)) &
anyadir_precond("devolver",(disponible(false))&anyadir_RI_estática(cod_libro(N)&N>1&N<1000)
```

Figura 2. Traza de la clase libro

Las evaluaciones asociadas a los eventos de la metaclase nos dan el cálculo del estado a partir de la traza. La figura 3 muestra el estado de la clase libro, vista como instancia de la metaclase. Este estado no es más que la representación abstracta de la clase libro, cuya especificación en el lenguaje R-OASIS, aparece en la figura 4.

[1] En este caso, la implentación elegida ha sido KAOS.

```
nombreClase("libro").
lista_atr_ctes(almaL(almaL(almaL(vaciaL,atr("cod_libro","nat")), atr("titulo","string")),atr("tema","string")).
lista_ident(almaL(vaciaL,"cod_libro")).
lista_atr_vbles(almaL(vaciaL,atr("disponible","bool"))).
lista_atr_deriv(vaciaL).
lista_deriv(vaciaL).
new("alta_libro").
destroy("baja_libro").
lista_ev_prop(vaciaL).
lista_ev_comp(almaL(almaL(vaciaL,ev_comp(ev("prestar",vaciaL),almaL(vaciaL,"socio"))),
                                  ev_comp(ev("devolver",vaciaL),almaL(vaciaL,"socio"))).
lista_precond(almaL(almaL(almaL(vaciaL, precond(baja_libro,disponible(true))),
                                        precond(prestar,disponible(true))),
                                        precond(devolver(disponible(false))).
lista_triggers(vaciaL).
RI_estática(cod_libro(N) & N>1 & N<1000).
lista_eval(almaL(almaL(almaL(vaciaL, eval(alta_libro,true,disponible(true))),
                                     eval(prestar,true,disponible(false))),
                                     eval(devolver,true,disponible(true))).
actualID(0).
poblacion(vaciaL).
```

Figura 3. Estado de la clase libro.

```
class libro
    constant_attributes
        key     cod_libro(nat);
                titulo(string);
                tema(string)
    constraints     N:nat
        static
        cod_libro(N) & N>1 & N<1000.
    variable_attributes
    disponible(bool)
      valuation
        [alta_libro]        disponible(true).
        [prestar] disponible(false)
        [devolver]          disponible(true).
      end_valuation
    private_events
        new     alta_libro;
        destroy baja_libro.
    shared_events
        prestar with socio;
        devolver with socio.
    preconditions
        baja_libro <- disponible(true).
        prestar   <- disponible(true)
        devolver  <- disponible(false).
    processes
        libro  <- alta_libro & prestar & libro0
        libro0 <- (prestar or devolver) &
                  libro0 or libro1.
        libro1 <- baja_libro.
end_class
```

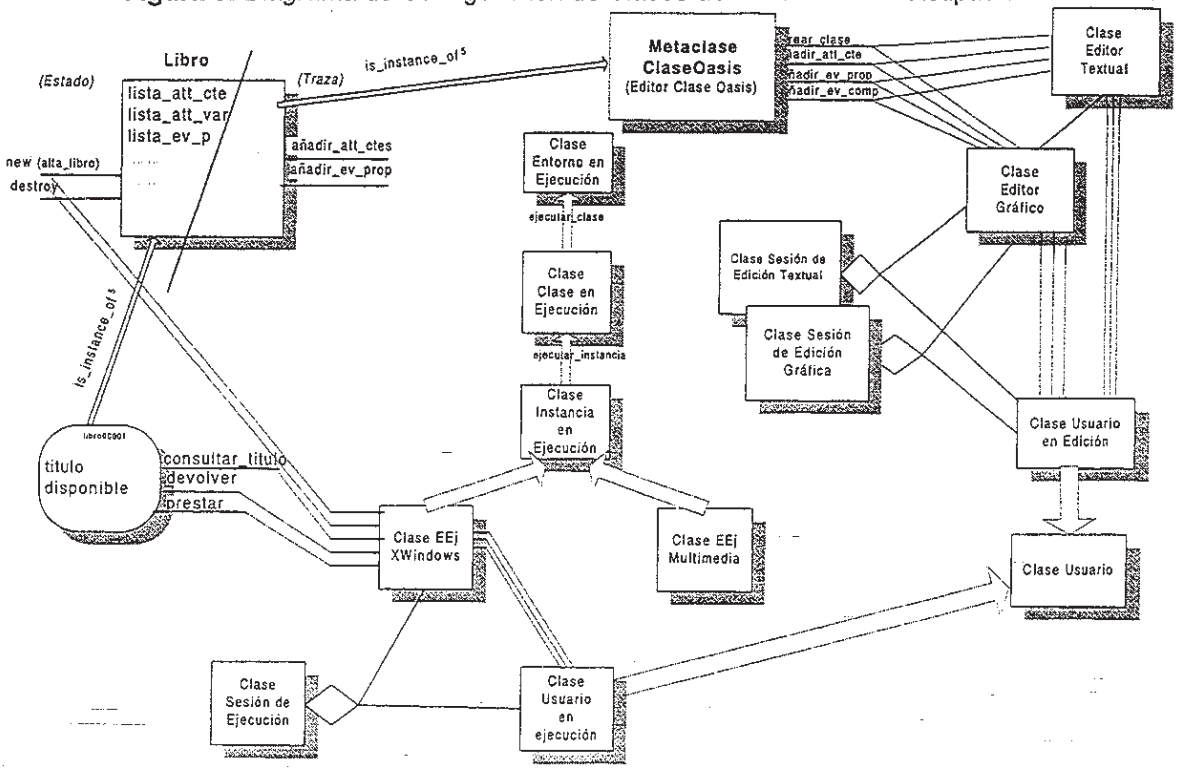
Figura 4. Especificación OASIS de la clase libro

El elemento principal del Entorno de Prototipación es la metaclase, puesto que ésta es la que nos permite dar cuenta de esa doble perspectiva que acabamos de comentar (nivel meta y nivel clase). Lógicamente, un Entorno de Prototipación Automático más completo, constaría de otros elementos. En este sentido la figura 5 nos da una visión global de la estructura de dicho entorno. Además de la metaclase, se ha detectado en el sistema una *clase usuario*, la cual, a través del evento *login* se especializa dinámicamente en *usuario en edición* o en *usuario en ejecución*, según quiera editar clases oasis o ejecutarlas. El Entorno debe permitir que un *usuario en edición* pase a ser en cualquier momento *usuario en ejecución* y viceversa. Un *usuario en edición* crea y modifica las instancias de la *metaclase*, vistas como objetos, mientras que el *usuario en ejecución* trabajará con las instancias de la *metaclase* vistas como clases y con sus instancias (los objetos que poblarán nuestros SI).

El *usuario en edición* accede a los servicios de la *metaclase* a través de una interfaz, ya sea gráfica o textual, representada por la *clase editor gráfico* y *clase editor textual*. Del mismo modo, la *clase entorno de ejecución* representa la interfaz abstracta con la que se comunica el *usuario en ejecución* para solicitar los servicios a las clases y a sus instancias.

Una *sesión de edición gráfica/textual* no es más que la instancia de objeto relación que se establece entre el *usuario* y la *clase editor gráfico* o la *clase editor textual*. De igual forma, la *sesión de ejecución* modeliza la instancia de objeto relación que surge entre el *entorno de ejecución* concreto y el *usuario*, durante la ejecución de un esquema conceptual.

Figura 5. Diagrama de Configuración de Clases del Entorno de Prototipación Automática



La metaclase es el elemento principal del Entorno de Prototipación Automático propuesto. Por ello, en la figura 6, presentamos cuál es estructura de la metaclase a un nivel de abstracción menor. La *metaclase Clase_OASIS* es un editor abstracto de clases OASIS y como hemos comentado en el apartado 2, OASIS proporciona tres unidades de diseño que son: clases primitivas, clases elementales y clases complejas. Por lo tanto, la *metaclase* deberá permitir al usuario poder construir cualquiera de estas clases.

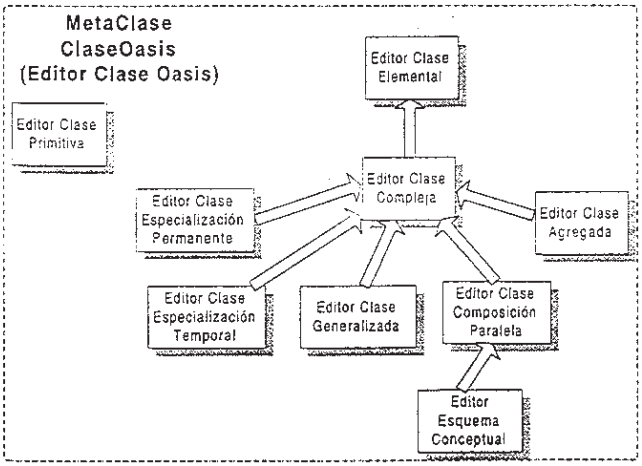


Figura 6. La metaclase

Un aspecto fundamental de toda clase compleja es que va a ser una clase más del sistema y seguirá el mismo patrón dado para una clase elemental, pero con información adicional dependiente del operador utilizado como generador de la clase compleja (agregación, especialización, ...). Por esta razón, el *editor de clases complejas* aparece como una especialización del *editor de clases elementales*.

El esquema conceptual puede verse como la composición paralela de las distintas clases (primitivas, elementales y complejas) que componen el sistema a modelizar, por lo tanto, el *editor de esquemas conceptuales* aparece como una especialización del *editor de clase composición paralela*.

El *editor de clases Oasis* puede ser visto como una composición paralela del editor de clases primitivas y el editor de clases no primitivas.

Una vez completado el análisis del Entorno de Prototipación Automática, del cual sólo hemos comentado lo más relevante, el siguiente paso es la obtención de la especificación OASIS del sistema que estamos modelando, que por motivos obvios de espacio no se incluye en el presente artículo. Para una mayor información tanto de los aspectos estructurales como los correspondientes a toda la parte de la dinámica (de la cual no hemos comentado nada), así como para consultar la especificación OASIS obtenida, remitimos al lector a [Ram95-1], [Ram95-2] y [Bon95].

5. Meta-KAOS.

Una vez realizado el análisis y diseño del Entorno de Prototipación Automática, centrando principalmente en la metaclase, el siguiente paso lógico es la obtención de una implementación real. Con este propósito nace Meta-KAOS ([Tol96],[Fer96]), el primer prototipo ejecutable que implementa el concepto de metaclase.

El desarrollo de Meta-KAOS se ha basado principalmente en KAOS ([Can96]) un Sistema Gestor de Bases de Datos Deductivas, Activas y Orientadas a Objetos, que utiliza OASIS como lenguaje único para el diseño, consulta, actualización y programación de aplicaciones. KAOS implementa el modelo orientado a objetos subyacente a OASIS, pero al abordar el análisis y diseño del Entorno de Prototipación Automática e incorporar el concepto de metaclase, ha sido necesario enriquecer KAOS con todas las nuevas características y funcionalidades que supone la *metaclase*, especialmente el hecho de poder proporcionar esa doble perspectiva de las instancias de la metaclase (traza y estado) comentadas anteriormente.

Meta-KAOS es una herramienta CASE que incorpora una visión objetual en todas las fases de prototipación automática. Permite no sólo especificar los sistemas de información, sino también ejecutarlos. Por ello, Meta-KAOS se debe ver desde una doble perspectiva: por una parte, permite la construcción de nuevas clases que modelizan los SI y por otra parte, permite el prototipado rápido de las intancias de las clases. Es decir, el usuario trabajando con Meta-KAOS puede definir sus propias especificaciones que modelan los SI en los que se está trabajando y posteriormente animar y validar los prototipos creados automáticamente. Todo ello gracias a la noción de metaclase.

Una característica fundamental de Meta-KAOS es la uniformidad que proporciona, pues la forma de trabajar del usuario es la misma independientemente de que esté creando nuevas clases o prototipando las existentes.

Para finalizar comentar que Meta-KAOS es la primera implementación del Entorno de Prototipación Automática presentado anteriormente, en el sentido de que implementa el principal componente, la metaclase. Actualmente está finalizada la versión textual de la herramienta lo que nos permite probar su funcionalidad. El siguiente paso a realizar es el dotarlo de una capa gráfica que muestre al usuario de una forma más ergonómica, todas estas características de homogeneidad e integración en el modelo OO de la herramienta. Actualmente se está trabajando en este tema y aunque no existe una implementación del

mismo, sí que se ha elaborado una propuesta de como debe ser ese entorno (Ver [Bon96]). En este sentido, Meta-KAOS ofrece al usuario una visión del sistema modelado como una sociedad de objetos que interactúan entre ellos, e incluso el propio usuario se sumergirá en esta sociedad de objetos como un objeto más. A partir de es momento podrá crear, evolucionar y validar especificaciones que modelen SI. Las figuras 7 y 8 muestran un primer esbozo de la interfaz gráfica propuesta.

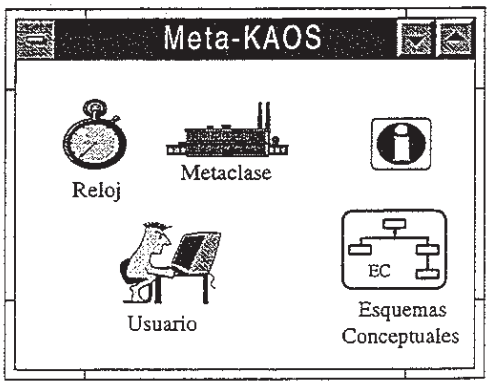


Figura 7. Pantalla inicial de Meta-KAOS.

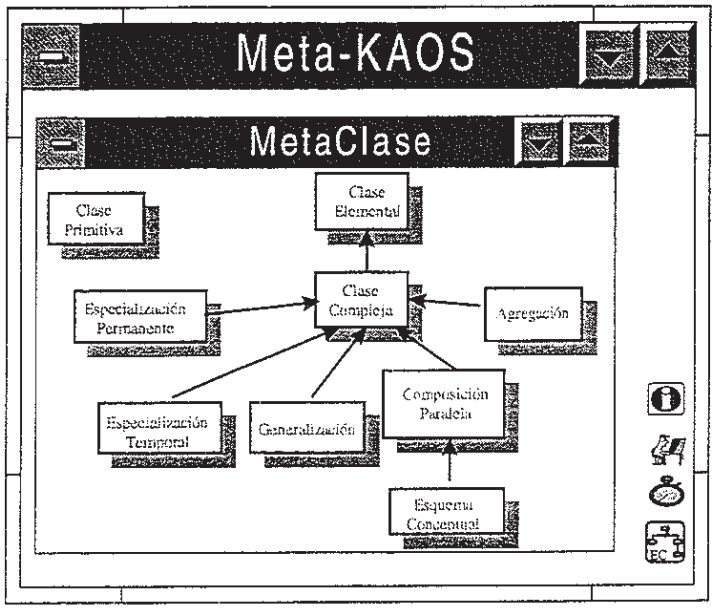


Figura 8. Inicio de una sesión de trabajo con la Metaclase.

6. Conclusiones y Trabajos Futuros.

En este artículo se presentan esquemáticamente algunos de los principales resultados obtenidos al abordar la tarea de especificar el modelo de objetos de OASIS usando al propio OASIS como herramienta. Hay una larga enumeración de trabajos que sustentan todo lo presentado en este artículo, de los cuales los más importantes se han mencionado a lo largo del mismo.

La descripción que se ha realizado del Entorno de Prototipación Automática ha introducido una visión totalmente objetual de un entorno CASE homogéneo, soportado metodológicamente en el que todo, incluso los usuarios, son objetos que forman parte de una sociedad. En el Modelo de Objetos que se ha tomado como marco de referencia se trabaja con objetos de muy alto nivel que no dependen del lenguaje de especificación utilizado. En concreto, este modelo nos ha permitido abordar, con toda naturalidad, el concepto de **metaclase**.

La *metaclase*, en nuestra aproximación es un *Editor Abstracto de Clases* que nos sirve para construir abstractamente nuestro modelo OASIS. La Metaclase introduce un nuevo nivel en el modelo puesto que ofrece al usuario todos los servicios necesarios para poder construir nuevas clases que, como instancias de la metaclase pueden ser vistas también como objetos que poseen atributos y eventos propios. Llegados a este punto, no sólo se ha realizado el análisis y el diseño, sino que existe un primer prototipo en funcionamiento que recoge las principales características de la noción de *metaclase*.

Somos conscientes de que esta línea de trabajo iniciada ha abierto otros muchos temas relacionados. Algunos de estos temas en los que se está trabajando y pronto se esperan tener resultados son: incorporar una interfaz gráfica al Entorno de Prototipación Automática, algunos de cuyos aspectos se han adelantado en el apartado anterior; enriquecer la representación de la metaclase para que puede captar también la semántica del modelo, así como la definición de servicios más complejos; abordar el tema de evolución de esquema, desde la perspectiva de la metaclase y las facilidades que esta nos proporciona; definir cual sería la metaclase mínima a partir de la cual podríamos definir el propio modelo objetual en el que vamos a trabajar, lo cual nos permitiría evolucionar el propio modelo, e incluso la evolución de las propias aplicaciones, etc.

Bibliografía.

[Bal83] Balzer, R.; Cheatman, T.E.; Green, C., *"Software Technology in the 1990's: Using a New Paradigm"*, IEEE Computer, Nov. 1983, pp. 39-45

[Bon95] Bonet, B.; *"Análisis y Diseño Orientado a Objetos de un Entorno de Prototipación Automática basado en Oasis"*, PFC, Facultad de Informática, UPV, Diciembre 1995.

[Bon96] Bonet, B.; Pelechano, V.; Penadés, M.C.; Ramos, I.; Pastor, O.; Carsí, J.A.; *Descripción de la Interfaz Gráfica correspondiente a un Entorno de Prototipación Automática basado en OASIS*, Tech. Report, DSIC, Universidad Politécnica de Valencia, 1996

[Can95] Canós, J.H; Penadés, M.C.; Ramos, I.; Pastor, O.; *A Knowledge-Base Architecture for Objects Societies*, Proceedings of DEXA-95, London, 1995

[Can96] Canós, J.H; OASIS: *Un lenguaje único para BDOO*, Tesis Doctoral dirigida por I.Ramos, 1996, DSIC-UPV (Valencia).

[Cat94] Catell,R.G.G. (ed.): *The Object Database Standard ODMG-93, Release 1.1*, Morgan Kaufmann Publishers. 1994.

[Fer96] Ferrer, J.; *Incorporación del concepto de Metaclase en un SGBDDAOO: Implementación del nivel Clase*, PFC, Facultad de Informática, UPV, 1996. (En preparación)

[Man90] Manley J.,Cox A.,Harrison K.,Syrett M.,Wells D.; *KBMS1. A User Manual*, Information System Centre, HP-Labs. 1990. Bristol UK.

[Pas92] Pastor, O.; *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos*. Tesis Doctoral dirigida por Isidro Ramos, 1992, DSIC-UPV (Valencia)

[Pas95] Pastor,O.: Ramos, I.: *Oasis 2.1.1 A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, Febrero 94 (1ª ed.) Marzo 95 (2ª ed.). Octubre 95 (3ª de.)

[Pas96] Pastor,O.; Pelechano,V.; Bonet,B.: *OO-Method una Metodologia de Análisis y Diseño Orientado a Objetos*. Tech. Report, DSIC, Universidad Politécnica de Valencia, 1996.

[Rai92] Raimundo, F.; *Interfaces y Protección en un Sistema de Especificación Orientado a Objetos (OASIS)*, PFC, Facultad de Informática, UPV, Octubre 1992.

[Ram93] Ramos,I.:Pastor,O.:Cuevas,J.:Devesa,J.; *Objects as Observable Processes*, Proceedings of the 3rd Workshop on the Deductive Approach to Information System Design, Roses, Costa Brava, 1993

[Ram95-1] Ramos, I.; Pelechano, V.; Penadés, M.C.; Bonet, B.; Canós, J.H.; Pastor, O.; *Análisis y Diseño Orientado a Objetos de un Entorno de Prototipación Automática basado en OASIS,* Tech. Report, DSIC, Universidad Politécnica de Valencia, 1995.

[Ram95-2] Ramos, I.; Pelechano, V; Penadés, M.C; Bonet, B; Canós, J.H.; Pastor, O; *Especificación en R-OASIS de un Entorno de Prototipación Automática,* Tech. Report, DSIC, Universidad Politécnica de Valencia, 1995.

[Tol96] Tolsá, J.; *Incorporación del concepto de Metaclase en un SGBDDAOO: Implementación del nivel Meta,* PFC, Facultad de Informática, UPV, 1996. (En preparación)

Computación y Sistemas. Vol. 1, No. 2, Octubre - Diciembre, 1997, pp. 101 - 111

REPRESENTACIÓN DE OBJETOS OASIS EN TEORÍAS Y SU TRADUCCIÓN A UN MODELO DE CONCURRENCIA

Pedro Sánchez Palma, Patricio Letelier Torres e Isidro Ramos Salavert
Departamento de Sistemas de Información y Computación
Universidad Politécnica de Valencia
e-mail :{ppalma | letelier | iramos} @dsic.upv.es

RESUMEN

La validación temprana de requisitos del software durante el desarrollo de un sistema de información es un aspecto de reconocida importancia. La animación automática a partir de la especificación de requisitos del sistema puede llegar a ser una valiosa ayuda en dicha tarea. OASIS es un lenguaje formal para especificación de modelos conceptuales. En este trabajo se presenta una formalización del proceso de obtención de un prototipo dada una especificación OASIS, un modelo básico de ejecución y un Modelo de Concurrencia como son las Redes de Petri clásicas. Si definimos las características formales que debe cumplir un prototipo para que satisfaga una especificación OASIS habremos conseguido una representación ejecutable fiel al modelo original. El planteamiento que se hace del trabajo es un comienzo que permite ser extendido al estudio de otros tipos de redes (Object Petri Nets, Coloured Petri Nets, etc.) u otros modelos de concurrencia.

1. INTRODUCCIÓN

OASIS [13][14] es un lenguaje formal para la especificación conceptual de sistemas de información. Actualmente existe un creciente interés en incorporar total o parcialmente técnicas formales para modelar sistemas de información [16][3]. Otros lenguajes de especificación formales con enfoques similares al ofrecido por OASIS son ALBERT [1][2], CMSL [22] y TROLL [4]. La propuesta metodológica basada en OASIS es OO-Method [16]. Se ha desarrollado una herramienta CASE basada en OO-Method [15][17].

Una funcionalidad atractiva es la posibilidad de animar la especificación del sistema mediante un prototipo generado automáticamente. En esta línea, hemos experimentado usando Redes de Petri [19] y lenguajes de Programación Lógica Concurrente [11][12] como dominio semántico para especificaciones OASIS. El extender los conceptos expuestos a otros modelos de concurrencia puede hacerse redefiniento el concepto de estado y adecuándolo al planteamiento escogido. El objetivo de este trabajo es presentar una justificación formal del proceso de traducción y de la obtención del prototipo de especificaciones OASIS en un entorno de programación concurrente. En nuestro caso, la problemática de verificar la corrección de la animación respecto del modelo conceptual se ve apoyada por dos factores importantes: el modelo conceptual se especifica en un lenguaje formal y el modelo de ejecución está inspirado en la semántica operacional de dicho lenguaje formal [10]. Esto, por sí mismo determina la corrección exigida y provee un camino de formalización, objetivo de este trabajo.

El trabajo se estructura comenzando con una presentación formal de los conceptos objetuales, tales como, clase, objeto, estado, mundo, etc. Posteriormente, estos conceptos se relacionan con la semántica asociada a una especificación OASIS determinada en el Modelo de Ejecución descrito en [9][10]. En último lugar, se formalizan las condiciones necesarias para que la implementación de un prototipo utilizando Redes de Petri sea consistente con el modelo planteado.

2. DEFINICIÓN DE CLASE Y OBJETO

La mayoría de los conceptos que se exponen a continuación ya han sido definidos en otros trabajos como [21][23] pero los consideramos necesarios para sentar las bases de las definiciones posteriores.

Sánchez, P. Letelier, P., Ramos, I. Representación de Objetos OASIS como Teorias y Traducción a un Modelo de Concurrencia. Computación y Sistemas, Vol. I, Nº. 2. 1997.

Definición 1.1 Un **Esquema de Clase EC** es una cuatro-tupla (A, E, C, P), siendo **A** el alfabeto de Atributos, **E** el alfabeto de Eventos, **C** un conjunto de axiomas (fórmulas en lógica dinámica) y **P** un conjunto de ecuaciones en un Álgebra de Procesos.

Sea *SORT* un conjunto de nombres de géneros (*sorts*). Para cada atributo $a \in A$ de un esquema de clase *EC*, existe un sort $S_{a \in A} \in SORT$. Denotamos por $|S_a|$ al conjunto soporte[1] del sort S_a.

Definición 1.2 Una **Clase** en Oasis puede representarse como la tripleta (name, EC, ID), siendo **name** el nombre de la clase, **ID** los mecanismos de identificación de sus instancias (objetos) y **EC** un esquema de clase o plantilla de los aspectos estáticos y dinámicos que comparten todas las instancias.

Definición 1.3 Un **Objeto** es un proceso observable [18] cuya evolución viene caracterizada por la ocurrencia de eventos y cuyos aspectos estáticos y dinámicos están determinados por el EC de la clase a la que pertenece.

Los objetos pueden verse desde una doble perspectiva [14]: *estática* y *dinámica*. Desde un punto de vista estático poseen un estado representado (entre otros) por valuaciones de sus atributos en el sort apropiado; por otro lado (perspectiva dinámica), la ocurrencia de eventos causa una transición entre estados. El lenguaje de la lógica dinámica utiliza dos lenguajes: un conjunto *F* de fórmulas de primer orden como abstracción de las propiedades que caracterizan el estado, y un conjunto *P* de procesos como abstracción de cambios de estado.

3. SEMÁNTICA DECLARATIVA

La semántica declarativa de la lógica dinámica viene dada [21] mediante estructuras de Kripke $S = (W, \pi, m)$ tales que *W* es un conjunto cuyos elementos se llaman mundos posibles; π es una función que asigna a cada fórmula (perteneciente a *F*) un subconjunto de *W* en los que es cierta (π: $F \rightarrow 2^W$), y **m** una función con dominio del sort de procesos P y con codominio un conjunto de relaciones binarias sobre *W* (asignando a cada elemento de P una relación de accesibilidad entre mundos, es decir,

$$m: \qquad P \rightarrow (W \rightarrow W)).$$

Definición 2.1 Una **Estructura S** es una dos-tupla (W, π) con W un conjunto y π: $F \rightarrow 2^W$ una función.

Definición 2.2 Para cada estructura S = (W, π), se define la función prop : $W \rightarrow ((A \times S_A) \cup (E \times S_A^n)$ como prop(w) = $\{(_{a \in A}\{a\} \times \{|sort(a)|\}) \cup (_{e \in E}\{e\} \times \{|sort^n(e)|\}))\}$tal que ($\pi$(F) = w y de manera que podemos deducir a partir de las fórmulas de primer orden de F las valuaciones a(valor) y e(argumentos))}. Donde $|S_A^n|$ son los n argumentos de un evento[2].

Es decir, proyecta cada uno de los mundos posibles en el conjunto de partes finitas de los pares (*atributo, valor_atributo*) y (*evento, valor_argumentos*).

Observemos que a partir de la función *prop* podemos deducir π de forma unívoca dado que $\pi(f) = \{ w \in W \mid$ *podemos deducir f a partir de prop(w)*$\}$. Esta simetría la expresaremos en adelante de forma simplificada como $w \in \pi(f) \Leftrightarrow f \in prop(w)$.

Definición 2.3 Dada una estructura S = (W, π), un elemento del process p $\in$ P, y un mundo w $\in$ W, entonces un mundo w' $\in$ W es un **mundo p-successor** de w si (w, w') $\in$ m(p). El conjunto de todos los mundos p-**successor** de w en una estructura S se escribe como successor(p)(w), es decir:

$$successor(p)(w) = \{w' \in W \mid (w, w') \in m(p)\} \quad (1)$$

Escribiremos S, w |= ϕ si w $\in$ $\pi(\phi)$. Si $\forall$ w $\in$ W: S, w |= ϕ lo notamos por S |= ϕ y |= ϕ (ϕ es válida) si para todas las estructuras S se cumple que S |= ϕ.

Definición 2.4 La función π se define inductivamente de la forma[3] [21] :

1) $\pi(\phi \vee \psi) = \pi(\phi) \cup \pi(\psi)$
2) $\pi(\neg\phi) = W \setminus \pi(\phi)$
3) $\pi([p]\phi) = \{w \in W \mid \forall w' \in W, (w, w') \in m(p) \Rightarrow w' \in \pi(\phi)\}$

Proposición 2.5 Para cada estructura S = (W, π), un mundo w $\in$ W, fórmulas ϕ, ψ y p $\in$ P deberá cumplirse que [21]:

1) $S, w \models \neg\phi \Leftrightarrow S, w \not\models \phi$
2) $S, w \models \phi \vee \psi \quad \Leftrightarrow S, w \models \phi$ *or* $S, w \models \psi$
3) $S, w \models$ *true*
4) $S, w \not\models$ *false*
5) $S, w \models \phi \wedge \psi \quad \Leftrightarrow S, w \models \phi$ *and* $S, w \models \psi$
6) $S, w \models \phi \rightarrow \psi \Leftrightarrow$ *if* $S, w \models \phi$ *then* $S, w \models \psi$
7) $S, w \models \phi \leftrightarrow \psi \Leftrightarrow$ *if* $S, w \models \phi$ *iff* $S, w \models \psi$
8) $S, w \models [p]\phi \Leftrightarrow \forall w' \in succ(p)(w')$: $S, w' \models \phi$

4. LÓGICA DINÁMICA PARA OASIS

En OASIS [14] se consideran los siguientes tipos de fórmulas en lógica dinámica correspondientes a determinadas secciones de la especificación de una clase.

[1] Ej., el conjunto soporte del sort de los números *Naturales*, es $|Naturales| = \{0, 1, 2, ...\}$

[2] Es importante resaltar que aumentamos el concepto de estado con los eventos pendientes de ejecutarse.

[3] $[p]\phi$ expresa que tras *p*, ϕ es necesariamente cierta.

- **Evaluaciones**. Son fórmulas de la forma $\phi[p]\psi$ cuya semántica viene dada por la función m(p) que a partir del término p ∈ P devuelve una relación entre posibles mundos. Es decir, dada una estructura S = (W, π), estando en el estado[4] w, antes de la ejecución de p se cumplirá que w ∈ π(ϕ) y, además, siendo w' ∈ W un **mundo p-successor** de w, se cumplirá que w' ∈ π(ψ) .

- **Derivaciones.** Son fórmulas del tipo $\phi \rightarrow \psi$ que nos permiten definir atributos derivados (indicados en ψ) en términos de una condición de derivación (expresada en ϕ).

Difieren de las fórmulas de valuación en que las derivaciones relacionan un único estado, es decir, si dado $w \in W$, $w \in \pi(\phi) \Rightarrow w \in \pi(\psi)$.

- **Restricciones de Integridad.** Son fórmulas que deben cumplirse en cada mundo posible. Se distingue entre **estáticas** y **dinámicas** dependiendo de si su definición se expresa sobre el mundo actual o si relacionan varios mundos, respectivamente.

- **Precondiciones.** Son fórmulas del tipo ¬ϕ[p]false. Estando en un mundo w, tal que si S, w |=ϕ y se produce p, entonces no existe un mundo w' alcanzable (dado que en ningún mundo se evalúa false).

La sintaxis de OASIS v.2 permite expresar la precondición anterior de la forma *p if ϕ*.

- **Triggers.** Son fórmulas de la forma ϕ[¬p]false. donde ¬p representa que no se lleva a cabo la acción indicada por p. Es decir, estando en un mundo w tal que S, w |=ϕ, entonces no hay estados alcanzables para ¬p.

Con este tipo de fórmulas dinámicas forzamos la ocurrencia de las acciones, es decir, a que ocurra $(w, w') \in m(p)$ en el estado w si se cumple que $w \in \pi(\phi)$. La sintaxis de OASIS v.2 permite expresar en la sección de Triggers de la forma *p if ϕ* dada la fórmula $\phi[\neg p]false$.

Es importante resaltar que no se indica que no pueda ocurrir otra cosa distinta de *p*, de hecho, esto último debe ser posible, por ejemplo, para indicar el disparo más de un trigger en un estado. Es este caso, se tiene que $p = p_1 \parallel p_2 \parallel \ldots \parallel p_n$, (|| es el operador de composición paralela del Álgebra de Procesos), donde los p_i , i=1, .., n ,son las son las acciones de los triggers $\phi_i [\neg p_i]$ *false,* tales que $S, w \models \phi_i$ i=1, .., n.

- **Process.** La estructura de Kripke definida puede extenderse, tal como hace Wieringa en [24], con el álgebra de procesos de manera que se interpreten los términos de proceso. Es decir, cada término del álgebra de procesos (que representa al proceso que es el objeto) determina una relación de accesibilidad sobre la estructura de Kripke.

De esta forma se puede redefinir la relación de alcanzabilidad:

$$m : T_{Process}(X) \rightarrow (\Sigma \rightarrow (W_1 \rightarrow W_2)) \qquad (2)$$

donde, $T_{Process}(X)$ es el conjunto de términos asociados al álgebra de procesos de la especificación, X es un conjunto de variables, Σ son asignaciones a dichas variables en los sorts respectivos, y W_1 , W_2 son conjuntos de mundos. De esta forma, siendo σ una posible asignación de variables, se llega a que:

$$S, w, \sigma \models [p]\phi \Leftrightarrow \forall p \in P \text{ se cumple } m(p)(\sigma)(w) \models \phi \quad (3)$$

En todas las fórmulas anteriores, ϕ y ψ son fórmulas bien formadas que siguen las siguientes reglas, tal como se define en [14]:

- **Términos.**

- Cada constante de un sort es un término de tal sort.
- Cada variable es un término del sort dado.
- Si $f : s_1 \times s_2 \times \ldots \times s_n \rightarrow s$ es una función, y $t_1, t_2, \ldots, t_n$ son términos, donde cada t_i pertenece al sort s_i, entonces $f(t_1, t_2, \ldots, t_n)$ es un término del sort s.
- Cada atributo es un término del sort dado.
- Sólo son términos los que siguen estas reglas.

- **Átomos.**

- Las constantes *true* y *false* son átomos.
- Si t_1 y t_2 son términos del mismo sort S, y R es uno de los siguientes operadores relacionales (que están definidos para el sort S): =, <>, <, <=, >, >=, entonces $t_1 R t_2$ es un átomo.

- **Fórmulas bien formadas.**

- Cada átomo es una fórmula bien formada.
- Si ϕ y ψ son fórmulas bien formadas, entonces *not* ϕ, ϕ *and* ψ, ϕ *or* ψ, son también fórmulas bien formadas con el significado lógico clásico.
- Sólo son fórmulas bien formadas las que se construyen con estas reglas.

[4] Utilizaremos el concepto de mundo equivalente al de estado

5. EJEMPLO OASIS

Incluimos un ejemplo típico de especificación OASIS:

class Account
identification
number:(number).
constant_attributes
number:**nat**; name:**string**.
variable_attributes
balance:**real**(0); times:**nat**(0); rank:**nat**(0).
derived_attributes
good_balance:**bool**.
private_events var N:**nat**.
open **new**; close **destroy**;
deposit(N); withdraw(N);
pay_commision; change_rank(N).
valuation
balance=M[deposit(N)]balance=M+N and times=times+1;
balance=M[withdraw(N)]balance=M-N and times=times+1;
balance=M[self:pay_commision]balance=M-0.5;
[self::pay_commision] times=0;
[change_rank(N)] rank=N.
derivation
good_balance={balance>=100}.
preconditions
withdraw(N) **if** balance>=N or
(balance<N and rank=2);
close **if** balance=0.
triggers
self::pay_commision **if** times=5 and
good_balance=false and rank=0.
processes
CUENTA = open.CUENTA0;
CUENTA0 = close +
deposit(N).CUENTA0 + withdraw(N).CUENTA0 +
pay_commision.CUENTA0 + change_rank(N).CUENTA0.
end_class

5.1. CONJUNTO DE ATRIBUTOS Y EVENTOS

Conjunto de Atributos A

A = {"number", "name", "balance", "times", "rank", "good_balance"}; S = {"nat", "string", "real", "bool"}

$sort_{number}$ = nat $\in S$; $sort_{name}$ = string $\in SORT$

$sort_{balance}$ = real $\in S$; $sort_{times}$ = nat $\in SORT$

$sort_{rank}$ = nat $\in S$; $sort_{good\_balance}$ = bool $\in SORT$

[5] Asumimos por defecto que el sort relativo al mecanismo de identificación de la instancia está presente en los argumentos del evento. El símbolo ε representa la cadena vacía.

Veremos para esta especificación OASIS las fórmulas asociadas considerando las definiciones anteriores.

Conjunto de Eventos E

El conjunto E de eventos para la especificación del ejemplo es el que se detalla[5]:

E = {"open", "close", "deposit", "withdraw", "pay_commision", "change_rank"}

$sort_{open}$ = nat × string ; $sort_{close}$ = ε

$sort_{deposit}$ = nat $\in S$; $sort_{withdraw}$ = nat $\in SORT$

$sort_{pay\_commision}$ = ε ; $sort_{change\_rank}$ = nat $\in SORT$

5.2. CONJUNTO DE AXIOMAS C

A continuación se detallan los axiomas en lógica dinámica correspondientes a las secciones de Valuaciones, Derivaciones, Precondiciones y Triggers.

Valuaciones

Analicemos para cada una de las valuaciones las propiedades asociadas desglosándolas en la Tabla 1 que se adjunta:

1) balance=M[deposit(N)] balance=M+N and times=times+1;
2) balance=M[withdraw(N)] balance=M-N and times=times+1;
3) balance=M[self:pay_commision] balance=M-0.5;
4) true [self::pay_commision]times=0;
5) true [change_rank(N)]rank=N.

Derivaciones

El único atributo derivado es good_balance la fórmula de derivación viene dada por:

good_balance ← {balance>=100}.

Esta fórmula debe ser desglosada en dos para expresarse de la forma $\phi \rightarrow \psi$:

balance>=100 → good_balance=true.
balance < 100 → good_balance=false.

Precondiciones

La Tabla 2 muestra en qué mundos son posibles las acciones que llevan asociada la precondición:

1) withdraw(N) **if** balance>=N or (balance<N and rank=2);

2) close **if** balance=0.

Val.	*Estado[6] de Partida w*	*Estado de Final w'*	*Acción p*
1)	$w \in W$: S, w |= (balance=M)	$w' \in W$: S, w' |= (balance=M+N and times=times+1)	deposit(N)
2)	$w \in W$: S, w |= (balance=M)	$w' \in W$: S, w' |= (balance=M-N and times=times+1)	withdraw(N)
3)	$w \in W$: S,w|=(balance=M)	$w' \in W$: S, w' |=(balance=M-0.5)	self:pay_commision
4)	$w \in W$: S, w |= *true*	$w' \in W$: S, w' |= (times=0)	self::pay_commision
5)	$w \in W$: S, w |= *true*	$w' \in W$: S, w' |= (rank=N)	change_rank(N)

Tabla 1: Representación de Valuaciones

Prec	*Estado posible w*	*Acción permitida*
1)	$w \in W$: S, w |= (balance>=N or(balance<N and rank=2))	withdraw(N)
2)	$w \in W$: S, w |= (balance=0)	close

Tabla 2: Representación de Precondiciones

Trig	*Estado w de condición de disparo del Trigger*	*Acción que debe disparar*
1)	$w \in W$: S, w |=(times=5 and good_balance=false and rank=0)	self::pay_commision

Tabla 3: Representación de Trigger

Triggers

La Tabla 3 muestra qué transición entre mundos es una obligación cuando se dispara el trigger asociado a la especificación del ejemplo:

1) self::pay_commision **if** times=5 and good_balance=false and rank=0.

La fórmula dinámica del trigger expresa que $\phi[\neg p]false$. Debemos observar que el estado $w \in \pi(\phi)$, es tal que nada más llegar a dicho estado, el trigger debe ser disparado pues estar en w y no haber disparado p lleva a *false* (tal como se define en [14] y se contempla en [9][10][19]). En el estado previo a w es donde el objeto detecta la obligación del disparo.

Process

Para el ejemplo OASIS, si *pre* es la precondición del objeto que representa el proceso y *post* es la precondición postcondición, entonces se cumple que *pre*[*CUENTA*]*post* (donde la definición del término de procesos *CUENTA* que representa al objeto es definida de forma recurrente tal como se indica en la sección de procesos del ejemplo).

6. EVOLUCIÓN DE OBJETOS

En los puntos anteriores se han planteado algunas de las definiciones necesarias para expresar la evolución de objetos que son instancias de clases. Consideraremos una versión simplificada de OASIS en la que se incluyen eventos privados, valuaciones, derivaciones, precondiciones y triggers. La evolución que sufre un objeto desde el instante en que es creado ha sido expresada según el Modelo de Ejecución de forma detallada en [10]. Podemos resumir el ciclo que lleva a cabo un objeto en los siguientes pasos:

[6] En nuetro planteamiento podemos usar el concepto de *estado* de forma equivalente al de *mundo*.

1) Seleccionar de la cola de eventos[7] un subconjunto consistente[8] que cumpla sus precondiciones.
2) Efectuar las evaluaciones asociadas a dichos eventos.
3) Comprobar las condiciones asociadas al disparo de triggers.
4) Volver al punto inicial para alcanzar el nuevo estado y poder entonces disparar los trigger calculados.

Es importante recordar que un objeto puede evolucionar no sólo por las acciones que el resto de la sociedad le demanda (perspectiva servidor), sino también por el hecho de pedir servicios al resto de objetos (perspectiva cliente).

Este hecho se asume en la Figura-1 en la que la acción p que lleva a cabo el objeto puede ser ofrecer o pedir un servicio:

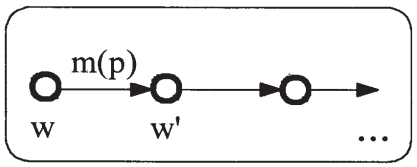


Figura-1: Transición entre estados de un objeto

Definición 4.1 El estado w ∈ W en el que un objeto detecta la obligación de disparar el trigger expresado en la fórmula dinámica $\phi[\neg p]$false, se denomina **estado-detección** y viene determinado por:

- *Siendo $q \in P$ la acción que va a ejecutar, deberá cumplirse que $(w, w') \in m(q) \wedge w' \in \pi(\phi)$.*

Dicho de otro modo, el **estado-detección** de trigger es el estado previo al cual se cumple la condición de trigger.

7. Traducción a un Modelo de Concurrencia. La Función h

Dado que los lenguajes de descripción de sistemas tienden a representar abstracciones de sistemas más que sistemas concretos, el significado de una expresión en dichos lenguajes se da de forma general como *clases de equivalencia* en un nivel de abstracción concreto. Modelos de Concurrencia tales como Redes de Petri, Estructuras de Eventos, Árboles de Sincronización y los Sistemas de Transición-Estado modelan, esencialmente, un *proceso* como una evolución de estados, consecuencia de la ocurrencia de eventos.

Estamos interesados en representar la evolución de la sociedad de objetos de OASIS utilizando como modelo semántico las Redes de Petri Objetuales (OPNs)[8][6][5] [7] tal como hemos hecho en [19] o con Programación Lógica Concurrente (PLC) tal como se hace en [11].

Evidentemente, es necesario justificar de manera formal la traducción que de OASIS se haga a las OPNs o a PLC. Por simplificar el problema, en un primer estudio se abordará únicamente la perspectiva básica de un objeto, como entidad autónoma y aislada que de forma concurrente convive con el resto de la sociedad. Definiremos, dada una representación de nuestra sociedad de objetos basada en teorías de la lógica dinámica, una función de traducción a las redes de Petri objetuales OPN. Tratamos de verificar que la evolución que sufra el objeto visto como teoría, es equivalente a la que tenga como red de Petri o como cláusulas concurrentes.

Es decir, se desea establecer las bases para que el diagrama de la Figura 2 conmute. Tal como se muestra en la Figura 2, dada una teoría que represente a un objeto en un estado w, la ocurrencia de $p \in P$ implicará la transición al nuevo estado w' dado que $(w,w') \in m(p)$.

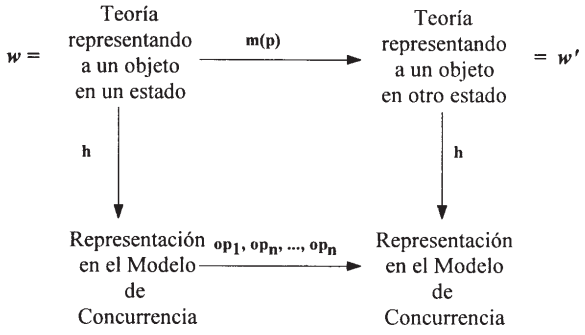


Figura-2: Diagrama de equivalencia

Por otro lado, aplicamos una **función de transformación h** desde la teoría a la representación que se elija para el modelo de concurrencia escogido. Cada operación $p \in P$ en el nivel superior equivaldrá a una o más operaciones op_1, op_2, ..., op_n realizadas en el nivel inferior.

8. Representación de Objetos en Teorías

Tal como se indicó en la definición 2.2, el estado (o mundo) en el que se encuentra un objeto esta representado por la valuación de sus atributos y el conjunto de eventos pendientes de atender. Esto se definió a través de la función $prop(w) = (_{a\in A}\{a\} \times \{|sort(a)|\ ) \cup (_{e\in E}\{e\} \times \{|sort^n(e)|\ ))$. Consideraremos un lenguaje L de la Teoría Clausal de primer orden para expresar el valor de los atributos de la forma *atributo(valor)* y para representar los eventos pendientes de ejecutarse. Este lenguaje se define como sigue:

[7] En el Modelo se contempla el que todo objeto tiene un buffer no acotado de eventos pendientes de atender.

[8] Básicamente, que no actúen sobre el mismo conjunto de atributos.

- **Términos**

– Un elemento del conjunto soporte de un sort S_a, $\forall a \in A$, es un término de dicho sort. (Ej. $t \in |sort(a)|$).

– Sólo son términos los anteriores.

- **Átomos**

– Si $a \in A$ es un atributo y t es un término del soporte $\in |sort(a)|$, entonces $a(t)$ es un átomo.

– Si $e \in E$ es un evento de aridad n con argumentos t_1, ..., t_n que son términos del conjunto soporte S_a^i, i=1,..., n, respectivamente, entonces $e(t_1, ..., t_n)$ es un átomo.

– Sólo son átomos los obtenidos a través de las reglas anteriores.

- **Fórmulas bien Formadas (fbfs) del lenguaje L**

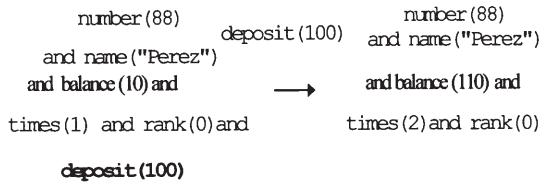
– Un átomo de L es una fbf.

– Si F, G son fbfs, entonces F *and* G es una fbf.\

– Sólo son fbfs de L las obtenidas a través de las anteriores reglas.

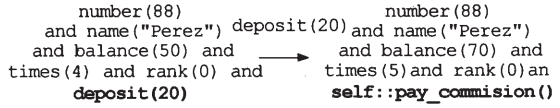
Ejemplo de estado

Un estado para un objeto instancia de la clase Account del ejemplo anterior y expresado como valuación de los atributos podría ser[9] el que se indica a la izquierda y tras el evento deposit(100) pasar a ser el de la derecha:

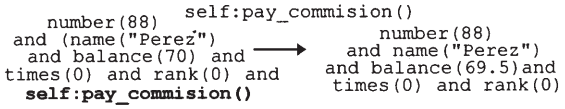


Esta consideración nos ha llevado a incluir como parte del estado la bolsa de eventos pendientes de ejecutar ya que permite representar tanto el hecho de disparar como el que el evento llegue a estar en la cola del servidor del evento. Un *estado-detección* puede ser el que se incluye a continuación en la parte izquierda. El estado al que llega el objeto tras el evento deposit(20) es el de la derecha. Se destaca el tener como evento pendiente de atender self::pay_commision (en algún estado posterior recibirá el trigger self:pay_commision como tal).

[1] Para una ejecución completa de una traza referirse a [12].



Algún instante posterior de recepción del trigger en sí (que podemos considerar por simplificación que sea el siguiente) puede tener:



En este ejemplo, la repercusión del disparo del trigger sólo influye en la teoría del propio objeto. En caso de que el servidor del trigger no sea la propia instancia, el hecho de disparar el trigger conlleva cambios en dos teorías: (1) en la relativa al objeto cliente del servicio, y (2), en la del objeto servidor del mismo.

9. TRADUCCIÓN A REDES DE PETRI

Las redes de Petri modelan los procesos en términos de cómo la ocurrencia de **eventos** *provoca cambios en los estados locales llamados* **condiciones** [25]. Esto se expresa mediante una dependencia causal entre los conjuntos de eventos y los conjuntos de condiciones y es ésta la que determina la conducta dinámica de las redes. En el DSIC-UPV estamos trabajando [19][20] con las mencionadas OPNs [8] por ser un tipo de redes de Petri de alto nivel que incorporan, entre otros, el concepto de *clase*, el mecanismo de *herencia*, el *polimorfismo*, el *enlace dinámico*, arcos inhibidores y de test. Para nuestro objetivo, contemplaremos por ahora el tipo más simplificado y general de redes de Petri tal como se introducen en [25].

9.1. MODELO GENERAL DE REDES DE PETRI

Introduciremos, según se desarrollan en [25], los conceptos necesarios para poder entender la formalización y el proceso de traducción a Redes de Petri.

Definición 9.1 Una **Red de Petri** es una cuatro-tupla (B, E, F, M_0) donde:

1. *B es un conjunto de* **condiciones**,
2. *E es un conjunto de* **eventos**,
3. *F is a multiset of* $(B \times E) \cup (E \times B)$, *llamada* **relación de dependencia causal**,
4. M_0 *es un multiconjunto no nulo de condiciones, llamado* **marcado inicial.**

que satisface las siguientes restricciones,

$\forall e \in E\ \exists b \in B: F_{b,e} > 0$ and $\forall e \in E\ \exists b \in B: F_{e,b} > 0$. (3)

Las redes de Petri se dibujan como grafos en los cuales los eventos (transiciones) se representan como rectángulos y las condiciones (lugares) como círculos con arcos dirigidos entre ellos y pesos asociados con números enteros positivos para representar la relación de flujo. El marcado inicial se representa mediante tokens con la multiplicidad apropiada en cada condición.

9.2. REDES VISTAS COMO ÁLGEBRAS

Es útil, tanto notacional como conceptualmente, ver una Red de Petri [25] como una **2-sorted** álgebra sobre multiconjuntos (entre otras cosas, facilita el cálculo de invariantes).

Las redes tienen una correspondencia 1-1 con una 2-sorted álgebra siendo los sorts μB un multiconjunto de condiciones y μE un multiconjunto de eventos, la constante $M_0 \in \mu B$, y las operaciones unarias siguientes:

$$\bullet(-): E \to \mu B \qquad y \qquad (-)\bullet: E \to \mu B$$

Donde, $\bullet(-)$ establece la precondición (en cuanto a disponibilidad de los recursos indicados en la función) para que ocurra el evento que es su primer argumento. De forma análoga $(-)\bullet$ representa las postcondiciones o recursos generados por la ocurrencia del evento indicado como argumento.

que satisfacen $M_0 \neq \underline{0}$ *and* $((\bullet A = \underline{0}$ *or* $A\bullet = \underline{0}) \Rightarrow A = \underline{0})$,

es decir, la ocurrencia de cualquier evento necesita obligatoriamente consumir y generar recursos.

Evolución Dinámica de las Redes

Los estados de la red se representan mediante *marcados* que son multiconjuntos de condiciones. Típicamente, las condiciones pueden verse como recursos y su multiplicidad como la cantidad de dicho recurso. Cuando un evento se ejecuta consume determinados recursos y produce otros. Esta cantidad viene determinada por la relación F (véase Def. 9.1). De esta manera, si existen recursos suficientes entonces más de un evento puede ocurrir concurrentemente, y, hasta es posible que un evento pueda ocurrir con determinada multiplicidad. Por simplificar el planteamiento se considerará una subclase de las anteriores redes llamadas **contact-free** o **safe**, en las que esta multiplicidad no puede darse.

Sea $N = (B, E, F, M_0)$ una red de Petri. Un marcado M es un multiconjunto de condiciones, es decir, $M \in \mu B$.

Sean M, M' marcados. Sea A un multiconjunto finito de eventos. Se define[10]:

$$M \xrightarrow{A} M' \Leftrightarrow \bullet A <= M \text{ and } M' = (M - \bullet A) + A\bullet \quad (4)$$

Esto nos da la *relación de transición* entre los marcados. Cuando queremos enfatizar la red N en la que se da la transición $M \xrightarrow{A} M'$ escribimos:

$$N: M \xrightarrow{A} M' \qquad (5)$$

Un *marcado alcanzable* de N es aquel marcado M para el cual se cumple que existen sucesivos A_i , i=0, .., n–1:

$$M_0 \xrightarrow{A_0} M_1 \xrightarrow{A_1} \circ\circ\circ \xrightarrow{A_{n-1}} M_n = M \quad (6)$$

Definición 9.2 *Una Red de Petri* $N = (B, E, F, M_0)$ *es* **contact-free** $\Leftrightarrow F <= \underline{1}$ y $M <= \underline{1}$ para todos los marcados alcanzables M.

Para este tipo de redes, una condición es cierta o no para que un evento ocurra, pero sin multiplicidad (redes *safe*).

Proposición 9.3 El comportamiento de una red de Petri $N = (B, E, F, M_0)$ **contact-free** se define como:

1. *A es un conjunto de* **eventos**. Entonces, $\bullet A$ y $A\bullet$ son conjuntos también.
2. Cualquier marcado alcanzable es un conjunto.
3. Siendo M un marcado alcanzable y siendo M' otro marcado de N, entonces :

$$M \xrightarrow{A} M' \Leftrightarrow (\forall e \in A.\ \bullet e \subseteq M) \text{ and}$$

$$(\forall e, e' \in A.\ \bullet e \cap e\bullet = \varnothing) \text{ and } M' = (M \setminus \bullet A) \cup A\bullet \quad (7)$$

Para un red **contact-free**, un evento e se dice que está **habilitado** en un marcado M si $\bullet e \subseteq M$. Si dos eventos e y e' están habilitados en un marcado M y comparten una precondición común, es decir, $\bullet e \cap e\bullet \neq \varnothing$, entonces los eventos e y e' se dice que están en **conflicto** en M. Por otro lado, si tenemos $M \xrightarrow{A} M'$ entonces los eventos en A se dice que ocurren de forma **concurrente**.

9.3. DE OBJETO OASIS A REDES DE PETRI

En [19] hemos visto unas guías para representar los aspectos básicos de comunicación y estructurales de OASIS en las OPNs. Como mencionamos en puntos anteriores, restringiremos el análisis únicamente a la perspectiva individual del objeto. Para exigir que el diagrama de la Figura-2 conmute, vamos a utilizar las redes introducidas en el punto anterior y más tarde podremos extender el estudio a las OPNs.

Previo al proceso de representación, necesitamos de las siguientes definiciones.

[10] Se considera que el lector interpreta las operaciones sobre multiconjuntos. Para mayor detalle, dirigirse a [17].

Definición 9.5 Siendo N = (B, E, F, M_0) una red de Petri representando un objeto instancia de una clase ∈ especificación OASIS y siendo $M_0 \xrightarrow{A_0} M_1 \xrightarrow{A_1} \circ\circ\circ \xrightarrow{A_{n-1}} M_n$ una secuencia de marcados alcanzables, se define un **marcado-estable** M_i el que representa un **nuevo estado** del objeto que se está representado en dicha red, de manera que ∀ j < i, M_j corresponde a un estado anterior a dicho estado.

Con esta última definición se establece una relación entre los posibles estados del objeto en la Teoría con los marcados en la Red de Petri. En la Figura-9.4 vemos que los estados posibles de un objeto tienen correspondencia un subconjunto de los marcados de la red de Petri (los que llamamos *marcados-estables*):

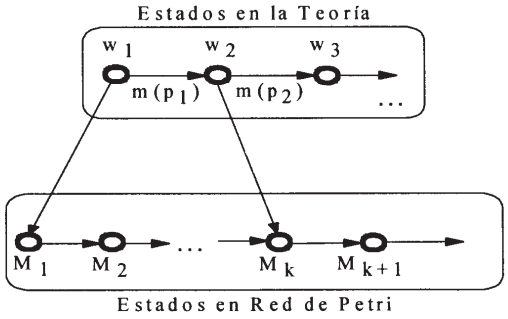


Figura-9.4: Correspondencia entre estados.

Definición 9.6 Siendo N = (B, E, F, M_0) una red de Petri para una especificación OASIS y M_i un **marcado-estable** de dicha red, se define la función **$prop_N$**:

$M_i \rightarrow ((A \times S_A) \cup (E \times S_A^r)$ como $prop_N(n, M_i) = \{(_{a\in A}\{a\} \times |sort(a)|\ ) \cup (_{e\in E}\{e\} \times |sort^r(e)|\ )\}$. Donde n ∈ N, S_A^r son los r argumentos de un evento con el sort asociado ∈ A.

Definición 9.7 Una **función de transformación**[11] **h** es la que asocia a cada estado del objeto su correspondiente marcado-estable.

Proposición 9.8 Decimos que una red de Petri N = (B, E, F, M_0) **satisface** el Modelo de Ejecución [10] de una Especificación OASIS si dada una función de transformación **h** : W → μB y ∀ p ∈ P, se cumple que:

(i)Mi = h(wk) ⇒ propN(Mi) = prop(w) ∀wk∈W, Mi∈M.

(ii)Mi = h(wk) ⇒ Mj = h(wk+1) ∀ wk , wk+1 ∈W, p ∈ P,

siendo w_{k+1} un mundo **p**-*succesor de w_k , M_j ∈ M el marcado-estable sucesor de M_i*

Es decir, hablar del estado desde la teoría que representa al objeto es equivalente a hablar del estado desde la representación que en el modelo de concurrencia de las Redes de Petri. Una vez se haya obtenido una red candidata a ser solución de la especificación OASIS debemos probar que los marcados que alcanza la red de Petri cumple las restricciones anteriores ∀ *p* ∈ *P*. Los formalismos matemáticos definidos sobre ellas permiten el poder calcular analíticamente si se cumplen las propiedades especificadas.

10. CONCLUSIONES

La obtención de prototipos de forma automática que ayuden a la validación temprana de especificaciones es un aspecto importante dentro de las nuevas tendencias en la ingeniería del software. Otro aspecto clave es la utilización de lenguajes formales que permitan la validación de los sistemas especificados de manera formal. Es obvio que las herramientas formales no son agradables al usuario que especifica y en general requiere una formación excesiva para poder abarcarlas. Por otro lado, es deseable obtener un soporte formal que ayude a verificar que el sistema que está especificando es el que realmente se necesita. De esta forma, el usuario no tiene porqué saber que el prototipo que está ejecutando *satisface* de manera formal su especificación; eso es tarea de quien investiga, de quien fija las pautas de construcción de los prototipos. Lo que hemos descrito en este trabajo puede ser extrapolado a la programación lógica concurrente, a otros tipos de redes de petri, y en general, al resto de modelos de concurrencia siempre que sea posible establecer la función de transformación *h* correspondiente.

Seguimos trabajando buscando:

- Incluir aspectos de comunicación más complejos tales como eventos compatidos.
- Considerar estructuras complejas de clases incluyendo la herencia y la agregación de OASIS.

[11] Ver Figura 2.

11. REFERENCIAS

[1] Dubois E., Du Bois Ph., Petit M., *ALBERT: An Agent-Oriented Language for Building and Eliciting Requirements for Real-Time Systems* in Proc. of the 27th HICSS, J.F. Nunamacker and R.H. Sprague (eds), IEEE Computer Society Press, Vol. 4, pp. 713-722, Jaunary 1994.

[2] Dubois E., Du Bois P., Dubru F., *Animating Formal Requirements Specifications of Cooperative Informa-tion Systems* in Proc. of the Second International Conference on Coopera-tive Information Systems, Toronto (Canada), May 1994.

[3] Hartel P., Denker G., Kowsari M., Krone M., Erich H.D., *Information Systems Modelling with TROLL Formal Methods at work*, Information Systems Vol.22, No.2/3, pp.79-99, 1997.

[4] Hartmann T., Saake G., Jungclaus R., Hartel P., Kusch J., *Revised Version of Modelling Language TROLL*, Infor-matik-Bericht 94-03, TU Braunsch-weig, 1994.

[5] Keen C., *Lakos C. Information Modelling using LOOPN++, an Object Petri Net Scheme*, Proceedings of 4th International Working Conference on Dynamic Modelling and Information Systems, 1994.

[6] Lakos C., Keen C., *LOOPN++: A new Language for Object-Oriented Petri*, Proceedings of Modelling and Simulation, Barcelona, Society for Computer Simulation, 1994.

[7] Lakos C., Keen C., *An Open Software Engineering Environment Based on Object Petri Nets*, Department of Computer Science, University of Tasmania, 1995.

[8] Lakos C., *From Coloured Petri Nets to Object Petri Nets*, Proceedings of 16th International Conference on the Application and Theory of Petri Nets, LNCS 935, Torino, Italy, Springer-Verlag, 1995.

[9] Letelier P., Sánchez P., Ramos I., *Animación de Modelos Conceptuales en un Entorno Concurrente*, Informe Técnico, DSIC, Universidad Politécnica de Valencia, 1997.

[10] Letelier P., Sánchez P., Ramos I, *Animación de Modelos Conceptuales para ayudar en la Validación de Requisitios*, ASOO'97, Buenos Aires, 1997 August.

[11] Letelier P., Sánchez P., Ramos I., *Animation of Systems Specifications Using Concurrent Logic Programming*, Symposium in Logical Approaches to Agent and Modelling and Design, ESSLLI'97, Aix-en-Provence, France, 1997, August. (accepted).

[12] Letelier P., Sánchez P., Ramos I., *Guías para la Generación Automática de un Programa Lógico Concurrente que sirve de prototipo para una especificaión OASIS*, Informe Técnico, DSIC, Universidad Politécnica de Valencia, 1997.

[13] Pastor O. *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo orien-tado a Objetos*, Tesis doctoral dirigida por Isidro Ramos, 1992, DSIC Universidad Politécnica de Valencia.

[14] Pastor O., Ramos I., *OASIS versión 2 (2.2) : A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, SPUPV-95.788, Universidad Politécnica de Valencia, 1995.

[15] Pastor O., Barberá J.M., Merseguer J., Romero J., Insfrán E., *The CASE OO-METHOD graphic environment description*. Tech. Report, ITI-DT-96.

[16] Pastor O., Insfrán E., Pelechano V., Romero J., Merseguer J., *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods*. CAiSE97. Barcelona Spain, June 1997.

[17] Pelechano V., Pastor O., *Case OO-Method : Un entorno de producción automática de software orientado a objetos*, Proc.of CIL-95, Barcelona, 1995.

[18] Ramos I., Pastor O., Cuevas J., Devesa J., *Objects as Observable Processes*, Proceedings of the 4th Workshop on the Deductive Approach to Information Systems Design, Catalonia, 1993.

[19] Sánchez P., Letelier P., Ramos I., *Constructs for Prototyping Information Systems using Object Petri Nets*, IEEE System Man and Cybernetics, Orlando, October 1997. (accepted)

[20] Sánchez P., Ramos I., *Object Petri Nets: Modelo de Implementación para OASIS*, Technical Report DSIC-II/17/96, Universidad Politécnica de Valencia.

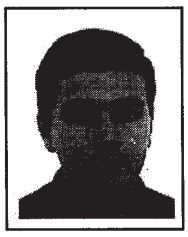
[21] Spruit P., Wieringa R., Meyer J-J., *Axiomatization, declarative semantics and operational semantics of passive and active updates in logic databases*, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam.

[22] Wieringa R.J., *A Conceptual Model Specification Language (CMSL Version 2)*, Technical Report IR-248, Vrije Universiteit, Amsterdam, 1991.

[23] Wieringa R.J., *A Formalization of Objects using Equational Dynamic Logic*, 2nd Conference on Deductive and OO Databases, pags 431-452, Springer 1991, Lecture Notes in Computer Science 566.

[24] Wieringa R.J., Meyer J.-J.Ch, *Actors, Actions, and Initiative in Normative System Specification*, Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

[25] Winskel G., *Categories of Models for Concurrency*, LNCS 197, pp.246-267, 1984.



Pedro Sánchez Palma. *Nacionalidad española, es licenciado en informática, por la Universidad de Málaga, España. Sus áreas de interés son: modelización conceptual de sistemas de información, orientación a objetos, ingeniería de la programación, validación de requirimientos y redes de Petri de alto nivel (Object Petri Nets). Actualmente trabaja como profesor en el departamento de sistemas informáticos y computación en la Universidad de Politécnica de Valencia, Valencia , España y esta desarrollando su tesis doctoral en las áreas antes mencionadas.*



Patricio Letelier Torres. *Nacionalidad chilena, es ingeniero civil en informática, egresado de la Universidad Técnica Federico Santa María, Valparaiso, Chile. Sus áreas de interés son: ingeniería de requisitos, modellización orientada a objetos, animación de modelos conceptuales y programación lógica concurrente. Actualmente trabaja como profesor asociado en el departamento de sistemas infomáticos y computación de la Universisdad Politécnica de Valencia, España y esta desarrollando su tesis para la obtención del grado de Doctor en Informática.*



Isidro Ramos Salavert.*Nacionalidad española, obtuvo el grado de Doctor en Ciencias. Su área de interés: modelización conceptual de sistemas de información. Actualmente es catedrático de Universisdad, director de grupo de Programación Lógica e Ingeniería de Software de la UPV (DSIC).*

Evolución y Perspectivas en el Desarrollo de Software: Nuevas tendencias Orientadas a Objetos.

María Dolores Lozano[1]
Isidro Ramos Salavert[2]

Resumen

En este articulo se presentan los problemas existentes entorno al desarrollo de software, se da una visión general de su evolución y perspectivas y se presenta la nueva tendencia orientada a objetos. El paradigma orientado a objetos está siendo en los últimos años objeto de estudio de muchos equipos de investigación. Con el objetivo de desarrollar Sistemas de Información más cercanos al entorno real como alternativa fiable a las técnicas tradicionales, el modelo orientado a objetos se ha convertido en un método de análisis, diseño e implementación que cada vez está cobrando más importancia en lo que respecta la Ingeniería del Software, Lenguajes de Programación, Bases de Datos y Lenguajes de Especificación. Las ventajas que aporta este modelo impulsan a informatizar los Sistemas siguiendo esta nueva metodología, todavía en fase de crecimiento.

1.- INTRODUCCIÓN.

En el proceso de desarrollo de software, cuando analizamos sistemas[3], creamos modelos de las áreas de aplicación que nos interesan. Se podrían crear modelos enfocados a un área específica del sistema, o cubrir su totalidad. Modelizar la empresa es importante a la hora de plantear su informatización. Por ello, en una primera fase de análisis, se empieza modelizando áreas del Sistema para luego enlazarlas todas juntas y así formar el modelo global [4]. El modelo representa un aspecto de la realidad y se construye de tal modo que nos ayude a entender dicha realidad. Además, es mucho más simple que ella. Manipular el

[1] Prof. ayudante del Departamento de Informática. UCLM.
[2] Catedrático de la Universidad Politécnica de Valencia. Dpto. de Sistemas Informáticos y Computación.
[3] Por sistema se entiende la empresa u organización objeto de estudio que se desea informatizar.

Lozano, M.D., Ramos, I. Evolución y perspectivas en el desarrollo de software: nuevas tendencias orientadas a objetos. Ensayos: revista de la Escuela Universitaria de Magisterio de Albacete, Nº. 12. 1997.

modelo nos ayuda a ver de modo más sencillo como modificar, ampliar o rediseñar áreas de la misma.

Al plantear la informatización de un sistema, es importante modelizar dicho Sistema [4]. Este proceso además, se debe plantear por partes, ya que el Sistema engloba diversos subsistemas que deben abordarse de forma individual para un mejor y más claro entendimiento del mismo. Los modelos de dichos subsistemas ayudan a entender más fácilmente su funcionamiento.

La siguiente figura ilustra el proceso que se sigue para construir sistemas:

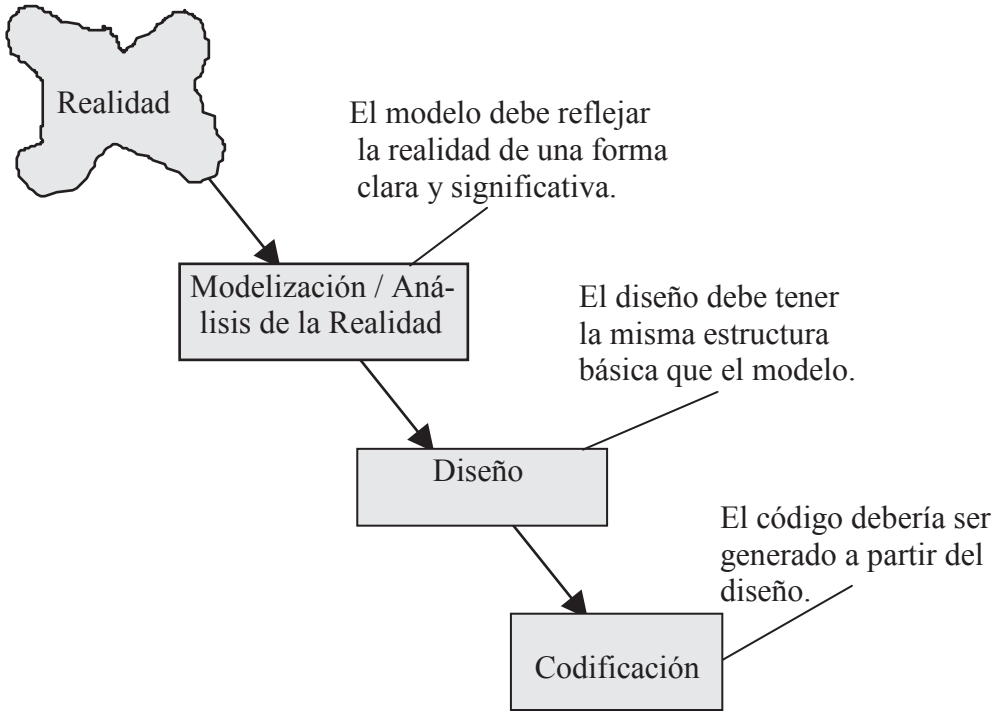


Figura 1: *Cómo informatizar un sistema.*

El proceso es el siguiente:

La fase de **Análisis** consiste en estudiar el problema planteado para obtener una idea clara y concisa sobre los datos de entrada de los que se parte, incluyendo la forma en que estos datos iniciales son obtenidos, así como la información que se desea obtener como resultado y de que manera debe presentarse [4].

Es decir, después de analizar el problema, se han de conocer tres cosas:

1) *Datos* de entrada de que se dispone.

2) *Tratamiento* que ha de realizarse con esos datos.

3) *Información de Salida* deseada.

Por ejemplo, para hacer las facturas de los clientes de una empresa se necesita saber:

1º.- Los datos de cada uno de los clientes a facturar. Si esos datos se van a recibir impresos en papel o si se hallan en un fichero o tabla de Clientes, donde en cada ficha o registro se encuentra toda la información referente a cada uno de ellos.

2º.- La formula matemática para confeccionar el Importe Total de la factura.

$$\text{Importe\_Total (Cliente, mes)} = \sum \text{albaranes(Cliente, fecha)} \ \forall \ \text{fecha} \in \text{mes}$$

3º.- El formato de salida del informe donde se desean imprimir los datos de la factura.

Fase de **Diseño:** Una vez analizado el problema, se precisa diseñar un algoritmo que indique claramente los pasos a seguir para resolverlo. Es decir, es escribir un pseudocódigo de las instrucciones a seguir para llevar a cabo los requerimientos solicitados por el Sistema.

La última fase indicada en la figura 1 es la **Codificación o Implementación**. Para que el algoritmo diseñado en la fase anterior pueda ser introducido

en el ordenador, debe ser escrito (codificado) en un lenguaje de programación, siguiendo las reglas de sintáxis del lenguaje escogido.

En definitiva, se va siguiendo un proceso lineal desde que se plantea el problema hasta que se automatiza en una máquina.

2.- ¿QUÉ ES LA INGENIERÍA DE LA PROGRAMACIÓN?.

El término *Ingeniería* hace referencia a la aplicación de un enfoque sistemático, basado en las Ciencias y las Matemáticas, a la producción de una estructura, máquina, producto, proceso o sistema. Así, la *Ingeniería del Software o de la programación* es el resultado de llevar la tradicional disciplina de las ingenierías al mundo de la construcción de sistemas software. Es una disciplina dependiente de diversas tecnologías: usa *técnicas de gestión* para controlar y gestionar proyectos, y *técnicas de análisis de sistemas* para captar una serie de descripciones que un sistema capaz de ser transformado, mediante la *programación*, a su forma ejecutable en un computador [1].

Surgió como disciplina a partir de la crisis de software producida en los años 70 como consecuencia de la baja calidad del software, su alto coste y un mantenimiento casi imposible.

Así, la Ingeniería del Software abarca tres elementos clave que facilitan el proceso de desarrollo de software y proporcionar a los que practiquen dicha ingeniería las bases para construir software de calidad de una forma productiva. Dichos elementos son:

- Métodos: Suministrar el *cómo* construir técnicamente el software.
- Herramientas: Suministran un soporte automático o semiautomático a los métodos.
- Procedimientos: Integran los métodos y herramientas y facilitan un desarrollo racional y correcto del software.

Ingeniería de la Información es la aplicación de un conjunto de técnicas interrelacionadas para la planificación, análisis, diseño, construcción y mantenimiento de sistemas de información para la totalidad de una empresa o para un sector grande de la misma.

3.- EVOLUCIÓN Y PERSPECTIVAS.

La Ingeniería del Software ha evolucionado rápidamente en las dos últimas décadas. Este avance ha supuesto una auténtica revolución en el mundo del software, ya que se ha pasado de la programación artesanal al desarrollo sistemático de programas que ya pueden ser considerados *productos* desde el punto de vista industrial. En este apartado damos un repaso histórico de cómo han avanzado las técnicas de desarrollo de programas hasta nuestros días y a continuación veremos las perspectivas de futuro de esta disciplina[1].

No entraremos en detalle pero si mencionaremos los principales hitos que han marcado las diversas facetas del desarrollo de programas, fundamentalmente relacionadas con el diseño y la programación.

* La Programación Estructurada: Surge a principios de los 70. Hasta entonces los programadores se habían preocupado más de la eficiencia de los programas en términos de velocidad y uso de la memoria que de su legibilidad o mantenimiento, debido a la baja calidad del hardware, pero a medida que este fue mejorando y los programas aumentaban en complejidad, se empezó a ver la importancia del mantenimiento del software y así surge este estilo de programación que dio lugar a la metodología estructurada.

* Las Metodologías de Diseño: Surgen a mediados de los 70. Las compañías de desarrollo de software se dieron cuenta de que los sistemas construidos directamente a

partir de requisitos informales eran poco eficaces y se necesitaba de una especificación y un diseño del sistema previo a su implementación.

* Las Metodologías de Análisis: Fuertemente relacionadas con las ideas de la Programación Estructurada y las Metodologías de Diseño Estructurado, usrgen a finales de los 70.

* La Automatización de los Métodos de la Ingeniería del Software: Dado que las actividades de desarrollo de software muchas veces consisten en procesos repetitivos o rutinarios, la Ingeniería del Software siempre ha sido una buena candidata a la automatización. Es por ello que surge a principios de los 80. Inicialmente consistió en llevar a una máquina las herramientas gráficas asociadas a determinadas metodologías de análisis o diseño. Posteriormente se han ido perfeccionando, dando lugar a unas mas completas herramientas CASE (Computer-Aided Software Engineering).

* La Introducción de la Programación Orientada a Objetos: Surge a mediados de los 80 y continua siendo la tendencia de los 90. Su influencia se compara a la de la programación estructurada de los años 70 [5]. En los siguientes apartados se comenta de forma mas detallada.

En cuanto a las perspectivas de futuro podemos señalar 4 puntos principales [1]: La programación orientada a objetos que acabamos de mencionar, el refinamiento de requerimientos y prototipado rápido, que ayuda enormemente a identificar correctamente los requisitos de los sistemas, los avances en la tecnología CASE y por último la reusabilidad como solución alternativa al desarrollo constante, es decir, poder disponer de librerías de componentes software fácilmente ensamblables puede ahorrar mucho trabajo a los equipos de desarrollo.

4.- PARADIGMA ORIENTADO A OBJETOS COMO INTENTO DE SOLUCIÓN.

El desarrollo Orientado a Objetos está jugando un papel cada vez más importante en el diseño e implementación de Sistemas Software, hasta el punto de que su influencia está siendo comparada a la de la programación estructurada en los años 70. Se considera como pionero de la aproximación Orientada a Objetos el lenguaje Smalltalk. Lo que en sus inicios fue tan solo un lenguaje de programación dio lugar posteriormente a toda una corriente en el mundo del desarrollo del Software, a cuyo amparo se han desarrollado metodologías de diseño y lenguajes de programación [2].

El *Diseño Orientado a Objetos* es una aproximación diferente a las tradicionales metodologías estructuradas. Estas se basan en la descomposición de un sistema en módulos atendiendo a consideraciones procedimentales y/o de datos. En la Orientación a Objetos, en lugar de descomponer los sistemas en módulos que denotan operaciones, los sistemas se estructuran alrededor de los objetos que existen en el modelo del sistema [3]. Si se consigue identificar todos los Objetos componentes de un cierto sistema, así como las relaciones de cada uno con los demás, se genera una visión del mismo mucho más cercana a la realidad, ya que no se separa entre datos y operaciones, sino que ambos se recogen en la definición de los Objetos.

Un Objeto es visto como una unidad que encapsula estructura y comportamiento. Dicho comportamiento es descrito fundamentalmente a través de las respuestas por parte del Objeto a acciones externas y a través también de las acciones que un Objeto puede requerir a otros. La comunicación entre ellos se realiza a través de mensajes, que no son

más que peticiones de servicio de unos Objetos a otros [7]. De esta manera, una primera propuesta metodológica para el Diseño Orientado a Objetos [6] podría resumirse en las siguientes:

1) Examinar los requerimientos del sistema e identificar los Objetos en el mismo.

2) Para cada Objeto, identificar las acciones que sufre y/o requiere de otros Objetos. En otras palabras, identificar los mensajes que puede recibir y los que puede enviar cada Objeto.

3) Diseñar las Estructuras de Datos apropiadas para representar los datos dentro de cada Objeto, y elegir algoritmos apropiados para implementar las operaciones a realizar por cada Objeto (sus Métodos).

Las Metodologías Orientadas a Objetos (MOO), basan sus principios en una concepción del mundo compuesto de Objetos que se agrupan en Clases en función de sus características particulares y la función que desempeñen dentro del Sistema. Es decir, las MOO identifican tipos de Objetos [9]. Así, entendemos por Clase la implementación de un tipo de Objeto. La Clase especifica la estructura de los datos y los métodos para implementar cada una de las operaciones concernientes a esos objetos.

Es decir, dividimos el entorno en Clases que a su vez están compuestas de Objetos y especificamos los métodos y operaciones características de cada Clase.

Los Sistemas diseñados siguiendo una Metodología Orientada a Objetos suelen a continuación ser implementados a través de un Lenguaje de Programación Orientado a Objetos. Estos lenguajes recogen toda la terminología de la aproximación Orientada a Objetos, y añaden nuevos mecanismos que son interesantes desde el punto de vista de la programación. Entre ellos, el más destacado es el de la *herencia entre clases*.

La idea básica, es definir nuevos elementos Software (es decir, nuevos Objetos) a partir de otros ya existentes, refinando la definición de estos a base de añadir y/o redefinir nuevos atributos o métodos, con la ventaja de heredar características de la Clase madre [7]. Gran cantidad de lenguajes de programación se han desarrollado de cara a ser usados siguiendo esta metodología.

Tres son las líneas en las que ese desarrollo se ha llevado a cabo [1]:

1.- Creación de lenguajes Orientados a objetos. Es la aproximación más pura, y consiste en diseñar nuevos lenguajes que recojan toda la terminología y características de la aproximación Orientada a Objetos. Ejemplos de este tipo de lenguajes son Smalltalk y Eiffel.

2.- Desarrollo de extensiones Orientadas a Objetos de lenguajes clásicos. Consiste en el enriquecimiento de lenguajes ya existentes con nuevas estructuras que den cabida al paradigma objetual. Destacan C++ como la extensión más aceptada del lenguaje C, y algunas de las extensiones del lenguaje LISP, como CLOS.

3.- Uso de lenguajes no orientados a Objetos como tales: existen autores que defienden el uso de las capacidades en cuanto a modularidad, abstracción ,etc., para realizar implementaciones de sistemas diseñados con la metodología orientada a objetos.

5.- CARACTERÍSTICAS DEL MODELO ORIENTADO A OBJETOS.

Una de las características más destacables del paradigma objetual es que aporta una notación apropiada para el desarrollo de cualquier tipo de Software. Esto no ocurría con las metodologías anteriores, que en cierto modo estaban orientadas demasiado a las aplicaciones de proceso de datos, dejando un tanto de lado las aplicaciones de control.

A continuación se enumeran las principales características ventajosas que las tecnologías con orientación a objetos aportan a la modelización de los sistemas [9]. Son las siguientes:

1.- REUSABILIDAD: Las clases son diseñadas para que puedan ser reutilizadas en muchos sistemas. Para maximizar la reusabilidad, las clases pueden ser construidas para poder ser personalizadas. El sistema debería contener una colección de clases reutilizables con posibilidad de crecer. Las librerías de clases tienen posibilidad de crecer rápidamente. Un objetivo primordial de las técnicas OO es facilitar la reusabilidad masiva en la construcción del Software.

2.- ESTABILIDAD: Las clases diseñadas para uso repetido se vuelven estables para facilitar esa reusabilidad.

3.- ENCAPSULACIÓN: Al diseñar se piensa en términos de comportamiento de objetos, no de detalles de bajo nivel. La encapsulación oculta los detalles y hace las clases complejas fáciles de usar. Las clases son como cajas negras, se usa esa caja negra y no se mira dentro de ella. Sólo se tiene que comprender el comportamiento de esa caja negra y como comunicarse con ella.

4.- CONSTRUCCIÓN DE OBJETOS DE COMPLEJIDAD CRECIENTE: Los objetos se construyen a partir de otros objetos que a su vez han sido construidos a partir de otros. Esto permite construir componentes Software complejas que se convertirán en nuevos bloques a partir de los cuales se podrá construir más Software complejo.

5.- FIABILIDAD: El Software construido a partir de clases estables y ya verificadas, es menos probable que tenga defectos que el Software inventado partiendo de cero, sin usar nada ya creado. Cada método en una clase es, en sí mismo, relativamente simple y puede ser diseñado para ser fiable.

6.- NUEVOS MERCADOS DE SOFTWARE: Las compañías de Software podrán proporcionar librerías de clases para áreas específicas, fácilmente adaptables a las necesidades de las organizaciones que las usen. Serán usadas como lo son las componentes Hardware.

7.- DISEÑO MAS RÁPIDO: Las aplicaciones son creadas a partir de componentes ya existentes. Muchas de las componentes se construyen de forma que puedan ser personalizadas para un diseño particular.

8.- DISEÑOS DE MAYOR CALIDAD: Los diseños son a menudo de mayor calidad, ya que se construyen a partir de componentes bien verificados que han sido probados y refinados repetidamente.

9.- INTEGRIDAD: Las estructuras de datos sólo pueden ser usadas con métodos específicos. Es decir, los objetos de una clase solo responden a los métodos definidos esplícitamente para ella. Esto ayuda a crear Sistemas seguros.

10.- PROGRAMACIÓN MAS FÁCIL: Los programas se construyen con pequeñas piezas, que son generalmente fáciles de crear.

11.- MANTENIMIENTO MAS FÁCIL: Esto se consigue ya que cada clase maneja sus operaciones independientemente de otras clases.

12.- CICLO DE VIDA DINÁMICO: El objetivo del desarrollo de sistemas a menudo cambia durante la implementación, pero cabe la posibilidad de adaptarse a los cambios del sistema, refinar los objetivos si así lo requiere la organización y mejorar constantemente el diseño durante la implementación.

13.- REFINAMIENTOS DURANTE LA CONSTRUCCIÓN: A menudo es necesario cambiar el diseño del trabajo durante la implementación. Esto conduce a la obtención de mejores resultados finales.

14.- MODELIZACIÓN MAS REALISTA: El AOO modeliza los sistemas de una forma más cercana a la realidad que el Análisis convencional. El Análisis se traduce directamente en Diseño e Implementación. En técnicas convencionales, el modelo cambia conforme pasamos del análisis al diseño y del diseño a la implementación. Con las técnicas OO, el Análisis, Diseño e Implementación usan el mismo paradigma y sucesivamente lo refinan.

15.- MEJOR COMUNICACIÓN: Existe una mejor comunicación entre los profesionales del Sistema de Información y sus clientes. Los clientes entienden más fácilmente el paradigma OO. Ellos piensan en términos de eventos, objetos y comportamiento de los objetos. Las metodologías OO proporcionan un mejor entendimiento entre los programadores y los usuarios finales, al compartir un modelo común.

16.- ESPECIFICACIÓN Y DISEÑOS DECLARATIVOS: La especificación y el diseño debería ser declarativo siempre que sea posible. Esto permite al diseñador pensar más como el usuario final que como la computadora.

17.- INDEPENDENCIA DE DISEÑO: Las clases se diseñan para ser independientes de los entornos del Software o del Hardware. Emplean peticiones y respuestas de formatos estándar. Esto permite que sean usados con múltiples Sistemas Operativos, manejadores de Bases de Datos, etc. No hay que preocuparse del entorno o esperar hasta que este sea especificado.

18.- AUTOMATIZACIÓN DE BASES DE DATOS DE MAYOR NIVEL: Las estructuras de datos en BDOO (Bases de Datos Orientadas a Objetos) como por ejemplo la presentada en [8], contienen métodos que toman acciones automáticas. Una BDOO incluye "inteligencia" en su contenido en forma de métodos, mientras que las BDR (Bases de Datos Relacionales) no.

19.- MEJORES HERRAMIENTAS CASE: Las herramientas CASE usan técnicas gráficas para diseñar clases y sus interacciones. Estas herramientas facilitan la modelización en términos de eventos, desencadenamiento de acciones, estados de los objetos, etc. Y además generan código tan pronto como se definen las clases.

20.- LIBRERÍAS DE CLASES: Muchos LPOO incorporan librerías de clases que facilitan enormemente la implementación de las aplicaciones.

6.- CONCLUSIONES.

Como conclusión se resumen las diferencias mas destacables entre las metodologias orientadas a objetos y las tradicionales.

Con **métodos Orientados a Objetos**, la forma en que se modela la realidad difiere de los métodos de Análisis Convencional. Con los métodos Orientados a Objetos, se modela el mundo en términos de tipos de Objetos y lo que le ocurre a esos tipos de Objetos [3]. El mismo modelo y la forma de concebir el sistema se mantiene durante las distintas etapas del proceso de modelización del sistema, es decir, durante las etapas de análisis, diseño y codificación. Es decir, se sigue un proceso mucho mas uniforme.

Con las **metodologías tradicionales** se establece una diferencia más notable en las diferentes etapas del desarrollo. Es decir, durante el análisis se elaboran una serie de esquemas y diagramas y una especificación del sistema que sirven de base para el diseño, pero este se lleva a cabo de forma diferente a como se realizó el análisis. Y posteriormente la implementación desarrollada de forma tradicional siguiendo las técnicas de la programación estructurada [4], sigue un modelo diferente.

7.- BIBLIOGRAFIA.

[1] Balzer, R., Cheatman, T.E. and Green, C., Nov. 1983. "Software Technology in the 1990's: Using a New Paradigm". IEEE Computer, pp. 39-45.

[2] Booch, G., 1986. "Object-Oriented Development". IEEE Transactions on Software Engineering.

[3] Meyer, B., 1988. "Object-Oriented Software Construction". Prentice-Hall.

[4] Pressman, R.S., 1988. "Ingeniería del Software. Un enfoque práctico", segunda edición. Mc.Graw-Hill.

[5] Kim, W. And Lochowsky, F.H. (eds.), 1989. "Object-Oriented Concepts, Databases and Applications". ACM Press, Addison-Wesley.

[6] Booch, G., 1991. "Object-Oriented Design with Applications". Benjamin-Cummings.

[7] Pelechano, V., Pastor, O., 1995. "Case OO-METHOD: Un Entorno de Producción automática de Software Orientado a Objetos". Proc. Of CIL-95, Barcelona.

[8] Canós, J.H., Penadés, M.C., Ramos, I., Pastor, O., 1995. "A Knowledge-Base Architecture for Objects Societies". Proceedings of DEXA-95, London.

[9] Lozano, M.D., 1995."Aplicación del Paradigma Orientado a Objetos: OO-METHOD y Especificación OASIS Comparativo con la Metodología Tradicional en la Informatización de una Estación de Servicio". Proyecto Fin de Carrera dirigido por Isidro Ramos, Facultad de Informática. Universidad Politécnica de Valencia.

CONSTRUCTS FOR PROTOTYPING INFORMATION SYSTEMS WITH OBJECT PETRI NETS

P. Sánchez, P. Letelier, I. Ramos
Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera, s/n, 46071 Valencia - Spain
☎ 3877350; Fax: 3877357
email {ppalma | letelier | iramos} @dsic.upv.es

ABSTRACT

OASIS is a Language for the Specification of Object Oriented Conceptual Models. Object Petri Nets (OPNs) support a full integration of object-oriented concepts into Petri Nets. We propose a way to represent object-oriented concepts used in the OASIS language with OPNs as a suitable semantic model for validating software specifications. We have developed a Basic Execution Model for OASIS Specifications including its main features. Communication aspects between objects are taken into account in our proposal (triggering mechanism and shared events). We will consider event preconditions reducing the worlds to be reached, attribute valuations changing the state of objects, creation and deletion of objects, and life cycles of objects. OPNs are an appropriate semantic foundation over build a concurrent software engineering environment for distributed computation because it allows a natural representation of concurrence. We show how the object-oriented concepts of an OASIS Specification are represented into OPNs.

1. INTRODUCTION

OASIS [4][5] is a Formal language for the Conceptual Specification of Information Systems. Nowadays there is a growing interest in formal approaches to model information systems. CMSL [9] and TROLL [1] are languages similar to OASIS that address the system specification in such way. OO-Method [6] is the companion methodology for the OASIS approach.

Is important to animate the system specification by means of a prototype automatically generated. Experiments have been carried out using Petri Nets [7] and Concurrent Logic Programming [2] as semantics domains for OASIS Specifications. These efforts have led to the establishment of a basic Execution Model [3]. This model can be used in order to animate OASIS Specifications implemented over concurrent programming environments. The aim of this paper is to present the mapping between OASIS concepts and Object Petri Nets. OPNs support a complete integration of object-oriented concepts. Inheritance, polymorphism, dynamic binding and multiple levels of activity can also be included.

2. BASIC CONCEPTS OF OASIS

OASIS is a language for Open and Active System Information Specification. In the object model of OASIS, an ***object*** is an observable process encapsulating structure and behaviour. An object is an operational unit that has static and dynamic features. Each object has an internal and unique identifier (*Oid*) that allows identifying the object during its whole existence. Object behaviour is characterised by the actions received, those which the object is able to carry out, and by the actions sent, those which it is able to request. Actions are received and sent by interaction mechanisms between objects.

The object properties are represented by ***attributes***. Thus, the ***state*** of the object in a given moment is determined by the value of its attributes. An ***event*** is the abstraction of a change of state. The state can change after the occurrence of an event that modifies the values of some attributes. An event does not have duration and occurs in an instant of time.

Each object possesses a fixed set of events that could affect it during its existence. The state of the object in a given instant will depend on the events occurred in its life until the moment during which is observed. In this way, an object is seen as an observable process.

The attributes of the object are classified as follows: ***constant*** if its value does not change during the life of the object, ***variable*** if its value can change as a consequence of the occurrence of events. Furthermore, if the value of an attribute is obtained from the value of other attributes, the attribute is said ***derived***.

Two kinds of events can be distinguished: ***private*** and ***shared***. Private events participate in the life of objects of only one class, they appear only in the signature of one class or in several classes belonging to the same hierarchy of specialisation. Shared events are part of the object life of more than one class. These events appear in

0-7803-4053-1/97/$10.00 © 1997 IEEE

Sánchez, P., Letelier, P., Ramos, I. Constructs for Prototyping Information Systems using Object Petri Nets. Proc. of the IEEE International Conference on System, Man and Cybernetics.1997.

the signatures of all the classes that share them. ***Agents*** (also called ***clients***) activate the events and they have that responsibility. The event requested by an agent object to some ***server*** object(s) is called an ***action*** An event will only be relevant in the life of an object if the state of the object is suitable for that. Event ***preconditions*** are formulas and they should be satisfied in order to allow the occurrence of an event.

The objects are not isolated. In OASIS, the interaction between objects is modelled in two forms: by ***events sharing*** and by ***triggers***. Triggers introduce activity in the society of objects, allowing an object to act as an agent whenever some conditions are satisfied in its state.

The values of attributes should follow some rules. These rules are named ***integrity constraints***. They are defined by formulas in some kind of logic. They should be satisfied in the states in the life of the object (static constraints) or among certain sequences of states (dynamic constraints). A group of events that operate as a unit of execution of greater granularity is called a ***transaction***. They follow the two basic principles for transactions: all or nothing, and not observation of intermediate states. It is possible to construct ***processes*** using events and transactions as atomic actions, They allow specifying the correct sequences of actions in the life of the objects.

A ***template*** is a set of common properties that are shared by of several objects. A ***class*** is defined as the group of objects that share the same template. An individual object is called a class instance. The Society of Objects of our object-oriented system is built from primitive, elementary and complex classes. Objects that have only one state constitute the Primitive Classes or domains and always exist. In general, they undertake the domain of the abstract data types. The ***Elementary Classes*** are built from the primitive classes. There are attributes, events, transactions, preconditions, integrity constraints, triggers and process description defined and they characterize the objects of the class.

Complex Classes can be defined from elementary ones by applying operators between classes. The operators that allow the construction of complex classes are ***Aggregation***, ***Association***, ***Specialisation*** and ***Parallel Composition***. These operators are orthogonal, that is, they are independent and they can be combined.

OASIS is a formal specification language that allows one definition of conceptual schemas according to the object model that has been presented. Here is an example representing an account in a bank system.

```
class Account
identification
   by_id:(id).
constant_attributes
   id:nat;
   name:string.
variable_attributes
   balance:int;
   number:nat (0).
derived_attributes
   good_balance:bool.
private_events var N:nat.
   open new;
   close destroy;
   deposit(N);
   withdraw(N);
   pay_commission.
valuation
   [deposit(N)] balance=balance+N and number=number+1;
   [withdraw(N)] balance=balance-N and number=number+1;
   [self:pay_commission] balance=balance-100 and number=0.
derivation
   good_balance={balance>100000}.
preconditions
   withdraw(N) if balance>=N;
   close if balance=0.
triggers
   self::pay_commission if (number>10) and
                           (good_balance=false).
processes
   ACCOUNT =   open.ACCOUNT0;
   ACCOUNT0    =  close +
               deposit(N).ACCOUNT0 +
               withdraw(N).ACCOUNT0 +
               pay_commission.ACCOUNT0.
end_class
```

This concrete syntax is formalized in dynamic logic [8] and it has been used in [5].

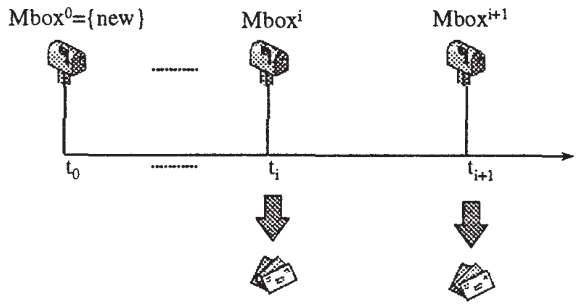


Figure 1: Object life cycle

Figure 1 shows life cycle of an object. It is shown according to the previous concepts. Each object has a mailbox. The change of state does not have duration and occurs at a precise instant of time, when the actions received in the mailbox are executed. To ***attend*** an action means to extract it from the *Mbox*. The attended actions can be ***admitted*** or ***rejected*** depending on they satisfy or not their preconditions. Hence, $Accept^i$ will be the set of admitted actions at the instant t_i , then $Accept^i \subseteq Mbox^i$.

Considering $Exec^i$ as the set of actions executed when the state is $State^i$, then $Exec^i \subseteq Accept^i$. Therefore, an

object will change from the state $State^i$ to the $State^{i+1}$ by the ***execution*** of the actions included in $Exec^i$. Intra-object concurrence is allowed, but the set of executed actions must not be in conflict [3]. Requested actions received by an object are called ***service actions***. For each action that the object must trigger, whenever it reaches the state $State^i$, it will exist another action in the mailbox in the instant t_i. Thus, the server perspective of an object is integrated with its client perspective in a homogeneous form of operation. These actions received by the object due to send triggers are called *trigger actions*. This means, the object must serve service actions as well as trigger actions.

Service actions in $Mbox^i$ are represented by $Mbox^i_{services}$.

Trigger actions in $Mbox^i$ are represented by $Mbox^i_{triggers}$.

Therefore, it is always true that:

$$Mbox^i = Mbox^i_{triggers} \cup Mbox^i_{services}$$

At each t_i , $Mbox^i_{triggers} \subseteq Exec^i$, because all trigger actions must be executed. Thus, $Exec^i$ can additionally contain subsets of accepted service actions. These actions are in conflict neither with one another nor with the trigger actions.

The contents of the mailbox must be updated in each t_i:

(a) Trigger actions or service actions at t_{i-1} have to be extracted.
(b) All the new actions received until t_i must be included.
(c) The actions accepted and not executed in previous states (because they were in conflict) must be stored again.

We have considered all these ideas in order to define a basic execution model including the main features of OASIS specifications. Hence, this model allows animating an OASIS specification accurately.

3. OBJECT PETRI NETS

The enhancements of OPNs [10], [13], [14], include allowing token values to be identifiers, inheritance, test and inhibitor arcs. Furthermore, OPNs provides functions for evaluating the state of the net without changing its state. The possible use of superplaces and supertransitions makes OPNs suitable to model both synchronous and asynchronous interaction between objects. Each instance of an OPN class could be an independent agent passing messages between OPN objects through tokens. The notion of OPN superplace permits synchronous interaction. One transition deposits in (or extracts from) a token of the superplace. This operation is synchronous with an internal transition of the superplace that accepts (or produces) the token. Both transitions can restrict the interaction by means of the adequate guard.

4. CONCEPTUAL SCHEMAS IN OPN

An OPN class at the highest level (meaning the *Conceptual Schema*) is designed to contain references to each OASIS class. Likewise, each OASIS class is represented by a OPN place. As shown in Figure 2, communication between classes is performed through a special *router*.

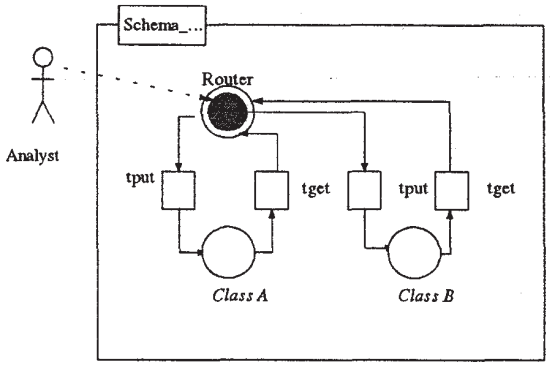


Figure 2: OPN Higest level

The analyst interacts with the running system. All incoming events are routed to the target class. Each event between classes is forwarded through the router, enhancing the facilities to broadcast operations and environment communication.

For example, *ClassA* (more accurately, some instance) sends an event to another class, say *ClassB*. The token extraction, by firing the *tget* transition, is synchronous with the token insertion in the router class. After that, the router class sends the same token (i.e. the event) to *ClassB*. Equally, the token admittance in *ClassB* will be synchronous with the token sent by the router.

5. OASIS CLASSES IN OPN

We will distinguish the OASIS class (which contains the template, the identification of instances, etc.) from the instances that belong to that class. Henceforth, *class* and *instances* will designate OASIS class and OASIS class instances, respectively. For each OASIS class (e.g. *ClassA*), an OPN class (e.g. *ClassA*) and another one to designate the instances (e.g. *InstancesA*) will be defined.

Example 1: The class *Accounts* in our bank system implies two OPN classes: (1) *Class_Account*, and (2) *Instances_Account*.

A general representation for each class is shown in Figure 3. Few details have been shown in order to have a more readable diagram and to emphasize the main interactions.

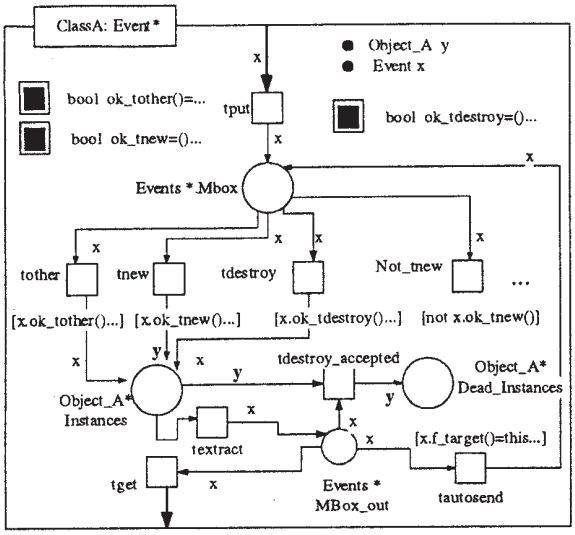


Figure 3: Internal representation of classes.

Notes:

(a) Following notation proposed by Lakos [13], a function is drawn with a double black square, a single transition with a square and a single place with a circle. Arrows reaching the class limits represent synchronous communication. The token type that can be received at the class level (or offered) is named after the class name (*Event* in our example).

(b) The transitions *tput* and *tget* (which redefine the original transitions *put* and *get* for capturing and offering tokens at the superplace, respectively) give the synchronous communication between the router and the classes [12]. Received events are put in the place *Mbox*. The related transition will be fired and questions like identification mechanism, event existence, etc., will be checked. Wherever required, functions (denoted with double black squares) will give information about the state of the object.

(c) The place *Instances* contains every class object. Each event received at the class level is routed to the instance level in a synchronous way.

(d) Each generated event from instances is put in the place *Mbox_out*. If the target is another class, then the event is routed through the *tget* transition. On the contrary, if the target is an object belonging to the same class, then it is forwarded once more to the place *Mbox* (see the transition *tautosend*).

Object Creation

If an object sends an event *new*, two possibilities arise: (1) the preconditions are satisfied and the event is admitted (see *ok_tnew()*), and (2) the event is rejected. The former provides the creation of objects at *Instances* place (note the token type of variable *y* added to the place *Instances* when transition *tnew* is fired). All active objects are tokens of the place *Instances*. Each object has an internal activity [13] that will depend on the OASIS specification. It is important to highlight that preconditions associated to the event *new* are checked by the class instead of the object instance, which is has not been created yet.

Example 2: The creation event *open* of the class *Account* will be received by the OPN class *Class_Account* (through *tput* firing). Then, the transition *tnew* is fired (if *ok_tnew()* is satisfied) and a new object *Account* is added.

Object Destruction

If the class receives an event to remove an object then this event is routed to the correspondent instance. Afterwards, if the event is accepted, the instance replies with another one, confirming the removing action. If the state of the object permits it, then (see *tdestroy_accepted*) it will be moved from the place *Instances* to the place *Dead_Instances*, otherwise it will remain alive.

Example 3: If the event *close* occurs the transition *tclose* will be fired (whether *ok_tclose()* is satisfied), and it will be sent to the corresponding instance of *Account*.

6. OASIS OBJECTS IN OPN

A superplace is also defined for class instances in the same way as it was previously done. The representation of instances is shown in Figure 4:

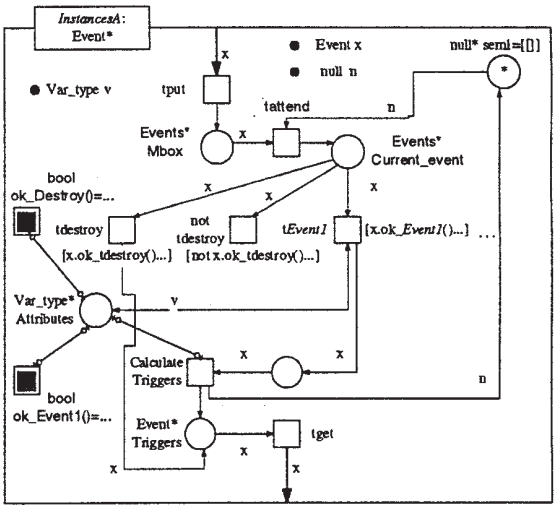


Figure 4: Representation of instances.

Notes:

(a) The events sent from the class level (of which the current object is an instance) are accepted through the transition *tput*. Similarly, events sent from the instance to the class are routed through the

transition *tget*.

(b) The place *Attributes* contains pairs (*attribute, value*) and represents (partially) the state of objects.
(c) The place *Current_Event* contains the selected current event to process next.
(d) The place *sem1* permits to catch only one event each time and to wait for the following state.

Event Preconditions

Event preconditions are always checked (whether they exist or not) by means of defined functions that test the state of the object with tests and inhibitors arcs [11] (see *ok_destroy(), ok_Event1()*, etc.). If an event is accepted then the variable attributes may change. Possible variations of preconditions will only affect these functions. This permits to change the system specification in an easy way. It is possible to break down any Boolean condition into OPN functions by including test and inhibitor arcs.

Example 4: *Balance=0* is the precondition associated to the event *close*. In *Instances_Account* there is a function named *ok_tclose()* that tests the attribute *balance*.

Change of Object State

When an object accepts an event, the associated valuations (coming from OASIS specification) determine the possibility of the modification of the attribute values. The transition *tevent1* achieves the actions that change the tokens representing the variables (in place *Attributes*). After that, the object will probably have a set of triggers to send. Some of those triggers will be forwarded to the same object; others will be sent to another instance (in the same class or another).

Example 5: The event *deposit* implies some valuations, which are performed through the firing of transition *tdeposit*.

Derived Attributes

We can express the relation between the derived and non-derived attributes by defining OPN functions. Each access to a derived variable needs to recalculate its value through the function.

Example 6: The derived attribute *good_balance={balance>100000}* can be expressed with the following LOOPN++ [11] function that can be used in whatever transition:

```
Boolean good_balance() = exists
    Var_type  x -- Attributes | x.balance >100000;
End exists
```

Inter-Object Communication

We are interested in two kinds of communication: asynchronous and synchronous communication. The former gives the triggering mechanism; the second refers to the OASIS shared events.

Triggers: The triggering is performed by:

1. Whoever sends the trigger prepares an event token and offers it through the *tget* transition.
2. The related class gets the token and sends it to the router class.
3. The router class forwards the event to the target class where it is sent to the server.

The transition *Calculate_Triggers* gives the relevant triggers according to the reached state (after valuations). These triggers are offered (transition *tget*) to the related class. Afterwards, the class sends the trigger event to the router or to another instance in the same class.

Example 6: The condition *(Number>10) and (good_balance=false)* enables the trigger *self::pay_Commision*. The object will put the event *pay_Commision* into the place *Triggers*. Afterwards, it is routed to the class *Class_Account*. To make it simple, we have omitted some relevant matters, such as:

1. If the target is the same object, the trigger is get into the place *Current_Event* without outgoing the instance.
2. Transition *Calculate_Triggers* may need several transitions to express the OASIS specifications.
3. Some additional features are needed to allow calculating triggers although there were no event in the previous state.

Shared Events: We need an object (e.g. a coordinating object) that broadcasts toward all the objects that have defined the shared event. Furthermore, it must be executed following an *all-or-nothing* policy. A dialogue between the coordinator and the servers is established and the protocol that follows is similar to the Two-Phase Commit protocol used in Distributed Databases. We have established an algorithm to control efficiently these kinds of events [15].

Process Specification

Possible objects lives are defined by means of a process specification. These processes are constructed using events and transactions as terms.

OPNs gives a natural way to represent process specification expressed by process algebra. Each possible state determines an OPN place. A token in such place indicates that the object is in such state. Changes from one state to another may be restricted with boolean functions (restricted to current state).

A simplified OPN that implements the process section in the example *Account* is shown in Figure 5.

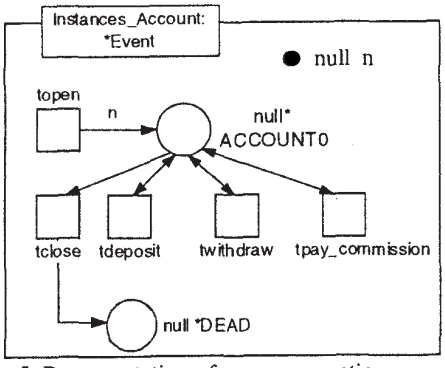


Figure 5: Representation of a process section.

Observe that the transitions included are defined in the same way as previously done in order to indicate acceptance of events. Thus, we can limit transitions firing depending on the current state and on the history of previous events. As already mentioned, condition over variables can be included in process section. As usually, those conditions will be expressed by OPN functions.

7. CONCLUSION

We have shown how the main features of our Object Oriented Conceptual Modelling Language OASIS can be naturally and directly represented in OPNs. Object Oriented concepts (such as classes, instances, interaction between objects, encapsulation, etc.) have been addressed by using the properties of the OPNs.

The support of multiple levels of activity in the OPNs makes the design of architectures easy enough to prototype a society of concurrent objects. Using the execution model of OASIS as a guide, we can guarantee an accurate animation of the OASIS specification. We are now working to obtain LOOPN++ code automatically from an OASIS specification. Our final aim is to integrate this work inside a CASE tool for system modelling supporting the OASIS model.

8. REFERENCES

[1] Hartmann T., Saake G., Jungclaus R., Hartel P., Kusch J., "Revised Version of Modelling Language TROLL", Informatik-Bericht 94-03, TU Braun-schweig, 1994.

[2] Letelier P., Sánchez P., Ramos I., "Conceptos Básicos de Especificaciones OASIS implementados utilizando Programación Lógica Concurrente", Technical Report DSIC-II/32/97, Universidad Politécnica de Valencia, 1996.

[3] Letelier P., Sánchez P., Ramos I., "Un Modelo Básico de Ejecución para Especificaciones OASIS en un Entorno Concurrente", Technical Report DSIC-II/10/97, Universidad Politécnica de Valencia, 1997.

[4] Pastor O., "Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo orientado a Objetos", Tesis doctoral (dirigida por Dr. Isidro Ramos), DSIC-UPV, Valencia, 1992.

[5] Pastor O., Ramos I.,"OASIS versión 2 (2.2): A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach", SPUPV-95.788, Universidad Politécnica de Valencia, 1995.

[6] Pelechano V., Pastor O., "Case OO-Method: Un entorno de producción automática de software orientado a objetos", Proc.of CIL-95, Barcelona, 1995.

[7] Sánchez P., Ramos I., "Object Petri Nets: Modelo de Implementación para OASIS", Technical Report DSIC-II/17/96, Universidad Politécnica de Valencia, 1996.

[8] Wieringa R.J. "A Conceptual Model Specification Language (CMSL Version 2)", Technical Report IR-248, Vrije Universiteit, Amsterdam, 1991.

[9] Wieringa R.J. "A Formalization of Objects using Equational Dynamic Logic", 2nd Conference on Deductive and OO Databases, pages 431-452, Springer-Verlag, Lecture Notes in Computer Science 566, 1991.

[10] Lakos C., "From Coloured Petri Nets to Object Petri Nets", Proceedings of 16th International Conference on the Application and Theory of Petri Nets, LNCS 935, Torino, Italy, Springer-Verlag, 1995.

[11] Lakos C., Keen C., "LOOPN++: A new Language for Object-Oriented Petri", Proceedings of Modelling and Simulation, Barcelona, Society for Computer Simulation, 1994.

[12] Keen C., Lakos C. "Information Modelling using LOOPN++, an Object Petri Net Scheme", Proceedings of 4th International Working Conference on Dynamic Modelling and Information Systems, 1994.

[13] Lakos C., Keen C., "An Open Software Engineering Environment Based on Object Petri Nets", Department of Computer Science, University of Tasmania, 1995.

[14] Lakos C., "The Consistent Use of Names and Polymorphism to Achieve an Elegant Definition of Object Petri Nets", Technical Report R95-12, Computer Science Department, University of Tasmania, 1995.

[15] Sánchez P., Letelier P., Ramos I., "Un Algoritmo para la Gestión de Eventos Compartidos de OASIS", Technical Report DSIC-II/31/96, Universidad Politécnica de Valencia, 1996.

Application of an Object-Oriented Methodology for Automatic Software Production: OO-METHOD

María D. Lozano
Univ. of Castilla-La Mancha

Isidro Ramos
Technical Univ. of Valencia

Pedro Cuenca
Univ. of Castilla-La Mancha

ABSTRACT:

The Object-Oriented Paradigm is currently the subject matter of many research groups [Ser87], [Har94], [Wie97]. With the purpose to develop Information Systems closer to the real environment as a reliable alternative to traditional techniques of Structured Analysis of Yourdon & DeMarco, the Object Oriented Model has become an Analysis, Design and Implementation Method that is getting more and more importance every day regarding to Software Engineering, Programming Languages, Data Base and Specification Languages. The advantages offered by this model encourage us to study and analyse the Systems using this new methodology which is still developing.

INTRODUCTION:

The main goal of this work is to present OO-Method, a methodology for Automatic Software Production**,** which covers the typical phases of Analysis, Design and Implementation within the principles of the Object Oriented Paradigm. This methodology offers an advanced programming environment in which the classical problems of the Software Crisis are solved by using a graphical environment and a Formal and Object-Oriented Specification Language named OASIS (Open an Active Specification of Information Systems)[Pas92], [Pas95]. Moreover, there are some advanced tools that support this methodology. OO-Method has been used in industrial environments for its experimentation and improvement within the frame of a project of the European Union and the Energy and Industry Ministry.

Besides, we also show a specific example in which this methodology is applied in the phases of Analysis and Design, showing the different steps in which it is developed.

THE CASE STUDY:

In order to illustrate the methodology, we will use the following example: A *petrol station* has to control all the activities regarding to clients and purveyors control, including orderings, delivery notes and credit clients control, invoices, tanks and fuel products control, shop products control, employees relays, cash control and all the related matters dealing with a petrol station management. A set of candidate classes emerges within this informal specification: Client, Purveyor, Employee, Products, Invoice, Delivery note, Relays, Cash, ...

Regarding to their corresponding attributes, we can identify particular characteristics belonging to every one of them. The static model with these classes and their relationships is showed in figure 1.

THE METHODOLOGY:

The methodology offered by OO-Method includes the generation of graphical and Object-Oriented Models: the Classes Configuration Diagrams (CCD's) and the Classes State Diagrams (CSD's) which are necessary as a previous phase to the System Specification in OASIS.

We now briefly describe the **methods used in OO-Method**:

1º) <u>Analysis Phase</u>:

In OO-Method, we can define the Object-Oriented Analysis within a double point of view. From one side, we can see it as a *classification process* to identify the existent classes and the relations among them. Relations such as aggregation, association, inheritance, specialisation and generalisation. On the other side, for each one of the identified classes we determine what the possible states of the their instances are, and what the events causing the transitions between one state to another

Lozano, M.D., Ramos, I., Cuenca, P. Application of an Object-Oriented Methodology for Automatic Software Production : OO-Method. Proc. Of the IEEE PACRIM 97. 1997.

are. Apart from the hypothetical conditions associated to the state changes.

The development of the Object-Oriented Analysis Phase consists of five steps:

(a) *Classes Identification,* in this step the elemental classes of the Information System Domain are identified.

(b) *Primitive Attributes Definition,* for each one of the previously identified classes we precise its primitive attributes i.e. those whose types (domains) are ADT's.

(c) *Events Declaration,* for each class in step (a), the set of the their relevant events are declared, distinguishing between private and shared events, and between the former, the new and the destroy events of the classes' instances.

(d) *Activator Classes Identification,* for every class we have to see what services (events) it requires from the rest of the classes in the system, thus we define, for every class, what events can activate it. This is particularly complicated to identify as many events not only have a triggering condition but also are produced by the interaction of several factors.

(e) *Classes Relationship*, the classes of the Object Society are not independent. There are structural and hierarchical relationships, which we are going to model in terms of aggregation (part_of), association and inheritance (is_a), distinguishing between specialisation and generalisation, taking into account both aspects, the dynamic specialisation (role) and the static specialisation (universal specialisation). The existent relations in the system are modelled by means of the Classes Configuration Diagrams. The modularity and abstraction features needed to specify complex systems are offered by this Object-Oriented Model by using inheritance, aggregation and parallel composition operators to create complex classes representing conceptual sub-schemata.

The parallel composition operator comes from the Process Algebra since in the object model used, the objects are observable processes. And the Information System is modelled as a society of full concurrent and interactive objects.

OO-Method offers a graphical notation in a friendly environment to generate the following Graphical and Object Oriented Models as a result of the System Analysis:

* **Classes Configuration Diagrams** (CCD), that constitute the structural / static model of the System. It is an Extended Semantic Model which shows the static structure of the Object System, but which also includes, for each one of the classes, its relevant events and actions together with the classes that act as their agents. Every class in the CCD has associated a Class State Diagram.
In order to represent the structure of the classes society which compose the Information System, we have to follow two steps:

- A Classification Process in order to determine the classes composing the Object System.
- Defining the structure of each class in terms of its set of attributes and events.

Thus, the CCD allows us to describe the classes composing the Object System, by defining its anatomic aspects and the relationships among its classes, getting so an static vision of the Information System as an interactive object society.

Finally, the graphical notations we use for the diagram representation are the classes and the relationships among them. In the next figure we show the resulting CCD obtained in the Analysis Phase of the system.

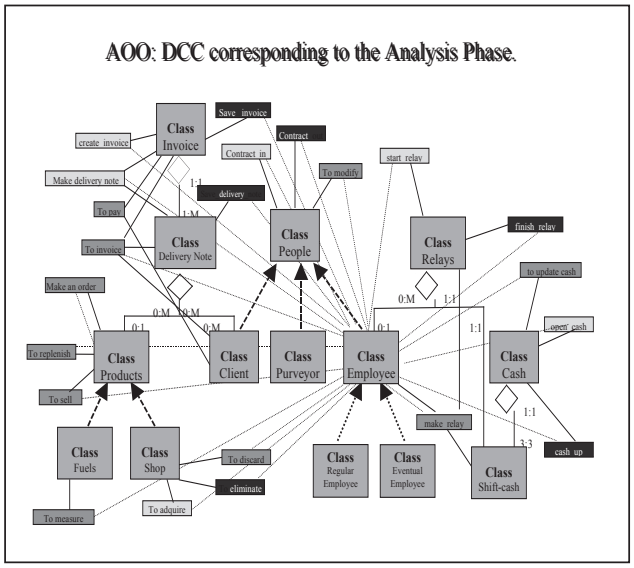


Fig.1: DCC corresponding to the Analysis Phase.

* **Classes State Diagrams** (CSD), as a model of behaviour, that includes dynamic aspects

associated to each class specified in the previous model, i.e. every class in the CCD has a CSD associated to it. In this CSD we represent the potentially valid states for instances of the class and the events that could cause the transitions between these states. Thus, the CSD corresponding to every one of the classes specified in the CCD will include the following information: the *potential states* the class' objects could reach, the *transitions* between these states, the *preconditions* associated to the actions and finally, the possible actions to be activated as a result of the action occurrence which provoked the transition.

As an example, we show in the next figure the CSD corresponding to the class *Product*. There, we can identify the potential states the class can reach following the action occurrence.

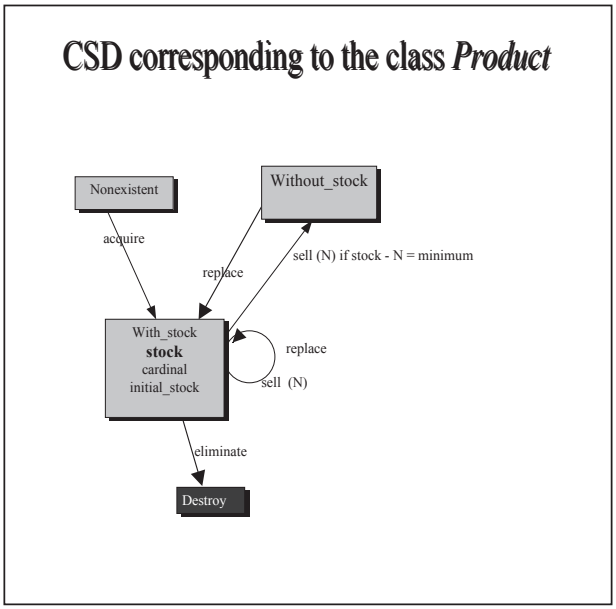


Fig.2: CSD corresponding to the Class *Product*.

In order to obtain the necessary information to construct a CSD we follow the next steps:

- Determine the set of the potentially valid states the objects may reach. Every state will have associated a set of attributes that characterise it. Each attribute is of a certain class: cardinal, state dependent and so on. The attribute class determines the pattern of behaviour with respect to its relevant events. The objects of one class reach or leave these states by means of the actions occurrence. Every state transition is labelled with the name of the action that caused it.
- Describe the behaviour of the object though the definition of the valid transitions between the object states.
- Determine the dependency of the Object State with respect to its relevant actions; so as to automatically design the transition function in the Design Phase.

2º) Design Phase:

The Graphical Models obtained in the first phase of Analysis are translated automatically into an OASIS specification, which is the Design tool in OO-Method. Thus, the design process consists on obtaining the specification code of the Information System in Oasis, starting from the graphical models.

As an example of this, we illustrate a fragment of the obtained Oasis code in the following figure.

OOD: OASIS Specification.

```
Conceptual_Schema Filling Station
domains nat(0), bool(false), string(''), date().
class person generalization of Client, employee,
    purveyor
    identification
        by_code: (cif/nif );
    constant_attributes
        name: string;
        address: string;
        city: string;
        telephone: string;
        cif/nif: string.
end_class
class client
    identification
        by_client_num: (client_num);
    constant_attributes
        client_num: nat.
    variable_attributes
        outstanding: nat;
    derived_attributes
        invoice:nat;
    private_events
        register_client new;
        dismiss_client destroy;
        pay ( N );
        invoice (N)
    constraints
    static
        outstanding > 1 and
        outstanding < 100.000
    valuation
        A, N, M:nat;
        [register_client] outstanding = 0
        outstanding = N [invoice(M)]
                outstanding = N+M.
        outstanding = N [pay (A)]
        outstanding = N-A.
    end_valuation
    preconditions
        dismiss_client if outstanding = 0
    end_class
```

Fig.3: OOD: Fragment of the OASIS specification.

2º) Implementation Phase:

Implementation is tackled in a different way depending on the programming environment used. In this case we translate the OASIS specification into a Logic Dynamic Programming environment

implemented over KBMS1[Ref HP]. Other approaches are taken in the methodology having as target environments standard visual programming tools such as Powerbuilder, Visual C++, and so on. [Pas97]

CONCLUSIONS:

Therefore, we can conclude that starting from these methods, some tools have been developed to cover all the phases of Software Production with a high level of automatization and in a well-defined Object-Oriented environment. Thus, OO-Method offers a set of operational methods which, starting from a graphical specification of the system generates its corresponding OASIS specification and automatically obtains the equivalent logic program.

BIBLIOGRAPHY:

[Can95] Canos,J.H., Penades,M.C, Ramos,I. *A knowledge-Based Arquitecture for Object Societies.* Proc. Of DEXA-95 (Workshop), pags:18-25, London, 1995

[Har94] Hartmann T., Saake, G. et al. *Revised Version of the Modeling Language Troll (Troll version 2.0).* Technische Universitat Braunschweig, Informatik-Berichte, 94-03 April 1994

[Pas92] Pastor,O, Hayes,F., Bear,S. *OASIS: An OO Specification Language.* Proc. Of CAiSE-92 Conference, Lncs (593), Springer-Verlag, 1995

[Pas95] Pastor,O, Ramos,I. *OASIS 2.1.1.: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach.* Octubre 95 (3 ed.)

[Pas96] Pastor,O, Pelechano,V., Bonet, B., Ramos,I. *An OO Methodological Approach for Making Automated Prototyping Feasible.* Proc. Of DEXA-96, Springer-Verlag, September 1996.

[Ser87] Sernadas,A., Sernadas,C., Ehrich,H.D. *OO Specification of Databases: An Algebraic Approach.* In P.M.Stocker, W.Kent eds.,Proc. Of VLDB87, pags: 107-116, Morgan Kauffmann, 1987

[Wie97] Wieringa, R.J., Jungclaus, R., Hartel, P., Hartmann T., Saake G., *OMTROLL Object Modelling in TROLL.* Proc. Of the International Workshop on Information Systems – Correctness and Reusability (ISCORE'93). Hannover, September 1993. Udo W. Lipeck, G. Koschorrek (eds.).

[Pas97] Pastor, O., Insfran, E., Pelechano, V., Romero, J., Meseguer, J. *OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods.* Procs. of 9th. International Conference, CAiSE'97. Barcelona, Catalonia, Spain, June 16-20, 1997.

[Man90] Manley, J. Cox, A., Harrison, K., Syrett M., Wells D., *KBMS1: A User Manual, Information System Centre,* HP-Labs, 1990, Bristol UK.

ASOO'97, In Proceedings of the Argentine Symposium on Object Orientation, Buenos Aires, Argentina, August 1997.

Animación de Modelos Conceptuales para ayudar en la Validación de Requisitos

Patricio Letelier, Pedro Sánchez, Isidro Ramos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n, 46071 Valencia
☎ 3877350; Fax :3877357
email {letelier | ppalma | iramos} @dsic.upv.es

Resumen

La validación temprana de requisitos del software durante el desarrollo de un sistema de información es un aspecto de reconocida importancia. La animación automática a partir de la especificación de requisitos del sistema puede llegar a ser una valiosa ayuda en dicha tarea. OASIS es un lenguaje formal para especificación de modelos conceptuales. En este trabajo, se presenta un modelo básico de ejecución para especificaciones OASIS. Este modelo de ejecución permite animar modelos conceptuales definidos en OASIS construyendo automáticamente prototipos ejecutables para ellos. El modelo de ejecución es definido para un Entorno de Programación Concurrente.

Palabras Clave: Modelización Conceptual, Prototipación.

1 Introducción

OASIS [Past 92][Past 95] es un lenguaje formal para la especificación conceptual de sistemas de información. Actualmente existe un creciente interés en incorporar total o parcialmente técnicas formales para modelar sistemas de información [Past 97] [Hart 97]. Otros lenguajes de especificación formales con enfoques similares al ofrecido por OASIS son ALBERT [Dubo 94a], CMSL [Wier 91a] y TROLL [Hart 94]. La propuesta metodológica basada en OASIS es OO-Method [Past 97]. Se ha desarrollado una herramienta CASE basada en OO-Method [Past 96]. Una funcionalidad atractiva es la posibilidad de animar la especificación del sistema mediante un prototipo generado automáticamente. En esta línea, hemos experimentado usando Redes de Petri [Sánc 97] y lenguajes de Programación Lógica Concurrente [Lete 96][Lete 97b] como dominios semánticos para especificaciones OASIS.

El objetivo de este trabajo es presentar una propuesta de modelo básico de ejecución que permita implementar prototipos para especificaciones OASIS en un entorno de programación concurrente. A diferencia de otros trabajos de animación de especificaciones formales [Dubo 94b] [Herz 95], el modelo de ejecución es dado explícitamente y con suficiente detalle para su posterior implementación. Por supuesto, la fidelidad de la animación respecto del modelo conceptual presenta los mismos problemas que se tienen en la fase de pruebas para verificar la corrección de la implementación respecto del modelo conceptual. Sin embargo, en nuestro caso, la problemática de verificar la corrección de la animación respecto del modelo conceptual se ve apoyada por dos factores importantes: el modelo conceptual se especifica en un lenguaje formal y el modelo de ejecución está inspirado en la semántica de dicho lenguaje formal. Aunque esto no determina la corrección exigida, provee un camino de formalización en el cual estamos trabajando.

2 Validación de Requisitos

Durante la fase de modelización conceptual se llevan a cabo esencialmente cuatro tareas [Dubo 91]:

1) Detección de requisitos: descubrir lo que el usuario quiere obtener como sistema
2) Modelización: describir el modelo conceptual del sistema.
3) Análisis: verificar que el modelo satisface ciertos criterios de calidad y consistencia.

Letelier, P., Sánchez, P., Ramos, I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitios. Proc. of the Argentine Symposium on Object Orientation. 1997.

4) Validación: determinar si el modelo conceptual reúne los requisitos informales obtenidos del usuario.

Estas tareas se realizan cíclicamente hasta que en 4) se obtiene la conformidad del modelo conceptual respecto de los requisitos informales del usuario.

Varias técnicas han sido desarrolladas para la validación de especificaciones conceptuales. Estas técnicas pueden clasificarse de la siguiente forma [Dubo 94b]:

- **Técnicas de conversión**: presentan el modelo conceptual en una forma que se supone es más fácil de comprender por el usuario.

- **Técnicas de comportamiento**: presentan una ejecución de la especificación, lo que permite en un sentido amplio observar y experimentar propiedades de la especificación.

- **Técnicas de análisis**: se basan en el estudio de trazas posibles obtenidas de la especificación y contrastadas con el comportamiento esperado. Para obtener trazas de la especificación puede utilizarse una técnica de comportamiento.

Entre las técnicas de comportamiento, la prototipación ha demostrado ser una ayuda efectiva para apoyar la tarea de validación de requisitos. Herramientas típicas de prototipación son los asistentes para diseño de pantallas, generadores de informes, generadores de programas, animadores de sistemas, etc.

Si la modelización conceptual del sistema está basada en un lenguaje formal tanto la tarea 3) como la 4) se ven beneficiadas. En este caso, el modelo del sistema en sí contiene la información necesaria para realizar la tarea de análisis, con la posibilidad de verificar gran parte de los criterios de calidad de forma automática. Para la tarea de validación se tiene la posibilidad de obtener automáticamente un prototipo a partir del modelo conceptual.

El propósito es que el prototipo obtenido pueda exhibir algunos aspectos importantes de la funcionalidad del sistema, dejando de lado otros como eficiencia, seguridad y diseño de interfaces con calidad de producto final.

El disponer de un prototipo del sistema permite que el usuario y el analista juzguen el modelo conceptual de una forma más cómoda, principalmente para el usuario. Dos de los inconvenientes principales para la incorporación de métodos formales en el desarrollo de software son: por una parte la formación del personal informático y por otra la facilidad para que el usuario de su conformidad a las especificaciones del sistema. En este último aspecto la prototipación es un buen aliado para la aceptación y uso de métodos formales.

3 Nociones Básicas de OASIS.

OASIS es un lenguaje formal para especificación de sistemas de información abiertos y activos. Comenzaremos dando una descripción informal de los principales conceptos incluidos en OASIS. A continuación se muestra un ejemplo de especificación OASIS de una cuenta corriente en un sistema bancario.

```
class cuenta
identification
    by_num:(num).
constant_attributes
    num:nat;
    nombre:string.
variable_attributes
    saldo:real;
    veces:nat(0);
    categoria:nat(0).
derived_attributes
    buen_saldo:bool.
private_events var N:nat.
    abrir new;
    cerrar destroy;
    ingreso(N);
    reintegro(N);
    pagar_comision;
    asignar_categoria(N).
valuation
    [ingreso(N)] saldo=saldo+N and
                 veces=veces+1;
    [reintegro(N)] saldo=saldo-N and
                   veces=veces+1;
    [self:pagar_comision] saldo=saldo-0.2;
    [self::pagar_comision] veces=0;
    [asignar_categoria(N)] categoria=N.
derivation
    buen_saldo={saldo>50}.
preconditions
    reintegro(N) if saldo>=N or
              (saldo<N and categoria=2);
    pagar_comision if categoria=0;
    cerrar if saldo=0.
triggers
   self::pagar_comision if veces>=3 and
                    buen_saldo=false and
                    categoria=0.
processes
    CUENTA =abrir.CUENTA0;
    CUENTA0=cerrar +
            ingreso(N).CUENTA0 +
            reintegro(N).CUENTA0 +
            pagar_comision.CUENTA0 +
            asignar_categoria(N).CUENTA0.
end_class
```

Esta sintaxis concreta encierra una formalización en lógica dinámica que esbozaremos a continuación. Sea F un conjunto de fórmulas de primer orden que representan abstracciones de propiedades de estados. Se usará $\phi,\varphi,...$ para denotar elementos de F. Sea P un conjunto de procesos como una abstracción de cambios de estados. Se utilizará $p,q,...$ para denotar elementos de P.

La semántica declarativa de la Lógica Dinámica [Wier 91b] está definida con respecto a una estructura dada o modelo $U = (W,\tau,\rho)$, donde W es un conjunto cuyos elementos son llamados ***estados*** o ***mundos***. τ y ρ son funciones tales que:

$$\tau : F \rightarrow 2^W$$

$$\rho : P \rightarrow 2^{W \times W}$$

Informalmente, la aplicación τ asigna a una fórmula el conjunto de estados en los cuales se satisface. ρ asigna una relación binaria entre estados a cada proceso.

La interpretación dada es la siguiente: $(s,t) \in \rho(p)$ si y sólo si el proceso p puede alcanzar el estado t a partir del estado s. Una transición que lleva a p desde s a t es producida por la ejecución de un conjunto de acciones en un determinado estado del proceso.

Sintácticamente, una especificación OASIS puede ser formalizada como una signatura, un conjunto de axiomas (los cuales son fórmulas en lógica dinámica) y un conjunto de ecuaciones con variables de *sort process* resueltas en un álgebra de procesos. Las fórmulas dinámicas de OASIS en las cuales basaremos nuestro Modelo Básico de Ejecución son:

- **Evaluaciones** : ϕ $[p]$ ϕ', cuya semántica declarativa está dada por la relación ρ, tal que estando el objeto (representado por el proceso p) en un posible estado válido, si $(s,t) \in \rho(p)$ entonces $s \in \tau(\phi)$ y $t \in \tau(\phi')$, siendo t el estado alcanzado por p producto de la ejecución de acciones.

- **Precondiciones** : $\neg\phi$ $[a]$ *false* , la fórmula ϕ es evaluada en el estado a partir del cual se ejecutaría a. Así, sólo en los estados donde ϕ se satisface es posible la ejecución de a.

- **Triggers** : $\phi[\neg a]$ *false* , $\neg a$ se interpreta como la no-ejecución de a (no implica que otras cosas deban o no ejecutarse). Si ϕ se satisface y a no se ejecuta entonces no existe un estado siguiente alcanzable. Esto obliga a que a se ejecute en cada estado en el cual ϕ se satisface.

Nótese que a, en las fórmulas dinámicas para evaluaciones y precondiciones, es la acción solicitada al objeto, en cambio a en la fórmula dinámica de *triggers* se refiere a la acción "lanzar acción"; esto acontece en la vida del objeto que dispara la acción.

4 Ciclo de Vida de un Objeto

La vida de un objeto está ordenada respecto del tiempo. Existe un objeto "reloj" que lanza eventos especiales –llamados *ticks*- hacia cada objeto del sistema. Los *ticks* recibidos por un objeto son correlativos con los números naturales, t_1, t_2, ... Sean i y j números naturales tal que $i < j \Leftrightarrow t_i < t_j$. Estos *ticks* permitirán establecer instantáneas de diversos aspectos del objeto en un instante t_i concreto.

$State^i$ representa el estado del objeto en el intervalo $[t_i , t_{i+1})$. Desde de t_i y hasta justo antes de t_{i+1} el estado se mantiene constante.

Cada t_i el objeto alcanza un nuevo estado. El conjunto de acciones recibidas para ser servidas por el objeto estará contenido en lo que se denominará ***buzón***. El contenido del buzón en el instante t_i estará representado por $Mbox^i$.

En cada t_i el conjunto $Mbox^i$ podrá contener dos tipos de acciones: ***acciones de servicio*** y ***acciones de disparo***. Las acciones de servicio son aquellas solicitadas por otros objetos o por él mismo. Las acciones de disparo son acciones depositadas automáticamente para indicar que en dicho instante (cuando se alcance el nuevo estado) el objeto está obligado a disparar (solicitar)[1] ciertas acciones.

Las acciones que el objeto está obligado a disparar son los *triggers* de la especificación que satisfacen su condición de disparo. Por cada acción que el objeto deba disparar al alcanzar el estado $State^i$ existirá una acción de disparo en su buzón en el instante t_i. Es decir, el objeto debe atender tanto acciones de servicio como acciones de disparo.

[1] Se ha optado por una manipulación homogénea de todo lo que le puede acontecer a un objeto. Tanto las acciones que el objeto debe servir como las que debe disparar se incluyen en el *Mbox*.

Las acciones de servicio contenidas en $Mbox^i$ se denotan por $Mbox^i_{services}$. Las acciones de disparo en $Mbox^i$ se denotan por $Mbox^i_{triggers}$. Implícitamente, todas las acciones llevan una etiqueta temporal puesta por el agente y otra por el servidor, relativa al instante en el cual es atendida. Así, se cumple que:

$$Mbox^i = Mbox^i_{triggers} \cup Mbox^i_{services}$$

Consistente con la fórmula $\phi\ [\neg a]\ false$, antes de alcanzar el estado siguiente $State^{i+1}$, el objeto está obligado a disparar las acciones de disparo asociadas a las fórmulas ϕ que se satisfacen en t_i. Dichas acciones de disparo no tienen precondiciones. Sin embargo, una precondición para una acción de disparo siempre puede modelarse añadiendo dicha condición a la condición de disparo.

La Figura 1 muestra el ciclo de vida de un objeto según lo planteado hasta este punto. ***Atender*** una acción significa extraerla de $Mbox^i$. En cada t_i, todas las acciones contenidas en $Mbox^i$ son atendidas. Las acciones atendidas pueden ser ***admitidas*** o ***rechazadas*** según satisfagan o no sus precondiciones. Denotaremos por $Accept^i$ al conjunto de acciones admitidas en t_i, $Accept^i \subseteq Mbox^i$.

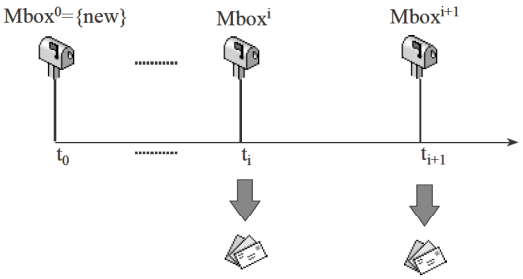


Figura 1: Ciclo de Vida del Objeto

Sea $Exec^i$ el conjunto de acciones ejecutadas en el estado $State^i$, se cumple que $Exec^i \subseteq Accept^i$. Así, un objeto cambiará desde el estado $State^i$ al $State^{i+1}$ por la ***ejecución*** de las acciones contenidas en $Exec^i$.

Se permite concurrencia intra-objetual con la restricción de que el conjunto de acciones ejecutadas en un cambio de estado no esté en conflicto, es decir, que cada acción efectúe modificaciones sobre distintos conjuntos de atributos (como se define en [Lete 97a]).

En cada t_i se cumple que $Mbox^i_{triggers} \subseteq Exec^i$, pues todas las acciones de disparo deben ser ejecutadas. Así, $Exec^i$ podrá contener adicionalmente un subconjunto de acciones de servicio admitidas que no estén en conflicto entre ellas ni con las acciones de disparo.

Las acciones admitidas en t_i y no ejecutadas por estar en conflicto serán incluidas en $Mbox^{i+1}$, es decir, quedan pendientes para ser consideradas al alcanzar el siguiente estado. La admisibilidad de dichas acciones volverá a evaluarse en el nuevo estado alcanzado.

El contenido del buzón debe estar actualizado en cada t_i:

- Las acciones de disparo o de servicio presentes en t_{i-1} habrán sido extraídas.
- Deben aparecer todas las nuevas acciones recibidas hasta t_i inclusive.
- Deben volverse a depositar las admitidas y no ejecutadas en el instante anterior por estar en conflicto.

5 Modelo de Ejecución

En cada instante t_i, además de ejecutar el evento *tick* y considerando el estado del objeto en dicho t_i, el objeto podrá:

a) Ejecutar todas las acciones de disparo que satisfacen sus condiciones de *trigger* en t_i.
b) Ejecutar un conjunto de acciones de servicio que satisfacen sus precondiciones en $State^i$.
c) (a) + (b)

El comportamiento de un objeto estará regido por un bucle de actividad intra-objetual. Esta actividad realiza la transición del objeto desde un estado inicial a un estado final.

Externamente el estado del objeto sólo es observable en los instantes t_i. El reloj podría ser implementado local o distruibuidamente. La separación real de tiempo entre *tick* y *tick* podría no ser constante.

Finalmente, el modelo básico de ejecución se resume en el siguiente algoritmo:

FOR EACH t_i

$Accept^i \neg \{tick(t,i)\}$

IF $Mbox^i_{triggers} \neq \emptyset$ **THEN**

$Accept^i \neg Accept^i \cup Mbox^i_{triggers}$

send_triggers($Mbox^i_{triggers}$)

END_IF

IF $Mbox^i_{services} \neq \emptyset$ **THEN**

$Accept^i \neg Accept^i \cup$

check_conditions($Mbox^i_{services}$, $State^i$)

END_IF

$Exec^i \neg$ **select_actions**($Accept^i$, $State^i$)

$State^{i+1} \neg$ **calculate_new_state**($State^i$, $Exec^i$)

$Mbox^{i+1} \neg Mbox^{i+1} \cup (Accept^i - Exec^i)$

$\cup$ **calculate_triggers**($State^{i+1}$)

END_FOR

En la última sentencia se considera que mientras el objeto está efectuando su cambio de estado, *Mbox* puede estar recibiendo acciones desde otros objetos de la sociedad. Por esto se añaden acciones con la operación unión de conjuntos en lugar de asignar un valor.

Notas aclaratorias

Intencionalmente el modelo se ha establecido a un nivel de abstracción alto. Así, se deja suficiente libertad para su implementación en diferentes entornos de programación. En [Lete 96][Lete 97b] y [Sánc 97] se han realizado implementaciones concretas usando lenguajes de programación lógica concurrente y LOOPN++, un lenguaje para obtener una representación ejecutable para Redes de Petri Orientadas a Objeto.

a) **Definición 5.1** : Sea *ACC* el conjunto de *acciones de servicio* que puede reconocer el objeto, *S* el conjunto de los estados válidos alcanzables por el objeto, entonces :

check_conditions : $2^{ACC} \times S \rightarrow 2^{ACC}$

Acc_1, $Acc_2 \subseteq ACC$, $s \in S$, entonces, *check_conditions*(Acc_1,s)=Acc_2 si $Acc_2 \subseteq Acc_1$ y para cada $acc_2 \in Acc_2$ se satisfacen sus precondiciones en *s*.

La función *chek_conditions* debe obtener un subconjunto de las acciones recibidas contenidas en $Mbox^i_{services}$, tal que dichas acciones satisfacen sus precondiciones en $State^i$.

b) **Definición 5.2** : Sea *ACC* el conjunto de *acciones* que puede reconocer el objeto, *S* el conjunto de los estados válidos alcanzables por el objeto, entonces:

select_actions : $2^{ACC} \times S \rightarrow 2^{ACC}$

Acc_1, $Acc_2 \subseteq ACC$, $s \in S$, entonces, *select_actions*(Acc_1,s)=Acc_2 si $Acc_2 \subseteq Acc_1$ y Acc_2 es un conjunto consistente de acciones en el estado *s*.

La función *select_actions* debe obtener un subconjunto de acciones que no estén en conflicto entre ellas. Dicho conjunto tendrá al menos todas las acciones de disparo en $Mbox^i_{triggers}$. La función *select actions* recibe el conjunto de acciones resultante de aplicar *check_conditions*, es decir, acciones que satisfacen sus precondiciones. Así, a partir de $Accep^i$ y $State^i$, se obtiene $Exec^i$.

c) **send_triggers** es un procedimiento que a partir de cada acción de disparo en $Mbox^i_{triggers}$, envía la acción correspondiente al servidor.

d) **Definición 5.3** : Sea *S* el conjunto de los estados válidos alcanzables por el objeto y *ACC* el conjunto de acciones que puede reconocer el objeto, entonces :

calculate_new_state : $S \times 2^{ACC} \rightarrow S$

$s_1, s_2 \in S, Acc \subseteq ACC$, si s_2 es el estado alcanzado al ejecutar las acciones en *Acc* y s_1 el estado previo), entonces se cumple que: *calculate_new_state*(s_1,*Acc*)=s_2

La función *calculate_new_state* obtiene el conjunto de pares atributo-valor que representa el siguiente estado que alcanzará el objeto, representado por $State^{i+1}$. Las evaluaciones (*valuations*) definen cuál es el valor de cada atributo después de ejecutar las acciones de $Exec^i$.

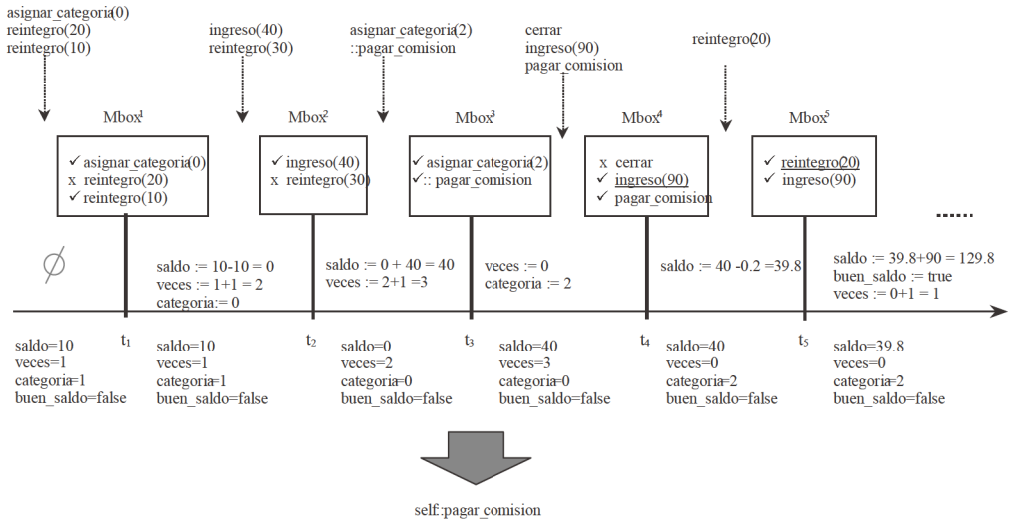


Figura 2: Ciclo de Vida de una instancia de `cuenta` entre t_0 y t_5

e) **Definición 5.4** : Sea S el conjunto de los estados válidos alcanzables de un objeto, ACC_t el conjunto de *términos acción de disparo* que puede reconocer el objeto, entonces :

calculate_triggers : $S \rightarrow 2^{ACC_t}$

$Acc_t \subseteq ACC_t$, $s \in S$. Entonces, *calculate_triggers(s)*= Acc_t si Acc_t es el conjunto de acciones de disparo asociadas a triggers que satisfacen su condición de disparo en *s*.

La *función calculate_triggers* obtiene el conjunto de acciones de disparo correspondientes a *triggers*. Las de dichos triggers se satisfarán en el siguiente estado que alcance el objeto. Así, a partir de $State^{i+1}$ se determina $Mbox^{i+1}_{triggers}$.

6 El Modelo de Ejecución en Funcionamiento

Usando la especificación anterior OASIS de una cuenta bancaria, realizaremos una animación del ejemplo basada en el modelo de ejecución propuesto. La Tabla 1 muestra una posible secuencia de eventos recibida por un objeto de la clase `cuenta`.

La Figura 2 muestra parte del ciclo de vida de un objeto (entre t_1 y t_5). Para esta instancia de la clase `cuenta`, consideraremos que su estado actual (antes de t_1) es: `num=1010, nombre='juan', saldo=10, veces=1, categoría=1 y buen_saldo=false`

El eje horizontal representa el tiempo de izquierda a derecha. Bajo el eje del tiempo se indican los valores de atributos variables del objeto.

Intervalo (t_i - t_{i+1}]	Acciones Recibidas

(... - t_1]	`asignar_categoria(0)`
	`reintegro(20)`
	`reintegro(10)`

(t_1 - t_2]	`ingreso(40)` `reintegro(30)`
(t_2 - t_3]	`asignar_categoria(2)` `::pagar_comision`

(t_3 - t_4]	`cerrar`
	`ingreso(90)`
	`pagar_comision`

(t_4 - t_5]	`reintegro(70)`

Tabla 1. Acciones recibidas entre t_0 y t_5

El estado del objeto es sólo observable en cada t_i, por lo tanto, para un observador, entre [t_i – t_{i+1}) el estado del objeto no cambia. Sobre el eje del tiempo y bajo los rectángulos se indica los cambios que sufrirán los atributos variables y que se reflejarán en el siguiente t_i. Los rectángulos representan el estado del buzón ($Mbox^i$) en cada t_i.

El "✓" delante de un evento en el $Mbox^i$ indica que el evento satisface sus precondiciones en el instante t_i,

la "x" delante de un evento indica lo contrario. Los eventos recibidos entre (t_i – t_{i+1}] están indicados por las flechas descendientes entre $Mbox^i$ y $Mbox^{i+1}$. La flecha ancha inferior representa un disparo hecho por el objeto. En este ejemplo, el disparo es hacia sí mismo (servidor=`self`) y se ha supuesto que la acción es recibida durante el mismo intervalo en el que es lanzada (podría sufrir algún retraso en el sistema de comunicación). Nótese que en t_3 se recibe la acción de disparo que señala la obligación de disparar el evento `pagar_comision`, posteriormente en t_4 es atendida la acción de servicio `pagar_comision`.

En t_4 y t_5 el buzón contiene dos acciones que satisfacen sus precondiciones pero que forman un conjunto inconsistente pues en ambos instantes el atributo saldo puede recibir más de una modificación. Se han subrayado las acciones que satisfacen sus precondiciones pero que forman un conjunto inconsistente. En t_4 la acción `ingreso(90)` está en conflicto con la acción `pagar_comision`. Se ha supuesto que `ingreso(90)` es postergada hasta el siguiente *tick,* siendo puesta en $Mbox^5$. En t_5 la acción `ingreso(90)` forma un conjunto inconsistente con `reintegro(20)`. En este caso se ha supuesto la ejecución de `ingreso(90)` y la postergación de la otra. Podemos observar, que la implementación concreta de la función *select_actions* (que selecciona el conjunto consistente de acciones a ser ejecutadas) debe evitar postergación indefinida y quizás el tratamiento de prioridades para las acciones.

7 Conclusiones y Trabajo Futuro

Hemos presentado el Modelo Básico de Ejecución que permite animar modelos conceptuales especificados en OASIS. Aunque aún falta considerar otros aspectos de la expresividad de OASIS tales como eventos compartidos, restricciones de integridad y transacciones.

El modelo de ejecución planteado ha permitido obtener implementaciones acordes a la semántica del lenguaje OASIS. Se está trabajando en aspectos de formalización de la relación entre una especificación y el prototipo obtenido basándose en modelo de ejecución planteado y en la semántica del lenguaje declarativo usado para la implementación.

El modelo de ejecución propuesto coincide en muchos aspectos con el modelo de computación de ACTORS [Agha 86] lo cual abre un importante camino para su planteamiento desde esa perspectiva.
Desde el punto de vista práctico, el paso siguiente es lograr una implementación robusta del modelo de ejecución e integrarla en la herramienta CASE que ha sido construida para OO-Method. Por otra parte, el modelo de ejecución debería ser extendido para incorporar el tratamiento de eventos compartidos, restricciones de integridad y transacciones.

Actualmente se está definiendo un entorno de animación basado en el modelo presentado y cuyo propósito es apoyar al analista en la validación de los requisitos del sistema y la verificación de consistencia de la especificación.

Referencias

[Agha 86] Agha G.A., *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[Dubo 91] Dubois E., Hagelstein J., Rifaut A., *A formal language for the requirements engineering of computer systems*. In André Thayse, editor, From natural language processing to logic for expert systems, chapter 6, Wiley, 1991.

[Dubo 94a] Dubois E., Du Bois Ph., Petit M., *ALBERT: An Agent-Oriented Language for Building and Eliciting Requirements for Real-Time Systems* in Proc. of the 27th HICSS, J.F. Nunamacker and R.H. Sprague (eds), IEEE Computer Society Press, Vol. 4, pp. 713-722, January 1994.

[Dubo 94b] Dubois E., Du Bois P., Dubru F., *Animating Formal Requirements Specifications of Cooperative Informa-tion Systems* in Proc. of the Second International Conference on Cooperative Information Systems, Toronto (Canada), May 1994.

[Hart 97] Hartel P., Denker G., Kowsari M., Krone M., Erich H.D., *Information Systems Modelling with TROLL Formal Methods at work*, Information Systems Vol.22, No.2/3, pp.79-99, 1997.

[Hart 94] Hartmann T., Saake G., Jungclaus R., Hartel P., Kusch J., *Revised Version of*

Modelling Language TROLL, Informatik-Bericht 94-03, TU Braunschweig, 1994.

[Herz 95] Herzig R., Gogolla M., *An Animator for the Object Specification Language TROLL light*, in Proc. Colloq. on Object–Orientation in Databases and Software Engineering, Montreal, 1995.

[Lete 96] Letelier P., Sánchez P., Ramos I., *Conceptos Básicos de Especificaciones OASIS implementados utilizando Programación Lógica Concurrente*, Informe Técnico, DSIC, Universidad Politécnica de Valencia, 1996.

[Lete 97a] Letelier P., Sánchez P., Ramos I., *Animación de Modelos Conceptuales en un Entorno Concurrente*, Informe Técnico, DSIC, Universidad Politécnica de Valencia, 1997.

[Lete 97b] Letelier P., Sánchez P., Ramos I., *Guías para la Generación Automática de un Programa Lógico Concurrente que sirve de prototipo para una especificaión OASIS*, Informe Técnico, DSIC, Universidad Politécnica de Valencia, 1997.

[Past 92] Pastor O. *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo orientado a Objetos*, Tesis doctoral dirigida por Isidro Ramos, 1992, DSIC Universidad Politécnica de Valencia, 1992.

[Past 95] Pastor O., Ramos I., *OASIS versión 2 (2.2) : A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.

[Past 96] Pastor O., Barberá J.M., Merseguer J., Romero J., Insfrán E., *The CASE OO-METHOD graphic environment description*. Tech. Report, ITI-DT-96.

[Past 97] Pastor O., Insfrán E., Pelechano V., Romero J., Merseguer J., *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods*. CAiSE97. Barcelona Spain, June 1997.

[Pele 95] Pelechano V., Pastor O., *Case OO-Method : Un entorno de producción automática de software orientado a objetos*, Proc.of CIL-95, Barcelona, 1995.

[Sánc 97] Sánchez P., Letelier P., Ramos I., *Constructs for Prototyping Information Systems using Object Petri Nets*, IEEE System Man and Cybernetics, Orlando, October 1997. (accepted)

[Wier 91a] Wieringa R.J., *A Conceptual Model Specification Language (CMSL Version 2)*, Technical Report IR-248, Vrije Universiteit, Amsterdam, 1991.

[Wier 91b] Wieringa R.J., *A Formalization of Objects using Equational Dynamic Logic*, 2nd Conference on Deductive and OO Databases, pags 431-452, Springer 1991, Lecture Notes in Computer Science 566.

Animation of System Specifications using Concurrent Logic Programming*

Patricio Letelier Pedro Sánchez Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera, s/n, 46071 Valencia - Spain
☎ 3877350; Fax: 3877357
email {letelier | ppalma | iramos} @dsic.upv.es

ABSTRACT

OASIS is a Language for the Specification of Object Oriented Conceptual Models. In OASIS the conceptual schema of an information system is represented as a society of concurrent objects interacting with one another. Animation of such models in order to validate the Specification of Information Systems is an interesting topic. We have developed a basic Execution Model for OASIS Specifications. The Model includes features such as the representation of attributes, events, derivations, preconditions, evaluations and triggers. On the other hand, Concurrent Logic Programming is a suitable paradigm for distributed computation because it allows a natural representation of concurrence. Using Concurrent Logic Programming we are able to animate OASIS Specifications according to the OASIS Execution Model. In this work we show how the object-oriented (OO) concepts of an OASIS Specification are represented directly using Concurrent Logic Programming concepts. We give an example of a bank account codified in Parlog language and we show parts of the program that animates its corresponding OASIS specification. A CASE tool supporting the OASIS Model has been implemented and our aim is to add a module for animation of specifications.

1 Introduction

OASIS [9,10] is a Formal Language for the Conceptual Specification of Information Systems. Nowadays there is a growing interest in formal approaches to model information systems. CMSL [20] and TROLL [6] are languages near to OASIS that address the system specification in a similar way. OO-Method [12] is the methodological framework based on the OASIS model.

In order to validate the system requirements it is useful to animate the system specification by means of a prototype automatically generated. Experiments have been carried out using Petri Nets [15] and Concurrent Logic Programming (CLP) [7] as semantics domains for OASIS Specifications. These efforts have led to the establishment of a basic Execution Model [8]. This model can be used in to animate OASIS Specifications implemented over concurrent programming environments.

The aim of this paper is to present the mapping between OASIS concepts and clauses in CLP. Experiments have been carried out with Parlog [2] and Kl1 [3] as Concurrent Programming Languages, which have got similar features. We will concentrate on Parlog to show some examples because the code is simplier.

2 Basic Concepts of OASIS

In the object model of OASIS, an ***object*** is an observable process encapsulating structure and behaviour. An object is an operational unit that has static and dynamic features. Each object has an internal and unique identifier (*Oid*) that allows identifying it during its whole existence.

Object behaviour is characterised by the actions received, those which the object is able to carry out, and by the actions sent, those which it is able to request. Actions are received and sent by interaction mechanisms between objects.

The object properties are represented by ***attributes***. Thus, the ***state*** of the object in a given moment is determined by the value of its attributes. The attributes of the object are classified as follows: ***constant*** if its value does not change during the life of the object, ***variable***

* This work was funded in part by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC94-0557.

Letelier, P. Sánchez, P., Ramos. I. Animation of Systems Specifications Using Concurrent Logic Programming. Proc. of the Symposium in Logical Approaches to Agent and Modelling and Design (ESSLLI´97). 1997.

if its value can change as a consequence of the occurrence of events. Furthermore, if the value of an attribute is obtained from the value of other attributes, the attribute is said ***derived***.

An ***event*** is the abstraction of a change of state. The state can change after the occurrence of an event that modifies the values of some attributes. An event does not have duration and it occurs in an instant of time. Each object possesses a fixed set of events that could affect it during its existence. The state of the object in a given instant will depend on the events occurred in its life until the moment during which is observed.

Two kinds of events can be distinguished: ***private*** and ***shared***. Private events participate in the life of objects of only one class, they appear only in the signature of one class or in several classes belonging to the same hierarchy of specialisation. Shared events are part of the life of objects of more than one class. These events appear in the signatures of all the classes that share them. An event will only be relevant in the life of an object if the state of the object is suitable for that. ***Preconditions*** are formulas and they should be satisfied in order to allow the occurrence of an event.

The objects are not isolated. In OASIS the interaction between objects is modelled in two forms: by ***events sharing*** and by ***triggers***. Triggers introduce activity in the society of objects, allowing an object to act as an agent whenever some conditions are satisfied in its state.

The values of attributes must follow some rules. These rules are named ***integrity constraints***. They are defined by formulas in some kind of logic. They should be satisfied in all the states in the life of the object (static constraints) or among certain sequences of states (dynamic constraints).

A group of events that operates as a unit of execution of greater granularity is called ***transactions***. They follow the two basic principles for transactions: all or nothing, and not observation of intermediate states.

It is possible to construct ***processes*** using events and transactions as atomic actions, they allow specifying the correct sequences of actions in the life of the objects.

A ***template*** is a set of common properties that are shared by several objects. A ***class*** is defined as the group of objects that share the same template. An individual object is called a ***class instance***.

The Society of Objects of our object-oriented system is built from primitive classes, elementary classes and complex classes. Objects that have only one state constitute ***the primitive classes*** or domains and always exist. In general, they undertake the domain of the abstract data types. The ***elementary classes*** are built from the primitive classes. In the template, attributes, events, transactions, preconditions, integrity constraints, triggers and process description are defined and they characterize the objects of the class.

The ***complex classes*** are defined from the elementary ones by the application of operators between classes. The operators that allow the construction of complex classes are ***aggregation***, ***association***, ***specialisation*** and ***parallel composition***. These operators are orthogonal, that is, they are independent and they can be combined.

OASIS is a formal specification language that allows the definition of conceptual schemas according to the object model that has been presented. Here is an example of an account in a bank system.

```
class account
var N:nat.
identification
   by_id:(id).
constant_attributes
   id:nat;
   name:string.
variable_attributes
   balance:int;
   number:nat (0).
derived_attributes
   good_balance:bool.
private_events
   open new;
   close destroy;
   deposit(N);
   withdraw(N);
   pay_commission.
valuation
   [deposit(N)] balance=balance+N and
                number=number+1;
   [withdraw(N)] balance=balance-N and
                 number=number+1;
   [self:pay_commission] balance=balance-100 and
                         number=0.
derivation
   good_balance={balance>100000}.
preconditions
   withdraw(N) if balance>=N;
   close if balance=0.
triggers
   self::pay_commission if number>10 and
                           good_balance=false.
```

```
processes
  ACCOUNT  =  open.ACCOUNT0;
  ACCOUNT0 =  close +
              deposit(N).ACCOUNT0 +
              withdraw(N).ACCOUNT0 +
              pay_commission.ACCOUNT0.
end_class
```

This concrete syntax is formalized in dynamic logic [21] and it has been used in [10].

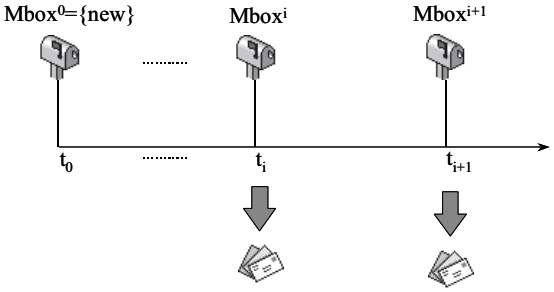


Figure 1: Object life cycle

The life cycle of an object is shown in Figure 1, according to the previous concepts. Each object has a mailbox. The change of state does not have duration and occurs at a precise instant of time, when the actions received in the mailbox are executed. To **attend** an action means to extract it from the *Mbox*. The attended actions can be **admitted** or **rejected** depending on whether they satisfy or not their preconditions. Thus, $Accept^i$ will be the set of admitted actions at the instant t_i, then $Accept^i \subseteq Mbox^i$.

Considering $Exec^i$ as the set of actions executed when the state is $State^i$, then $Exec^i \subseteq Accept^i$. Therefore, an object will change from the state $State^i$ to the $State^{i+1}$ by the **execution** of one or more actions included in $Exec^i$. Intra-object concurrence is allowed. The constraint is that the set of executed actions is not in conflict[1] [8].

Requested actions received by an object are called ***service actions***. In the instant t_i , if an object must send an action to a determined server(s), there will be an associated action in the *Mbox* to remind it about this obligation. Those reminding actions are called ***trigger actions***. It is to say, for each action that the object must trigger whenever it reaches the state $State^i$ it will exist a trigger action in the mailbox at the instant t_i. Thus, the server perspective of an object is integrated with its client perspective in a homogeneous form of operation. This means, the object must serve service actions as well as trigger actions.

Service actions in $Mbox^i$ are represented by $Mbox^i_{services}$. Trigger actions in $Mbox^i$ are represented by $Mbox^i_{triggers}$. Therefore, it is always true that:

$$Mbox^i = Mbox^i_{triggers} \cup Mbox^i_{services}$$

At each t_i $Mbox^i_{triggers} \subseteq Exec^i$, because all the trigger actions have to be executed. Thus, $Exec^i$ can additionally contain a subset of accepted service actions that are in conflict neither with one another nor with the trigger actions. The contents of the mailbox must be updated in each t_i:

(a) Trigger actions or service actions at t_{i-1} will have been extracted.
(b) All the new actions received until t_i must be included.
(c) The actions accepted and not executed in previous states (because they were in conflict) must be stored again.

All these ideas have been considered in order to define a basic execution model including the main features of OASIS specifications. This model allows animating an OASIS specification accurately.

3 From OASIS towards Agent Specification

OASIS is an OO specification language. The concept of object in OASIS can be seen as a specialization of the concept of agent. That means there are other interesting features of agent modelling that are not covered in the current OASIS expressiveness. However many basic aspects and modelling constructs are common or similar. Some key agent features incorporated in OASIS objects and considered in its basic Execution Model implemented in PLC are preconditions, valuations and triggers.

The actions received by the agent are executed only if the agent current state satisfies its precondition axioms. Thus, preconditions represent permissions for the action execution.

The execution of an action can change the agent state modifying the value of its attributes. These changes are specified by the axioms defined in the valuation section.

[1] Basically, the actions in $Exec^i$ have to affect different subsets of attributes.

Agents (OASIS client objects) can activate events triggering actions towards server objects. These calling relationships are established in the trigger section of the class. Whenever the condition of a trigger is satisfied, the agent is obliged to send the action defined by the corresponding trigger axiom.

A limitation in the current version of OASIS regarding agent capabilities is that the state of an agent would be represented only by the value of its attributes. A more versatile agent would be able to change its axioms as well, for instance to change its preconditions, valuations and triggers. A way to incorporate in OASIS objects this capability is shown in [1].

We think it is possible to obtain a sound extension of OASIS including features of agents that are not covered currently. A good example of this approach, going from OO formal modelling to agent specification, is shown in [13].

Hence, the basic Execution Model developed for OASIS specifications and the equivalences established with PLC can be extrapoled to animation of agent specifications considering the limitations mentioned.

4 CLP & OASIS

The concurrent logic languages rise as an attempt to improve the efficiency of logic languages by means of the exploitation of stream AND parallelism [16]. Besides, they are high level programming languages and are very convenient for parallel and distributed systems.

A concurrent logic program is a set of Horn Clauses with Guards.

$$H \leftarrow G_1,...,G_n : B_1,..,B_m \quad n,m \geq 0$$

A goal has the following form:

$$\leftarrow M_1,...,M_k \quad k > 0$$

The M_i are evaluated in parallel (AND parallelism), using the clauses of the program for their reduction.

For each M_i the clauses that can reduce it are examined in parallel (OR parallelism), selecting only one and backtracking is not allowed.

The criterion of selection is that the M_i unifies with the head of the clause, and the conjunction of guards $G_1,...,G_n$ will be satisfied (evaluated also in AND parallelism, if they exist). If more than one clause could be candidate, then a sequential search can be established (in textual order from top to bottom) separating those clauses by ";"

The integration of Concurrent Logic Programming and the OO model have generated many researches and corresponding proposals. We are interested in modelling objects as perpetual processes. This approach is inspired by the works by Shapiro and Takeuchi [17], in which an object is implemented as a tail-recursive process that passes the update state components of the object as arguments in the recursive call. The identity of the object is the name of an input stream argument of the process into which all messages for the object are merged. The works in this direction are principally based on making OO extensions to concurrent logic languages. Within this approach some proposals are: Polka [5] and L2||O2 [11] based on Parlog, and A'UM [19] based on GHC [18].

The main motivation in the proposals of integration between CLP and the OO model has been to provide OO qualities for a concurrent logic language in order to take advantage of the benefits of OO methods for systems development. Thus, the result has been a Logic and OO Programming Language. However, in the research frame of OASIS, our aim is to use Concurrent Logic Programming as a tool for:

a) Experimenting with OO concepts defined in OASIS. As it will be shown, many OO concepts have a direct and natural representation in Concurrent Logic Programming.
b) Providing a declarative and operational semantics for OASIS from the perspective of Concurrent Logic Programming and according to the equivalences settled down.
c) Providing a concurrent implementation support that allows the animation of OASIS specifications

5 Objects and Classes in CLP

Now we will sketch the essential features that allow to consider an OASIS specification as a CLP program. A more complete description can be found in [7].

In CLP a society of objects can be seen (at run time) as a goal to solve, where each object is a

subgoal. Each object is evaluated using AND parallelism (inter object concurrence).

$\leftarrow object_1(In_1,State_1),...,object_k(In_k,State_k) \qquad k > 0$

In_i is a channel for input messages, mixture of the messages received from itself and from other objects. $State_i$ is a list of pairs *(Attribute,value)* for each attribute of the object.

In general, creation (and destruction) of objects at run time is required. Subgoals that have the capacity of generating in their reduction a new instance of an object should exist. These subgoals correspond in natural form to the notion of class. Thus, classes are implemented as other objects of the society in the same way as in [4]. A class has an attribute called Population, which is a list of pairs (Mbox,Key) for each class instance. Mbox is the identifier of the input channel of an instance, Key is the list of constant attributes that allow referring to the object.

Example: The class accounts in our bank system (at run time) is represented in CLP as a goal. Classes are goals at the beginning of the execution. Thus the class account appears as the following subgoal:

$\leftarrow ...,account(Account,Population),...$

Classes will carry out the following operations:

- To create instances in the class, verifying the non-existence and updating the variable Population.
- To answer messages asking for instance existence in the class.
- To distribute messages to instances in the class.

5.1 Object Behaviour

Each object attempts to reduce itself using the clauses that represent its object specification. Each clause recognizes one action and is able to reduce the object to a set of subgoals. Here are an object goal and clauses with which this object goal could be reduced. Considering $p,q,s,t \geq 0$ and $r > 0$:

$\leftarrow ..., object_i(In_i,State_i), ...$

$object_{i1}([action_{i1}|In_i],State_i) \leftarrow$
$G_{i11},...,G_{i1p}$:
$B_{i11},...,B_{i1t}$.
...

$object_{ir}([action_{ir}|In_i],State_i) \leftarrow$
$G_{ir1},...,G_{irq}$:
$B_{ir1},...,B_{irs}$.

Object Creation

The classes are in the initial goal. We obtain the instances by means of reduction of its class when the action new occurs. The creation of an object implies that the following subgoals are generated:

```
create_channel(Oid,Mbox), object(Mbox,State)
```

A variable **Mbox** associated to the term Oid is created. This logical variable can obtain a value by means of partial instantiation. That behaviour simulates a channel receiving actions. Whoever knows the value of Oid could write in this variable.

Example: the clause to which the class account reduces when the action new occurs (action with the open[2] event). In the body of that clause there is the creation of a new object (goal) account.

```
account([action(Client,Server,open(Id,Name,Balance,
                    Number))|Rest],Population) <-
    create_channel(Oid,Mbox),
    derivate(Good_balance,Balance),
    o_account(Mbox,[(id,Id),(name,Name),
            (balance,Balance), (number,Number),
            (good_balance,Good_balance)]),
    account(Rest,[(Oid,Id)|Population]).
```

derivate(Good_balance,Balance) is a predicate that implements the derivation of attribute Good_balance based on the value of the attribute balance. When the event open(100,john,0,0,false) is received the goal account becomes:

```
account([open(100,john,0,0,false)|Rest],[Population])
```

This goal is reduced in the following two subgoals:

```
account(Rest,[(Mbox,100)|Population]),
o_account(Mbox,[(id,100),(name,john),(balance,0)
                    (number,0),(good_balance,false)]).
```

That is, a new object account has been created.

Change of State

We say that an object is implemented as a perpetual process, because among the subgoals in which an object is reduced the same object appears. This produces the effect of continuity in the object life. Whenever the object goal is thrown as subgoal in the reduction, some of its attributes may be modified. Thus a change of state due to the occurrence of the associate event is represented. The effect "to execute action" is

[2] An event of creation implicitly has as arguments the constants and variable attributes of the object.

obtained in the reduction when the new input channel is used as the original one without considering the last executed action.

The formulas that define the change of state when the event is executed are subgoals that assign new values to attributes of the object.

Example: The execution of the action associated with the event deposit(10000) sent to the recently created object is represented as the reduction of the goal:

```
o_account(Mbox,[(id,100),(name,john),(balance,0),
            (number,0),(good_balance,false)])
```

This goal is reduced using the following clause:

```
o_account([action(Client,Server,deposit(N))|Rest],
            [(id,Id),(name,Name),(balance,Balance),
            (number,Number),
            (good_balance,Good_Balance)]) <-
        NBalance is Balance + N,
        NNumber is Number + 1,
        derivate(NGood_balance,NBalance),
        o_account(Rest,[(id,Id),(name,Name),
            (balance,NBalance),(number,NNumber),
            (good_balance,NGood_balance)]).
```

Thus the goal is reduced to

```
o_account(Rest,[(id,100),(name,john),(balance,10000),
                (number,1), (good_balance,false)])
```

That is, the object has changed its state.

Object Destruction

The destruction of an object is obtained by reduction that is carried out by selecting a clause whose body does not contain the same object as subgoal. Hence, the execution of the object ends.

Example: when the object executes the action related to the close event, the following clause is selected:

```
o_account([close|Rest],[(mbox,Mbox)|Att]) <-
    close_channel(Mbox).
```

Preconditions

Event preconditions are incorporated directly as guards. For each precondition for an event (in addition to the clause that recognizes the event) there is one clause that recognizes the "non-action", that is to say, the refusal of the action whose event has a precondition unsatisfied.

Example: Preconditions are implemented as guards in clauses that execute the non-action. Here is the sequence (clauses separated by ";") of clauses that verify these preconditions when the event withdraw(20000) is considered.

```
o_account([action(Client,Server,withdraw(N))|Rest],
            [(id,Id),(name,Name),(balance,Balance),
            (number,Number),
            (good_balance,Good_Balance)]) <-
    not(Balance>=N):
        o_account(Rest,[(id,Id),(name,Name),
                (balance,Balance),
                (number,Number),
                (good_balance,Good_Balance)]);


o_account([action(Client,Server,withdraw(N))|Rest],
            [(id,Id),(name,Name),(balance,Balance),
            (number,Number),
            (good_balance,Good_Balance)]) <-
        NBalance is Balance - N,
        NNumber is Number + 1,
        derivate(NGood_balance,NBalance),
        o_account(Rest,[(id,Id),(name,Name),
                (balance,NBalance),
                (number,NNumber),
                (good_balance,NGood_balance)]).
```

Only the second clause produces a change of state. The second clause however is only used if the first has not satisfied its guard.

5.2 Inter-Object Communication

The communication mechanism offered by CLP is to share variable among subgoals, interpreted as a communication channel among objects. In order to allow that many objects could communicate with a determined object, the channel concept implemented in Parlog was used with the primitives create_channel/2, write_channel/2 and close_channel/1. It is the same as to instantiate a variable by mixing an indetermined number of inputs. Hence, many senders could send actions to an object. There is a unique channel for each object and it is named Mbox.

Actions

Each object is suspended, waiting for actions in its input channel. The trigger of an action from an object to an other is produced by the goal:

```
write_channel(Channel,Action)
```

This sentence will appear in the body of a selected clause in the reduction of a client object (subgoal). Channel is the identifier of the shared variable that constitutes the input channel of actions for the server object.

Until now, we have not made a precise distinction between event and action. Actions

have the following structure:

```
action(Client,Server,Event)
```

Client is a term representing the client object, Server is a term representing the server object and Event is the event itself. Actions must first arrive in the class in order to determine the channel of the server object. This channel identifier will be registered in the state variable Population.

Sending Actions

In [7], examples of implementation of several kinds of communication among processes (1:1, 1:N, N:M) are introduced. Here we will focus on the synchronous or asynchronous nature of the communication.

In CLP, the effect of sending an action from a client toward a server is produced when the client object shares a variable with the client. That variable is the input channel of the server. The sentence used is write_channel/2 and it is one of the subgoals that could appear in the body of a clause in the client object. We are interested in two kinds of communication:

a) **Asynchronous Communication**: the selected clause reduces the object to a sentence write_channel/2 and the object is reduced as a subgoal without waiting for the result of the communication.

b) **Synchronous Communication**: the selected clause reduces the object to a sentence write_channel/2, the action carries a not ground variable (incomplete messages [2]), and the object is forced to wait using the subgoal wait/1. When the server executes the action it assigns a value to the variable and this allows the client to continue its execution.

Triggers and Shared Events

In OASIS the communication among objects is established by means of Triggers and Shared Events.

A) Triggers: they represent asynchronous communication. Hence, triggers are represented by executing write_channel/2 without waiting for answer. Triggers are generated due to a change of state. Thus, in the body of each clause that represents a change of state related with a condition in the trigger, a goal that evaluates the condition and could shoot the trigger will be added.

Example: be the following goal representing an object account reaching a new state:

```
o_account(Mbox,[(id,100),(name,john),(balance,0),
          (number,11),(good_balance,false)])
```

In this case we are sure that the object mailbox contains the action:

```
action(this,this,trigger(this,self,pay_commision)
```

because the object must send such action. The clause that makes the reduction is:

```
o_account([action(this,self,pay_commision)|Rest],
          [(id,Id),(name,Name),(balance,Balance),
          (number,Number),
          (good_balance,Good_Balance)]) <-
        NBalance is Balance - 100,
        NNumber is 0,
        derivate(NGood_balance,NBalance),
        write_channel(Mbox,
                action(this,self,pay_commision),
        o_account(Rest,[(id,Id),(name,Name),
                (balance,NBalance),
                (number,NNumber),
                (good_balance,NGood_balance)]).
```

B) Shared Events: the agent of a shared event has to make a broadcast to all the objects that share that event. The fact that the action is executed in all servers must be controled. A dialogue among the servers is established and the protocol that they follow is similar to Two Phase Commit protocol used in distributed databases. A proposal to implementate this kind of events is presented in [15].

6 Conclusions and Future Work

We have shown how the main features of our object-oriented Conceptual Modelling Language OASIS can be naturally and directly represented in a CLP program. Using the execution model of OASIS as a guide we can guarantee an accurate animation of the OASIS specification.

Althougth OASIS is not an Agent specification language, since the agent and object oriented paradigms share some core features, the equivalences established with PLC in this work are useful in order to animate specifications in both approaches.

We are now working to obtain CLP code automatically from an OASIS specification using the established correspondences. Our final aim is to integrate this work inside a CASE tool for system modelling supporting the OASIS model.

References

[1] Carsí J.A., Ramos I., Bonet B., Jaén F.J., Penadés M.C., *Metaclass Implementation of an Automatic Prototyping Environment based in OASIS*, Technical Report II-DSIC-15/96, Valencia University of Technology, 1996 (in Spanish)..

[2] Conlon T., *Programming in PARLOG*, Addisson-Wesley, 1989.

[3] Chikayama, T., *KLIC User s Manual. (version 2.001),* Institute for New Generation Computer Tecnology, Tokyo JAPAN ,March 1995.

[4] Clark K., Wang T., *D_Parlog++: Object Oriented Programming with Distributed Active Classes*, in "Formal Methods an Object Terchnology", S.J.Goldsack and S.J.H.Kent (Eds), pages 63-85, Springer Verlag 1996.

[5] Davison A., *Polka: A Parlog Object-Oriented Language*, Tesis Doctoral, Depto. Computer Science, Imperial Collegue Londres, 1989.

[6] Hartmann T., Saake G., Jungclaus R., Hartel P., Kusch J., *Revised Version of Modelling Language TROLL*, Informatik-Bericht 94-03, TU Braun-schweig, 1994.

[7] Letelier P., Sánchez P., Ramos I., *Basic Concepts of OASIS Specifications Implemented Using Concurrent Logic Programming*, Technical Report II-DSIC-32/96, Valencia University of Technology, 1996. (in spanish)

[8] Letelier P., Sánchez P., Ramos I., *A Basic Execution Model for OASIS Specifications in a Concurrent Environment*, Technical Report II-DSIC-10/97, Valencia University of Technology, 1997. (in spanish)

[9] Pastor O., *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo orientado a Objetos*, PhD thesis, DSIC, Valencia University of Technology, 1992.

[10] Pastor O., Ramos I.,*OASIS versión 2 (2.2) : A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.

[11] Pimentel E., *L2||O2: Un Lenguaje Lógico Concurrente Orientado a Objetos*, Tesis Doctoral, Facultad de Informática, Universidad de Málaga, Marzo 1993.

[12] Pelechano V., Pastor, O., *Case OO-Method : Un entorno de producción automática de software orientado a objetos*, Proc.of CIL-95, Barcelona,1995.

[13] Saake G., Conrad S., Türker C., *From Object Specification towards Agent Design*, Proc.of the 14th Int.Conf.on Object-Oriented and Entity-Relationship Modeling (ooer'95), Gold Coast, Australia, Pages 329-240, LNCS 1021, Springer-Verlag, December 1995.

[14] Sánchez P., Letelier P., Ramos I., *An Algorithm for to Inplementate Shared Events of OASIS*, Technical Report II-DSIC-31/96, Valencia University of Technology, 1996. (in spanish)

[15] Sánchez P., Letelier P., Ramos I., *Constructs for Prototyping Information Systems using Object Petri Nets*, IEEE System Man and Cybernetics, Orlando, October 1997. (accepted)

[16] Shapiro E., *The Family of Concurrent Logic Programming Languages*, in ACM Computing Surveys. Vol.21, ACM, pp.413-510, 1989.

[17] Shapiro E., Takeuchi A., *Object Oriented Programming in Concurrent Prolog*, en New Generation Computing, Vol.1, pp 25-48, 1983.

[18] Ueda K., *Guarded Horn Clauses*, Technical Report, TR-103, ICOT, 1985.

[19] Yoshida K., Chikayama T., *A'UM: A String Based Concurrent Object-Oriented Language*, en Proc. of the Int.Conf. on FGCS, ICOT, pp.638-649, 1988.

[20] Wieringa R.J., *A Conceptual Model Specification Language (CMSL Version 2)*, Technical Report IR-248, Vrije Universiteit, Amsterdam, 1991.

[21] Wieringa R.J., *A Formalization of Objects using Equational Dynamic Logic*, 2nd Conference on Deductive and OO Databases, pages 431-452, Springer 1991, Lecture Notes in Computer Science 566.

UN INTÉRPRETE DE TRANSACTION FRAME LOGIC

José A. Carsí, José H. Canós, I. Ramos
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera, s/n, 46071 València

RESUMEN

Recientemente ha aparecido una nueva herramienta formal llamada *Transaction Frame Logic* (*TF*) que permite definir sociedades de objetos y dar cuenta de los cambios de estado desde una perspectiva declarativa, simple y formal, y que puede ser usada para una amplia variedad de aplicaciones como puede ser la programación lógica, las bases de datos y la IA. En *TF* el comportamiento, además del estado, es tratado declarativamente. En este artículo se presenta la implementación de un intérprete de *TF* ilustrándose con ejemplos sobre sus usos y aplicaciones. En el apéndice se ofrece un breve manual de usuario del intérprete.

Palabras clave: Desarrollo, bases de datos, orientación a objetos, programación lógica, prototipación automática.

1 INTRODUCCIÓN

En este trabajo se presenta la implementación de un subconjunto de la *Transaction Frame Logic* (abreviada *TF*), recientemente propuesta por Kifer, Lausen, Wu y Bonner ([Kif95], [KLW95], [BK95]), que proporciona un modelo formal para los lenguajes y las bases de datos deductivas y orientadas a objetos.

TF elimina los inconvenientes de algunas lógicas que no dan buena cuenta de la noción de cambio de estado. Las inserciones o borrados realizadas en Prolog con `assert` y `retract` no se deshacen con el *backtracking*, con lo que se pueden alcanzar estados inconsistentes. Además, la *TF* proporciona un marco adecuado en el que describir sociedades de objetos donde los sistemas se describen haciendo uso de los conceptos de objeto, identificador lógico de objeto, jerarquía de herencia y plantilla. Como ventaja añadida, al poder definir la plantilla que deben cumplir todos los objetos que pueblen el sistema y poder manipular de forma declarativa dicho esquema, la *Transaction Frame Logic* proporciona un marco adecuado en el que se pueden hacer evolucionar, no sólo las sociedades de objetos, sino también los esquema que definen las sociedades.

La *TF* tiene una semántica declarativa por teoría de modelos y una teoría de la demostración correcta y completa que la hace especialmente interesante a efectos de prototipación.

De manera similar a los lenguajes imperativos, la *TF* puede usar operadores secuenciales para especificar transacciones, componiendo acciones simples en otras más complejas. Puede dar soporte a un amplio abanico de aplicaciones como, por ejemplo,

Carsí, J.A., Canós, J.H., Ramos, I. Un intérprete de Transaction Frame Logic. Actas de las II Jornadas de Trabajo en Ingeniería del Software. 1997.

bases de datos deductivas, programación lógica o IA en las que el comportamiento, además del estado, es tratado declarativamente.

El intérprete que presentamos implementa identificación de objetos, objetos complejos, herencia, sobreescritura, métodos funcionales y evaluados a conjunto, comprobación de tipos, negación, relaciones, la conjunción serial y las transacciones de estado elementales.

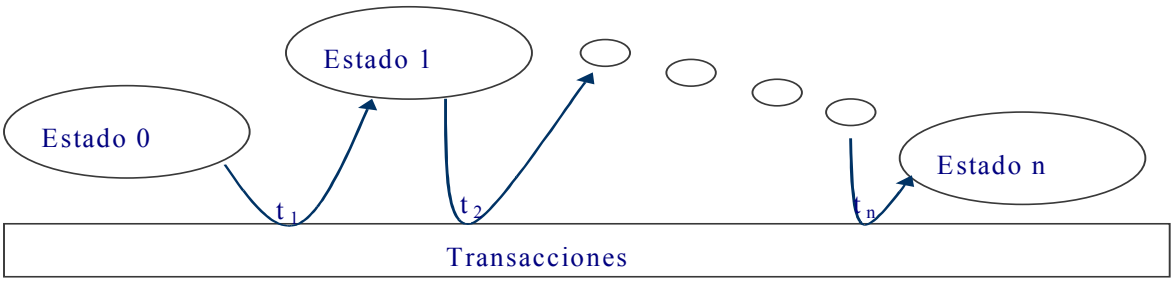
Este artículo está estructurado de la siguiente forma: tras la introducción, en el segundo apartado se presenta brevemente la *Transaction Frame Logic*; en el tercer apartado se presentan las conclusiones y las posibles áreas de aplicación del intérprete y por último en el apéndice A se ofrece un breve manual de usuario.

2 TRANSACTION FRAME LOGIC

La *TF* es una lógica de orden superior resultante de la unión de dos lógicas: la *Frame Logic* (F-logic [KLW95]) y la *Transaction Logic* (T-logic [KB95]). La F-logic es una lógica que nos permite representar adecuadamente el estado de sociedades de objetos pero que no incorpora ningún tratamiento para el cambio de estado. La T-logic es una lógica que nos permite representar los cambios de estado y ejecutar transacciones. De la unión de ambas lógicas se obtiene una lógica que permite representar y hacer evolucionar sociedades de objetos conforme se van ejecutando transacciones.

En *TF* se tiene, por un lado, la *base de datos orientada a objetos* (bdoo) que contiene hechos y reglas en F-logic que reflejan el estado actual de la sociedad de objetos. Por otro lado, existe un *conjunto de transacciones* que manipulan el estado.

El conjunto de transacciones no es susceptible de ser manipulado y permanece inalterado con el paso del tiempo. En cambio, el estado es manipulable y evoluciona a lo largo del tiempo conforme se van ejecutando transacciones (ver 2.2) en el sistema:



En este apartado, vamos a presentar en primer lugar la F-logic y después veremos cómo se enriquece la F-logic añadiéndole ciertas características (que nos permitirán definir transacciones) para convertirla en la *Transaction Frame Logic*.

2.1 Frame Logic

Los componentes básicos con los que se construyen los estados son los *objetos*. El estado de los objetos está caracterizado por el conjunto de valores de las propiedades

estáticas de los objetos, llamadas *métodos*. Los objetos proporcionan el *contexto* en el que se evalúan los métodos. Cada objeto tiene un *identificador* independiente del estado del objeto y que permanece inalterado a lo largo de la vida del objeto. Objetos similares (que comparten las mismas propiedades o plantilla) se agrupan en *clases*. La especificación de una clase es la descripción de las propiedades que una colección de objetos (la extensión de la clase) comparte. Para reflejar la complejidad de los sistemas reales, los objetos y las clases se organizan en *jerarquías de herencia*.

Sintaxis de la Frame Logic

Los id-term son términos de primer orden construidos a partir de funciones y constantes. Los id-term se usan como identificadores lógicos de objetos.

Existen tres tipos de fórmulas atómicas, los *hechos Is_a*, las *F-moléculas* y las *ecuaciones*, que pueden ser combinadas para construir fórmulas más complejas haciendo uso de las conectivas lógicas comunes: $\vee$, $\wedge$, $\neg$, $\leftarrow$.

- *hechos Is_a* de la forma **p:q** o **p::q**
- *F-moléculas* de la forma **p[expresión de método$_1$;...; expresión de método$_n$]**
- *Ecuaciones* de la forma **p = q**

donde p y q son id-terms.[1]

En F-logic se pueden encontrar diferentes tipos de expresiones de métodos que originan F-moléculas las cuales se muestran a continuación. Las primeras cuatro expresiones se llaman *expresiones de datos* (son usadas para definir las propiedades de los objetos y de las clases), y las dos últimas se llaman *expresiones de signatura* (se usan para definir la plantilla de las clases).

1. expresión **escalar no heredable**
   `salario(1997) -> 50000`
2. expresión, **evaluada a conjunto no heredable**
   `hijos(luis, ana) ->> { juan, jose, anabel }`
3. expresión **escalar heredable**
   `salarioPromedio(actual) *-> 200000`
4. expresión, **evaluada a conjunto heredable**
   `mesesLluviosos(1997) *->> { ene, feb, mar }`
5. expresión de **signatura**[2] **escalar**
   `salario(año) => integer`
6. expresión de **signatura evaluada a conjunto**
   `hijos(persona, persona) =>> persona`

donde `1997`, `luis`, `jose`, `ene`, `integer`, `persona` son id-terms que denotan objetos.

Las expresiones de datos pueden ser heredables o no heredables. Las flechas **->** y **->>** denotan expresiones de métodos no heredables y las flechas ***->** y ***->>** denotan expresiones de métodos heredables. Los métodos heredables son heredados como tales en las subclases de las clases en la que estén definidos, y son heredados como métodos no heredables en las instancias de las clases en las que estén definidos.

[1] Se puede hacer uso de *P-moléculas* que son fórmulas atómicas cuya apariencia es la misma que la de los predicados en lógica de primer orden.
[2] Nos permiten definir la plantilla de las clases.

Las expresiones de signatura definen de qué métodos disponen los objetos, el tipo de los argumentos y el tipo de los valores devueltos por los métodos.

Tanto las expresiones de datos como las expresiones de signatura pueden ser escalares o evaluadas a conjunto. Por ejemplo, `jose[salario(1997) -> 50000 ]` indica que, cuando se invoca el método `salario` con argumento `1997` en el objeto `jose`, se devuelve el objeto cuyo identificador es `50000`. Una molécula con una expresión evaluada a conjunto como `familia1[ hijos(luis, ana) ->> { juan, jose, anabel } ]` indica que, cuando se invoca el método `hijos` en el objeto `familia1` con argumentos `luis y ana`, se devuelve un conjunto formado por los identificadores de objetos `juan, jose` y `anabel`.

La semántica declarativa y operacional de la F-logic, así como su corrección y completidud pueden verse en [KLW95].

Ejemplos

Se usan identificadores de objetos lógicos para referenciar a los objetos (**oid**). Un oid (también llamado *id-term*) es un término de primer orden construido a partir de funciones y constantes. Algunos identificadores de objetos son:

```
13, _256, jose32, padre(maria), coche, socio, ...
```

Un oid puede hacer referencia tanto a un objeto como a una clase, lo que permite manipular los datos (objetos) y el esquema (las clases) con el mismo lenguaje.

Se pueden definir **jerarquías de herencia** a través de unas fórmulas llamadas *hechos Is_a*. Se dispone de dos tipos de relaciones de herencia entre objetos: por un lado se dispone de la relación *Is_a* que permite definir una clase como subclase de otra dada; por otro lado se dispone de la relación *Is_instance_of* que permite definir las instancias de las clases:

Is_instance_of:	`jose : piloto`	(jose instancia de piloto)
Is_a:	`piloto :: persona`	(piloto subclase de persona)

Los **atributos** pueden ser vistos como cuasi-funciones[3] (*atr: Oid -> Dom*) que reflejan el estado de los objetos. En el siguiente ejemplo se pueden observar los objetos `jose` y `carmen` con los atributos `nombre`, `salario` e `hijos` en F-logic.

```
jose[ nombre->"José García" ; salario-> 400000 ]
carmen[ hijos->>{luis, ana} ]
```

donde `jose`, `carmen`, `luis` y `ana` son identificadores lógicos de objetos que identifican a personas, `"José García"` es un identificador de un objeto de tipo string, `400000` es un identificador de un objeto integer y `nombre`, `salario` e `hijos` son atributos. El atributo `hijos` es un atributo evaluado a conjunto. Los atributos evaluados a conjunto tienen como co-dominio el superconjunto del dominio (*atr: Oid -> 2^{Dom}*).

Los **métodos** pueden ser vistos como cuasi-funciones que, dado un objeto y unos argumentos, devuelven unos resultados (*mth: Oid x Dom_1 x ... x Dom_n -> Dom*).

[3] Ya que su valor puede cambiar de una invocación a otra para un mismo argumento.

Formalmente, el objeto que proporciona el contexto es el primer argumento de la función y el resto son los argumentos propios del método. En el siguiente ejemplo se puede observar cuál es el `salarioIdeal` de un objeto dado.

```
jose[ salarioIdeal( 1992 ) -> 200 ].
jose[ salarioIdeal( 1993 ) -> 210 ].
jose[ salarioIdeal( 1997 ) -> 400 ].
```

A pesar de que en algunos sistemas orientados a objetos se marca una línea de separación entre atributos y métodos, en F-logic los atributos no son otra cosa que métodos 0-arios[4], esto es, métodos que no poseen argumentos propios.

En F-logic se pueden usar **construcciones de orden superior** que son útiles para, por ejemplo:

- Definir, manipular y razonar sobre las clases y sus instancias, usando el mismo lenguaje.
- Definir clases virtuales (o vistas) y sus atributos a través de reglas deductivas.
- Explorar el esquema y buscar información.

En F-logic, las expresiones de orden superior se obtienen eliminando la distinción entre clases, atributos y objetos. En concreto, no es necesario un *sort* especial para representar las clases. El mismo término sintáctico puede ser usado como identificador de objeto, clase o nombre de método, dependiendo únicamente de la posición sintáctica que ocupe el término en la frase. Por ejemplo, para preguntar a qué clases pertenece el objeto `jose`, escribimos:

```
?- jose : X
```

donde la variable `X` puede tomar valores sobre el conjunto de oids. Debido a la posición sintáctica de la `X`, los únicos oid que pueden ser sustituidos por la variable `X` son los de las clases.

Las posibilidades para buscar información son muy extensas. Se pueden definir reglas como la siguiente:

```
O[ atrs_interesantes_de(C)->>{A} ] <- O[A->V] & V:C.
```

en la que se define el conjunto de atributos interesantes `{A}` para cualquier objeto (`O`) de una clase (`C`) como el conjunto de atributos para los cuales el objeto tiene un valor (`V`) perteneciente a la clase `C`.

[4] Cuasi-funciones 1-arias.

2.2 Transaction Frame Logic

La *TF* proporciona una forma lógica para realizar manipulaciones sobre el estado de una base de datos orientada a objetos, representada por hechos y reglas en F-logic.

Hace uso de los conceptos:

1. **Conjunción serial** (es la forma de decir que después de una transacción[5] se ejecuta otra)

   `A <- B` **`sc`**[6] `C` (para ejecutar la transacción A primero hay que ejecutar B y en el estado alcanzado tras ejecutar B ejecutar C)

2. **Transiciones de estado elementales.** Son operaciones elementales que nos permiten cambiar el estado de los objetos.
   Las transiciones de estado elementales son un conjunto de operaciones que nos permiten manipular cualquier aspecto del estado a través de las acciones de inserción (**ins**) y borrado (**del**). En concreto, nos permiten:
   - insertar/eliminar F-moléculas:
     ```
     ( personas[ edad => integer ] ).ins
     ( jaime[ estado -> soltero ]  ).del
     ```
   - insertar/eliminar hechos Is_a:
     ```
     ( luis : socio ).del
     ( socio_no_fiable :: socio ).ins
     ```

3. **Uniformidad de representación de las consultas y las actualizaciones** (no existe diferencia notacional entre realizar una consulta de un método y ejecutar una transacción).

   ```
   ?- socio1[nombre->Nombre]                  (consulta de un método)
   ?- persona1[casarse(persona2)->LibroFam]   (ejecución de una transacción)
   ```

La T-logic está parametrizada por dos *oráculos*: el oráculo de datos y el oráculo de transiciones. La tarea del oráculo de datos consiste en responder a preguntas como: "¿Es una fórmula cierta en un estado dado?". El oráculo de transiciones responde a cuestiones como: "¿Puedo pasar de un estado a otro usando alguna operación elemental?".

Formalmente, cada oráculo es una función. El oráculo de datos, θ^d, es una función definida sobre estados que devuelve conjuntos de fórmulas ciertas. De manera similar, el oráculo de transiciones, θ^t, es una función entre pares de estados que devuelve conjuntos de transiciones elementales.

Intuitivamente, $\alpha \in \theta^d(D)$, quiere decir que la fórmula α es cierta en el estado D. De forma similar, $\beta \in \theta^t(D_1, D_2)$, quiere decir que la actualización elemental β cambia el estado D_1 al estado D_2.

[5] Una transacción en *TF* es una acción o una secuencia de acciones. Una acción puede ser una consulta o una transición de estado elemental.

[6] serial conjunction

Sintaxis de la TF

Los id-term son términos de primer orden construidos a partir de funciones, variables y constantes. Los id-term se usan como identificadores lógicos de objetos.

Existen cuatro tipos de fórmulas atómicas, los *hechos Is_a*, las *F-moléculas*, las *ecuaciones* y las *transiciones de estado elemental* que pueden ser combinadas para formar fórmulas más complejas haciendo uso de las conectivas lógicas comunes: $\vee$, $\wedge$, $\neg$, $\leftarrow$, y la conunción serial **sc**.

- *hechos Is_a*, *F-moléculas*, y *ecuaciones* son idénticas a las presentadas en el apartado de F-logic.
- *transiciones de estado elemental* de la forma **(** X **).ins (** X **).del**

donde X es una F-regla.

Una transacción es una regla con una fórmula atómica en cabeza y con un cuerpo en el que se usa la conjunción serial como conectiva.

La semántica formal de *TF* está basada en estados y caminos de ejecución. Un estado es un conjunto de fórmulas. Un camino de ejecución es una secuencia finita de estados. Cuando se ejecuta una transacción, la base de datos puede cambiar, pasando por algunos estados intermedios, hasta alcanzar el estado final. En *TF*, el éxito de una ejecución se define sobre los caminos, no sobre los estados. Por ejemplo, `(a).ins sc (b).ins` no es lo mismo que, `(b).ins sc (a).ins`, ya que los caminos de ejecución < D, D+{a}, D+{a,b} > y < D, D+{b}, D+{a,b} > son distintos.

En general, si el camino tiene longitud 1, entonces la transacción es una consulta; si el camino tiene longitud 2, entonces la transacción es una transición elemental y si el camino tiene una longitud superior a 2, entonces es una transacción compuesta. La estructura de caminos distingue entre un estado D y un camino <D> de longitud 1. El estado D representa las fórmulas almacenadas en la base de datos, mientras que el camino <D> representa las fórmulas obtenidas tras la combinación de la base de datos y el conjunto de transacciones.

La semántica declarativa y operacional de la *TF*, así como su corrección y completidud pueden verse en [BK95].

3 CONCLUSIONES

Hasta la fecha únicamente existían implementaciones de la Frame Logic por un lado y de la Transaction Logic (utilizando una representación del estado clausal) por separado[7]. El principal logro del trabajo presentado consiste en haber implementado la integración de ambas lógicas.

[7] Los diferentes prototipos existentes pueden encontrarse en: *http://www.cs.sunysb.edu/~kifer/dood/*

Se han presentado las características de un intérprete de la *TF* que proporciona una herramienta, con un sólido soporte formal, en el que abordar diferentes áreas de aplicación, como pueden ser:

Prototipación Automática: en la actualidad se está trabajando en la traducción automática de una especificación formal en OASIS de un sistema de información a *TF* de forma que se obtenga un prototipo, equivalente a la especificación, que ayude en las tareas de validación de requerimientos.

Bases de Datos: en *TF* se pueden definir esquemas de bases de datos en el que los tópicos siguientes se abordan de una manera natural.

- *Mantenimiento de la consistencia*: problema que surge cuando la actualización de cierta información puede provocar la pérdida de consistencia de la base de datos y son necesarias acciones adicionales para mantener la consistencia.
- *Actualización de vistas*: problema que surge cuando se han definido vistas derivadas y se desea actualizar la información de las mismas.
- *Bases de datos heterogéneas*: problema que surge cuando se tiene una colección de bases de datos que almacena información similar de diferentes formas y es necesaria su actualización.
- *Evolución del esquema*: dada la continua evolución de los sistemas reales, las bases de datos que reflejan dichos sistemas deben evolucionar adaptándose a los nuevos entornos[8].

Recuperación Inteligente de la Información: Al permitir construcciones de orden superior, las posibilidades de interrogación a las bases de datos son muy extensas.

Inteligencia Artificial: En el área de planificación es sencillo representar el conocimiento sobre las acciones que se deben realizar a través de la definición de las acciones y de las estrategias de planificación.

A través del servidor Web del grupo de investigación[9], en breve, se podrá acceder al intérprete de *TF* para su evaluación.

BIBLIOGRAFÍA

[BK95] Bonner A.J., Kifer M., ***Transaction Logic Programming***, Technical Report CSRI-323, disponible vía ftp en: *ftp:/csri.toronto.edu/csri-technical-reports/323/report.ps.Z*, November 1995.

[CRP96] Carsí J.A., Ramos I., Penadés M.C., Pelechano V., ***Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic***, I Jornadas de Trabajo en Ingeniería del Software, Sevilla, Noviembre 1996.

[8] En las I Jornadas de Trabajo en Ingeniería del Software se presentó el artículo [CRP96] en el que se mostraba un posible uso del intérprete de *TF* para definir esquemas conceptuales basados en el lenguaje de especificación formal OASIS y las posibilidades relacionadas con el tópico de evolución del esquema a través de la descripción en *TF* de la metaclase que refleja el modelo de objetos de OASIS.

[9] *http:/www.dsic.upv.es/users/oom/*

[Kif95] Kifer M., ***Deductive and Object Data Languages : A Quest for Integration***, Keynote address at the 4-th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, December 1995 (Springer's LNCS 1013).

[KLW95] Kifer M., Lausen G., Wu J., ***Logical Foundations of Object-Oriented and Frame-Based Languages***, Journal of the Association for Computing Machinery, May 1995.

[MCH91] Manley, John; Cox, A.: Harrison, K.; Syrett, M.; Wells, D., ***KBMS1 a User Manual (Version 3)***, HP-Labs. Internal Report HPL-91-178, 1991.

APÉNDICE A: MANUAL DE USUARIO

En este punto se ofrece un pequeño manual para los futuros usuarios del intérprete, donde se explica la forma de instalar el intérprete, cuáles son los servicios proporcionados, los predicados predefinidos y, además, unos ejemplos de bases de datos y conjuntos de transacciones listos para ser usados.

A.1 Requisitos Técnicos

Las únicas exigencias técnicas del intérprete provienen de la herramienta escogida para la implementación: el sistema gestor de bases de conocimientos KBMS1 ([MCH91]).

Concretamente KBMS1 funciona sobre las máquinas:

- Estaciones de trabajo HP de las series 300 (HP-UX), series 400 (HP-UX), series 700 (HP-UX), series 800 (HP-UX) y series 900 (MPE-XL).
- HP Vectras (OS/2) con un mínimo de 4MB de memoria (son recomendables 8MB).

A.2 Puesta en Marcha

Para poder utilizar el intérprete de *TF*, el usuario debe incorporar ciertas variables a su entorno de trabajo. La manera más fácil de hacerlo es modificando el archivo de configuración del usuario (`.profile` en ksh y otros) incluyendo las variables `KBDIR` (directorio de instalación de KBMS1), `KBPATH` (directorio con archivos relevantes para KBMS1) y modificando la variable `PATH`.

Por ejemplo, suponiendo que KBMS1 se encuentre en el directorio `/usr/local/KBMS1` y el intérprete de *TF* esté instalado en el directorio `$HOME/tflogic`. Se debe añadir al fichero `.profile` en ksh:

```
...
KBDIR = /usr/local/KBMS1
KBPATH = $KBDIR:$HOME/tflogic
PATH = $PATH:$KBPATH
export  KBDIR   KBPATH   PATH
...
```

Tras configurar correctamente las variables de entorno, se puede invocar el intérprete de *TF* tecleando desde el símbolo del sistema el ejecutable `runtf`,

```
_$ runtf
[ Running Transaction Frame Logic ]
```

```
?-
```

con lo que aparece el prompt del intérprete (`?-`), y se está en condiciones de empezar a trabajar con él.

A.2 Utilidades proporcionadas

Las utilidades que proporciona el intérprete las ofrece a través de un objeto especial llamado `sys`. La sintaxis para invocarlas es la siguiente:

sys[end]
Abandona el intérprete.

sys[loadbase(File)]
Carga en el intérprete los hechos y reglas que forman un estado en *TF*.

sys[loadtran(File)]
Carga en el intérprete un conjunto de transacciones que forman un programa en *TF*.

sys[save(File)]
Salva en un fichero los hechos y reglas que forman parte del estado actual.

sys[clear]
Elimina todos los hechos y reglas.

sys[printF]
Muestra por pantalla los hechos y reglas que forman la base de datos.

sys[printT]
Muestra por pantalla las transacciones que se hayan leido.

sys[debug]
Invoca al depurador. A partir de la invocación de este servicio, cada vez que el evaluador intente demostrar cualquier cosa lo mostrará por pantalla, esperando que el usuario pulse una tecla.

sys[nodebug]
Desactiva el depurador.

sys[tran]
Activa el modo de funcionamiento transaccional, en el que intervienen las transacciones, los hechos y las reglas de la base de datos en la interpretación de los objetivos.

sys[notran]
Activa el modo de funcionamiento no transaccional en el que sólo intervienen los hechos y reglas de la base de objetos en la interpretación de los objetivos (no las transacciones).

A.3 Predicados predefinidos

Al haber realizado una implementación específica de las p-moléculas se ha dejado abierta la posibilidad de acceder a todos los predicados predefinidos que proporciona ***kbProlog***[10]. Para una definición completa se puede consultar [MCH91].

Meta-lógicos: `var(Var), nonvar(NonVar), atom(Atom), integer(Int), atomic(Atomic), number(Number), string(String), functor(Term, Functor, Arity), arg(Number, Term, Argument), name(Atom, List), ground(Term)`

Comparación: `X = Y, X \= Y, X == Y, X \== Y, subsumes(X, Y), subsumable(X, Y), X > Y, X < Y, X =< Y, X >= Y, same_value(X, Y), less_than(X, Y), greater_than(X, Y), compare(Op, T1, T2)`

E/S: `get0(C), get(C), put(C), read(T), read_tty(T), write(T), write(nl), writeq(T), pretty_write(T), pretty_writeq(T), read_characters(L), write_characters(L), display(T)`

Aritméticos: `X is Y, sum(N1, N2, X), mult(N1, N2, X), sub(N1, N1, X), div(N1, N2, X), mod(N1, N2, X), power(N1, N2, X), floor(N, X), ceiling(N, X), abs(N, X), round(N, X), exp(N, X), log(N, X), log10(N, X), sqrt(N, X), sin(N, X), con(N, X), tan(N, X), asin(N, X), acos(N, X), atan(N, X), atan2(N, X), rand(X)`

Strings: `string(T, S), stringq(T, S), matches(S1, S2), strlen(S, N), toupper(S, U), tolower(S, L), left_string(S, N, L), right_string(S, N, R), mid_string(S, L, R, M), append_string(P, Q, R), characters(S, L), char(S, N, C), sprinf(S, Format, L), scanf(S, Format, L), scanf(S, Fromat, L, Rest)`

Múltiples Sol.: `setof(Pattern, Goal, Set)`

Miscelánea: `term(T, L), shell(S), gensym(A, NewA), sort(L, SortL), sort_by_argument(L, A, SortL), length(L, N), member(X, L), append(L1, L2, L3).`

A.4 Ejemplos.

En la siguiente figura aparece el ejemplo introductorio de [KLW95], que contiene un posible estado de la base de datos:

```
% Database Facts:

bob[ name -> "Bob"; age -> 40; works -> cs1 ].
cs1[ dname -> "CS"; mngr  -> bob; assistants ->> {john,sally} ].
mary[ name -> "Mary"; highestDegree -> ms; friends ->> {bob,sally};
      works -> cs2 ].
cs2[ dname -> "CS" ].

% General Class Information:

faculty[ boss => faculty;
         boss => manager;        % Typing: function attribute
         age  => midaged;
         highestDegree => degree;
         highestDegree *-> phd;  % Class property: inherited by
                                 % every faculty
        avgSalary -> 5000].
person[ name => string; friends =>> person;
        children =>> child(person) ].
empl[ works => dept; boss => empl; jointWorks(empl) =>> report ].
```

[10] Lenguaje de programación de KBMS1.

```
dept[ assistants =>> (student, empl); mngr => empl ].

% Deductive Rules:

E[boss -> M] <-
       E : empl &
       E[works -> D] &
       D : dept &
       D[mngr -> M] & M:empl.

X[jointWorks(Y) ->> {Z}] <-
       Y : faculty & Y[papers ->> {Z}] &
       X : faculty & X[papers ->> {Z}].
```

```
% Is-a Hierarchy

yuppie::young.
yuppie::midaged.
20:young.
30:yuppie.
40:midaged.
child(person)::person.
student::person.
empl::person.
faculty::empl.
manager::empl.
lecturer::faculty.
child(john)::child(person).
cacm::article.
jacm::article.
article::report.
```

```
% Inst-Hierarchy
alice:child(john).
sally:student.
sally:empl.
john:student.
john:empl.
mary:faculty.
bob:faculty.
bob:manager.
phil:empl.
cs1:dept.
cs2:dept.
string:datatype.
integer:datatype.
"CS":string.
"Mary":string.
"Bob":string.
ms:degree.
phd:degree.
codd70:cacm.
flogic94:jacm.
```

Como se puede ver, el estado está formado por hechos y reglas en F-logic. Una regla tiene una cabeza y un cuerpo. La cabeza es una fórmula atómica y el cuerpo es una secuencia de fórmulas atómicas conectadas por la conjunción clásica (`&`).

La especificación anterior se puede cargar en el intérprete haciendo uso del servicio:

```
?- sys[ loadbase('examples.fl') ]
```

que cargará el estado anterior en la base de datos del intérprete.

Podemos invocar el modo de funcionamiento no transaccional para que ofrezca todas las respuestas posibles cuando se interrogue sobre el estado:

```
?- sys[ notran ]
```

Podemos preguntar cuáles son las clases de las cuales `bob` es instancia:

```
?- bob : C
    bob : empl
    bob : faculty
    bob : manager
    bob : person
```

O bien quiénes son los jefes de los amigos de `mary`:

```
?- mary[ friends ->> F ] & F[boss->Boss]
    mary[ friends ->> bob ] & bob[ boss -> bob ]
```

En la siguiente figura se puede ver la transacción que permite cambiar el lugar de trabajo de los empleados y la transacción que añade un campo a las personas:

```
% Transaction Rules

E[ change_works( D ) ] <-
      E : empl sc
      D : dept sc
      ( E[ works -> _ ] ).del sc
      ( E[ works -> D ] ).ins.

person[ add_field( Name, Type, Default ) ] <-
      ( person[ Name => Type ] ).ins sc
      ( person[ Name *-> Default ] ).ins.
```

Un programa está formado por transacciones *TF*. Una transacción tiene una cabeza y un cuerpo. La cabeza es una fórmula atómica y el cuerpo es una secuencia de transacciones (o consultas sobre el estado) conectadas por la conjunción serial (`sc`).

Cargando el fichero `examples.tl` dentro del conjunto de transacciones:

```
?- sys[ loadtran('examples.tl') ]
```

Se puede disparar la transacción que sigue para cambiar el lugar de trabajo al objeto `mary` (cambiando antes al modo transaccional):

```
?- sys[tran]
?- mary[ change_works( cs1 ) ]
    mary[ change_works( cs1 ) ]
```

Si preguntamos, ahora, qué objetos tienen jefe:

```
?- sys[notran]
?- O[ boss -> B]
    bob[ boss -> bob ]
    mary[ boss -> bob ]
```

Se pueden disparar secuencias de transacciones. Por ejemplo, para cambiar el lugar de trabajo de `bob` e insertar una nueva empleada llamada `anna`:

```
?- sys[tran]
?- bob[ change_works( cs2 ) ] sc (anna:empl).ins
    bob[ change_works( cs2 ) ] sc ( anna : empl ).ins
```

Se puede añadir un nuevo método a la clase `person`:

```
?- person[ add_field( age, integer, 18 ) ]
    person[ add_field( age, integer, 18 ) ]
```

De forma que si preguntamos por la edad de todas las personas obtendremos:

```
?- sys[notran]
?- X:person & X[ age -> Y ]
    alice : person & alice[ age -> 18 ]
    anna : person & anna[ age -> 18 ]
    bob : person & bob[ age -> 40 ]
    john : person & john[ age -> 18 ]
    mary : person & mary[ age -> 18 ]
```

```
phil : person & phil[ age -> 18 ]
sally : person & sally[ age -> 18 ]
```

Por último, para abandonar el intérprete:

```
?- sys[ end ]
```

META-KAOS : UN METAENTORNO DE PROTOTIPACIÓN AUTOMÁTICA

Tolsá, J.M.; Ferrer, J.; Penadés, M.C.; Canós, J.H.; Ramos, I.
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Camino de Vera, s/n 46071 Valencia
Tel: 3877350 ; Fax: 3877357
e_mail { jtolsa | jferrer | mpenades | iramos}@dsic upv.es

RESUMEN

Este documento presenta Meta-KAOS como un metaentorno de prototipación automática que permite producir y utilizar software orientado a objetos, tomando el modelo formal OASIS como marco básico de referencia. Para su construcción se ha utilizado KAOS, un SGBDDAOO[1] a través del cual se permite el prototipado rápido de las especificaciones realizadas en la fase de diseño, y al que se le ha incorporado el concepto de Metaclase. Además se describen cuales son los requisitos técnicos necesarios para utilizar la herramienta así como una sesión de trabajo que ilustra su funcionalidad.

Palabras clave : Desarrollo, Orientación a Objetos, Prototipación Automática, CASE, Metaclase, Sistemas de Información.

1. INTRODUCCIÓN

Una herramienta CASE que permite producir y utilizar software orientado a objetos en un ambiente de programación automática basándose en el concepto de Metaclase[2] fue presentada en (Penadés et al. 1996). Dicho metaentorno parte del modelo objetual subyacente a OASIS (Pastor y Ramos 1995) e introduce una visión en el que *todo*, incluso los usuarios, son objetos que interactúan entre sí.

Desde un punto de vista funcional, podemos describir dicho Metaentorno de Prototipación Automática desde una doble perspectiva: por un lado, es un metaentorno de producción de software orientado a objetos y, por tanto soporta, entre otras, facilidades para la creación de clases; por otro, está basado en el paradigma de programación automática, lo cual significa que permite el prototipado rápido de las instancias de las clases que se creen en dicho metaentorno.

El documento presenta una implantación de dicho Metaentorno de Prototipación Automática. Dado que la Metaclase es la piedra angular de dicho Metaentorno, pasamos a describirla brevemente con el fin de facilitar la comprensión de la funcionalidad de la herramienta

---

[1] *Sistema Gestor de Base de Datos Deductivas Activas y Orientadas al Objeto*

[2] Dicho concepto difiere notablemente del pattern de diseño conocido como '*reflection*' (Buschmann et al. 1996) aunque la terminología utilizada en el artículo es similar

En el modelo objetual propuesto, la noción de metaclase introduce, tal y como se aprecia en la *figura 1*, una jerarquía de tres niveles:

1) En el primero aparece la *metaclase* como único objeto-clase predefinido en nuestro modelo OASIS. El estado de la metaclase es el que caracteriza a dicho modelo.

2) Las instancias de la metaclase, que aparecen en el segundo nivel, son un tipo especial de objetos, a los que llamaremos *clases*, porque serán precisamente ellos los que tendrán entre sus servicios aquellos que caracterizan la noción de clase en la Programación Orientada a Objetos tradicional, es decir, una plantilla, una factoría de objetos, y un almacén de objetos[3].

3) Finalmente, el tercer nivel está compuesto por todos los objetos (instancias de las clases) que poblarán nuestros Sistemas de Información (SI).

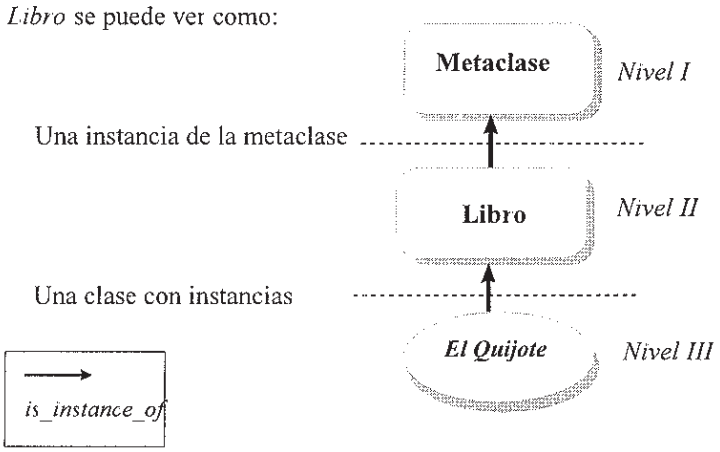


Figura 1 : Jerarquía introducida por la noción de Metaclase.

La *Metaclase* permite definir la estructura y el comportamiento de los objetos *clases*, de la misma forma que utilizamos las clases para definir la estructura y comportamiento de los objetos instancia. Los únicos servicios que el objeto Metaclase ofrece al exterior serán *new* y *destroy*[4], a través de los cuales se crearán las *clases* OASIS. Una vez creadas, serán los propios servicios de las clases los que construyan el tipo que sus instancias compartirán. En cierto modo, se puede considerar la Metaclase como un *"editor de clases"* que responden al modelo OASIS.

Con el fin de poder realizar tales servicios, las clases OASIS comparten un tipo que pasamos a describir a continuación.

[3] Siguiendo esta filosofía podemos definir al objeto Metaclase como una *clase de clases*.
[4] Eventos de *creación* y *destrucción* de objetos.

1.1 Atributos Definidos en las Clases

En la *Tabla 1* se muestran los atributos más importantes que poseen las clases, y que están directamente relacionados con los servicios que éstas ofrecen[5]. De entre todos los que aparecen, los cinco últimos (marcados con *) constituyen parte de lo que podríamos llamar el *estado propio* de las clases, mientras que los restantes definen el tipo que van a compartir todos los objetos que cree cada clase.

Cuando a un objeto-clase se le solicite la creación de una instancia, ésta estará caracterizada por un oid[6] y un tipo. Dicho tipo depende del estado del objeto-clase en el momento de la creación. Esto quiere decir, entre otras cosas, que objetos ejemplares de la misma clase pueden haber sido creados con diferentes tipos, dependiendo del estado del objeto-clase en el momento de la creación de cada uno de ellos, y ello puede que conlleve la necesidad de tener en cuenta detalles similares a los que tratan las técnicas de evolución del esquema en bases de datos OO.

ATRIBUTO	SIGNIFICADO
lista_atr_ctes	lista de atributos constantes de la clase
lista_atr_vbles	lista de atributos variables de la clase
lista_atr_deriv	lista de atributos derivados
lista_deriv	lista de fórmulas para calcular el valor de los atributos derivados
lista_ev_prop	lista de eventos propios de la clase
lista_ev_comp	lista de eventos compartidos
lista_precond	lista de precondiciones
lista_triggers	lista de *triggers*
lista_RI_estatica	lista de restricciones de integridad estáticas
lista_eval	lista de evaluaciones
lista_ident	lista de strings que representan a los atributos de la clase que funcionan como clave
lista_transacciones	lista de transacciones
ev_new *	nombre del evento new de la clase
ev_destroy *	nombre del evento destroy de la clase
actualID *	útimo oid que se ha constuido
poblacion *	población de una clase como colección de oid de las instancias vivas de la clase
*nextOid**	siguiente oid utilizado en el evento crear_instancia

Tabla 1 : Atributos más importantes de las clases OASIS.

1.2 Eventos Definidos en las Clases

Los principales eventos definidos en la signatura de las *clases* OASIS aparecen reseñados en la *Tabla 2*[7]. En ella se aprecia cómo a través de todos ellos se puede construir un tipo. Así, por ejemplo, si a una clase se le solicita el servicio

[5] Una descripción completa de todos los atributos aparece en (Ramos et al. 1995b).
[6] En inglés *Object identifier*
[7] En (Ramos et al. 1995a) y (Ramos et al. 1995b) aparecen detallados todos estos servicios

añadir_atr_cte, el efecto que tendrá sobre su estado será que su atributo *lista_atr_cte* contendrá un elemento más.

Los eventos *new* y *destroy* son los encargados de crear y destruir respectivamente ejemplares de la clase. Éstos, al ser creados, toman como estado inicial una instanciación de la plantilla que define el estado actual de la clase que lo ha creado.

En este documento se esboza cual ha sido el objetivo y la funcionalidad de la herramienta CASE Meta-KAOS: primer prototipo operativo de un Metaentorno de Prototipación Automática OO. Para ello, después de haber presentado las características más importantes de la Metaclase como pieza fundamental del prototipo, en la siguiente sección se ofrece una descripción global de la funcionalidad de Meta-KAOS, además de presentar un esbozo de los detalles de implantación y requisitos técnicos de la herramienta. En la sección 3 se presenta la extensión de OASIS como lenguaje de manipulación de objetos, dando paso a la sección 4 en la que se muestra una sesión de trabajo con Meta-KAOS. Finalmente, en la sección 5 damos paso a las conclusiones extraídas del presente trabajo.

EVENTO	SIGNIFICADO	EVENTO	SIGNIFICADO
iew	crea un ejemplar de la clase	*anyadir_trigger*	añade una expresión de disparo
lestroy	elimina un ejemplar de la clase	*borrar_atr_cte*	elimina un atributo constante
inyadir_atr_cte	añade un atributo constante	*borrar_atr_vble*	elimina un atributo variable
inyadir_atr_vble	añade un atributo variable	*borrar_atr_deriv*	elimina un atributo derivado
inyadir_atr_deriv	añade un atributo derivado	*borrar_deriv_atr*	elimina una fórmula de derivación
inyadir_deriv_atr	añade una fórmula de derivación	*borrar_ev_prop*	elimina un evento propio
inyadir_ev_prop	añade un evento propio	*borrar_ev_comp*	elimina un evento compartido
inyadir_ev_comp	añade un evento compartido	*borrar_transaccion*	elimina una transacción
inyadir_transaccion	añade una transacción	*borrar_precond*	elimina una precondición
inyadir_precond	añade una precondición	*borrar_RI_estatica*	elimina una restricción estática
inyadir_RI_estatica	añade una restricción estática	*borrar_evaluacion*	elimina una fórmula de valuación
inyadir_evaluacion	añade una fórmula de valuación	*borrar_trigger*	elimina una expresión de disparo

Tabla 2 : Eventos principales de las clases OASIS.

2. META-KAOS.

Una vez presentada la Metaclase como piedra angular del Metaentorno de Prototipación Automática (Ramos et al. 1995a),(Ramos et al. 1995b), el siguiente paso es la implantación de dichas ideas. Con este motivo nace Meta-KAOS (Tolsá 1996),(Ferrer 1997), el primer prototipo ejecutable que implanta el concepto de metaclase.

2.1 Funcionalidad

Meta-KAOS es una herramienta CASE que incorpora una visión objetual en todas las fases de la prototipación automática. La noción de metaclase permite no sólo

especificar los sistemas de información, sino también ejecutarlos. Por ello, Meta-KAOS se debe ver desde una doble perspectiva: por una parte, permite la construcción por los usuarios de nuevas clases que modelizan los SI y por otra parte, permite el prototipado rápido de las instancias de esas clases.

Una característica fundamental de la metaclase y, por tanto, de Meta-KAOS, es la uniformidad que proporciona, pues la forma de trabajar del usuario es la misma independientemente de que esté creando nuevas clases o prototipando las existentes.

Actualmente está finalizada la interacción con la herramienta mediante una interfaz textual, lo que nos permite probar su funcionalidad. El siguiente paso a realizar es el dotarlo de una capa gráfica que muestre al usuario de una forma más ergonómica todas estas características de homogeneidad e integración en el modelo OO de la herramienta. Actualmente se está trabajando en este tema y aunque no existe una implantación del mismo, sí que se ha elaborado una propuesta de cómo debe ser ese metaentorno gráfico (Bonet et al. 1996). En este sentido, Meta-KAOS ofrece al usuario una visión del sistema modelado como una sociedad de objetos que interactúan entre ellos, de modo que el propio usuario se sumergirá en esta sociedad como un objeto más. A partir de ese momento podrá crear, evolucionar y validar especificaciones que modelen SI.

2.2 Detalles de la Implantación

Meta-KAOS se puede ver como la evolución del sistema KAOS (Canós 1996), el cual ha sido enriquecido con las características y funcionalidades que ofrece la Metaclase.

KAOS es un SGBDDAOO que implementa el modelo orientado a objetos subyacente a OASIS, y utiliza dicho lenguaje para el diseño, consulta, actualización y programación de aplicaciones.

Meta-KAOS está implementado sobre KBMS1 (Manley et al. 1990) un Sistema Gestor de Bases de Conocimiento desarrollado en los laboratorios de Hewlett-Packard en Bristol. Dicha herramienta está basada en la programación lógica, y permite el almacenamiento, recuperación y actualización de grandes cantidades de *conocimiento*, entendido como conjuntos de cláusulas (hechos y/o reglas), es decir, teorías clausales. Tal conocimiento está expresado en **kbProlog**, una versión del lenguaje Prolog ampliada con la incorporación de *teorías* y primitivas para hacer persistentes las mismas y recuperarlas de manera eficiente y transparente al usuario.

2.3 Requisitos técnicos

El sistema Meta-KAOS se ha implementado sobre estaciones Hewlett Packard 9000 de las series 300 y 700 bajo sistema operativo HP-UX. Este tipo de máquina es la que debe albergar el núcleo del sistema. A continuación se indica cómo se debe realizar la configuración del entorno y la conexión al sistema.

- **Configuración del entorno**

Para poder utilizar Meta-KAOS, se debe modificar el fichero *.profile*, añadiendo la definición de tres variables de entorno. La sintaxis concreta varía de un intérprete de comandos a otro, pero para el caso de *ksh* se corresponde con las líneas que aparecen en la *figura 2*. Una descripción detallada de todos los pasos aparece en el *Manual de usuario* de la herramienta (Tolsá et al. 1997).

```
KBDIR = /usr/local/KBMS1[8]
KBPATH = $KBDIR : $KBDIR/BDOO/sourcesMETA
PATH = $PATH : $KBDIR/BDOO/binMETA
export KBDIR KBPATH PATH
```

Figura 2. Configuración del entorno.

- **Conexión al sistema**

Una vez configurado el entorno, la conexión al sistema Meta-KAOS se realiza desde el intérprete de comandos del sistema operativo a través del mandato:

$ *usar META*

A continuación aparecerá el 'prompt' del sistema, que viene representado por los símbolos "**?.-**" con lo cual ya estaremos en condiciones de trabajar con la herramienta.

3. PREDICADOS PREDEFINIDOS

En este punto veremos dos predicados que nos permitirán realizar operaciones de consulta y actualización sobre los objetos que forman parte de Meta-KAOS. La sintaxis y semántica de dichos predicados se encuentra definida en (Canós 1996), aunque a continuación exponemos sus principales características para favorecer la comprensión de la sesión de trabajo que aparece en la siguiente sección.

3.1 Observaciones

El objetivo de cualquier observación es obtener el valor de un atributo perteneciente a la signatura del objeto. Cuando se realiza una observación sobre un objeto, el estado del mismo no se ve modificado, es decir, ninguno de sus atributos cambia de valor. El predicado que se va a utilizar para las observaciones es ?? / 2. Es un predicado binario, que se utiliza en notación infija. Por tanto, una observación sobre un objeto no será más que un átomo de la forma A ?? B, siendo A el oid del objeto y B un término cuyo functor es el símbolo del atributo que se desea observar. El argumento del término B puede estar instanciado o no.

Por ejemplo, si se desea conocer el valor del atributo *atr* de un objeto *obj*, se realizará la siguiente observación: *obj ??atr(V)*, cuyo resultado será la instanciación de la variable *V* al valor actual del atributo.

[8] Path correspondiente al directorio donde esté instalado KBMS1.

3.2 ¡Error! Marcador no definido.Actualizaciones

Llevar a cabo una actualización sobre un objeto equivale a disparar en el objeto un evento de su interfaz; la ocurrencia del evento provocará un cambio de estado en el objeto. El predicado que se va a utilizar para las actualizaciones es **:: / 2**. Es un predicado binario, que se utiliza en notación infija. Por tanto, una actualización sobre un objeto no será más que un átomo de la forma **A :: B**, siendo **A** el oid del objeto y **B** un término cuyo functor es el símbolo del evento que se desea disparar.

Por ejemplo, si se desea disparar el evento ***ev*** con argumentos ***arg1,...,argn*** en un objeto ***obj***, se puede realizar la siguiente actualización: ***obj : : ev(arg1,...,argn)***

4. EJEMPLO DE SESIÓN DE TRABAJO CON META-KAOS

Hasta ahora se han introducido las propiedades que caracterizan a Meta-KAOS, pero se ha estimado oportuno presentar el aspecto práctico de la herramienta[9].

La exposición se ha estructurado en dos subapartados:

1) En primer lugar se mostrará lo que se ha denomidado "*sesión de edición*", donde se expondrá cómo se pueden crear clases y especificar sus relaciones. Como ejemplo se ilustrará la modelización conceptual de un sistema de información.

2) A continuación se presentará lo que se ha denominado "*sesión de ejecución*", donde se ilustrará la propiedad que tiene Meta-KAOS de animar prototipos generados automáticamente por la herramienta. Como ejemplo se mostrará la animación del EC especificado en el apartado *1)*.

4.1 Sesión de Edición : Modelización de un Esquema Conceptual.

Como se ha comentado Meta-KAOS incorpora el concepto de Metaclase, por tanto presenta los servicios que ofrece dicho objeto visto como **editor de clases OASIS**.

Como ejemplo, se mostrará la generación del EC que representa una versión muy simplificada de una *Biblioteca*, la cual estará formada por las clases *libro* y *socio*, además de su agregación *préstamo*. La *figura 3* muestra el diagrama de configuración de clases de dicha biblioteca.

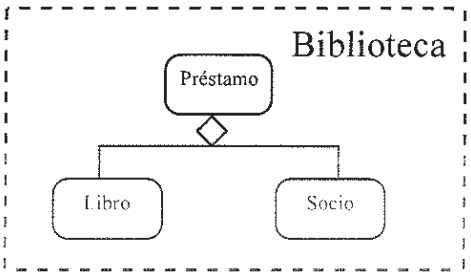


Figura 3: Diagrama de configuración de clases de una *Biblioteca*

[9] Por motivos obvios de espacio, se va a presentar una sesión de trabajo reducida. El manual de usuario de Meta-KAOS (Tolsá et al.1997) presenta un versión completa de dicha sesión de trabajo, además de mostrar, de una manera exhaustiva, la manera de utilizar la Metaclase para modelizar un EC.

Hasta ahora el esquema de trabajo propuesto por KAOS, era la definición de los EC mediante el uso del lenguaje de especificación formal OASIS. Sin embargo, Meta-KAOS propone que las clases que componen un EC sean vistas como instancias de la metaclase. La *figura 4* muestra una representación abstracta del EC *biblioteca*.

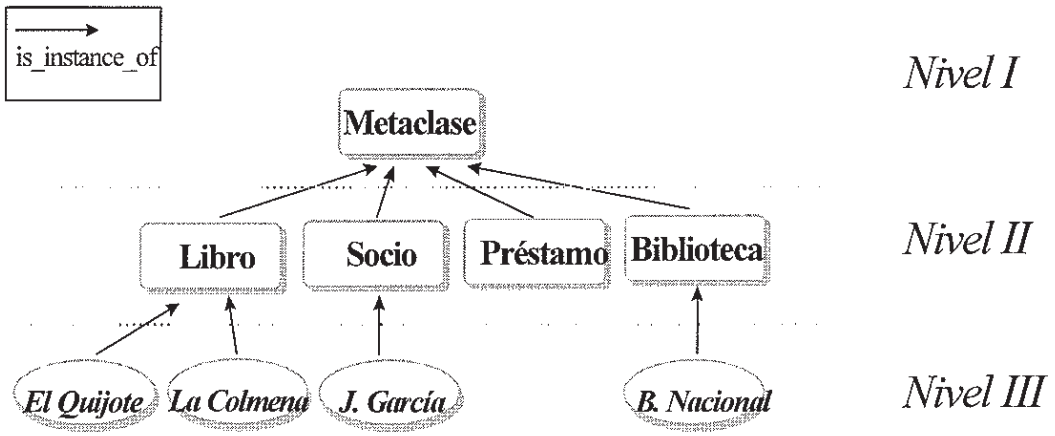


Figura 4 : Representación abstracta del EC biblioteca

Como se aprecia en la *figura 4*, la población de la metaclase coincide con lo que en Ingeniería del Software se conoce como *repositorio*, es decir, la colección de elementos software de que dispone la herramienta para la construcción de nuevos sistemas por reutilización de componentes. Se ha de incidir en que los propios EC formarán parte de dicho repositorio.

Siguiendo el ejemplo propuesto, el primer paso por parte del usuario es la definición de las clases y las relaciones que componen el EC. La *figura 5* muestra cómo se definiría la clase *libro* cuya especificación OASIS se presenta en la *figura 6*.

DEFINICIÓN DE LA CLASE *'libro'*

- **Creación:**

La clase 'libro' es simple. Por tanto para poder definirla se creará una instancia de la clase 'clase_elemental'. Para ello, se lanzará el evento de creación a dicha clase:

```
?.- clase_elemental::crear_clase(Oid,"libro","elemental","27-Abril-97")
Proved  clase_elemental::crear_clase(clase_elemental00001 [10],"libro","elemental" "27-Abril-97")
```

- new y destroy:

```
? - clase_elemental00001 [11]::anyadir_ev_new("alta_libro")
? - clase_elemental00001::anyadir_ev_destroy("baja_libro")
```

- atributos constantes:

```
? - clase_elemental00001::anyadir_atr_cte( cod_libro   nat )
? - clase_elemental00001::anyadir_atr_cte('titulo' 'string')
?.- clase_elemental00001::anyadir_atr_cte( autor', 'string')
```

[10] El sistema instancia la variable Oid con el valor *clase_elemental00001*, el cual será el Oid asignado a la instancia de la Metaclase de tipo *elemental* denominada *libro* y creada *27-Abril-97*

[11] En el ejemplo se ha utilizado el Oid para referenciar al objeto, sin embargo, el sistema facilita la tarea de identificar los objetos mediante la utilización de alias.

- **restricción de integridad estática:**

```
? - clase_elemental00001::anyadir_RI_estatica("cod_libro(N) & N>1 & N<1000")
```

- **atributos variables:**

```
? - clase_elemental00001::anyadir_atr_vble("disponible","bool")
? - clase_elemental00001::anyadir_evaluacion("disponible","true",["alta_libro",[]],"disponible(tru e)")
? - clase_elemental00001::anyadir_evaluacion("disponible","true",["prestar",[]],"disponible(false)")
? - clase_elemental00001::anyadir_evaluacion("disponible","true",["devolver",[]],"disponible(true)")
```

- **eventos compartidos:**

```
? - clase_elemental00001::anyadir_ev_comp("prestar",[],["socio"])
? - clase_elemental00001::anyadir_ev_comp("devolver",[],["socio"])
```

- **precondiciones:**

```
? - clase_elemental00001::anyadir_precond("baja_libro","disponible(true)")
? - clase_elemental00001::anyadir_precond("prestar","dis ponible(true)")
? - clase_elemental00001::anyadir_precond("devolver","disponible(false)")
```

- **procesos:**

```
? - clase_elemental00001::anyadir_proces("libro ← alta_libro & prestar & libro0.
                                          libro0 ← (prestar or devolver) & libro0 or libro1
                                          libro1 ← baja_libro.")
```

Figura 5 : Eventos lanzados a la Metaclase que permiten definir la clase *"libro"*.

```
class libro
        constant_attributes
                cod_libro(nat);
                titulo(string);
                autor(string) ;
        constraints     N:nat.
            static
            cod_libro(N) & N>1 & N<1000
        variable_attributes
        disponible(bool)
          valuation
            [alta_libro]        disponible(true).
            [prestar] disponible(false).
            [devolver]          disponible(true).
          end_valuation
        private_events
            new     alta_libro;
            destroy baja_libro
        shared_events
            prestar with socio;
            devolver with socio.
        preconditions
            baja_libro <- disponible(true).
            prestar    <- disponible(true).
            devolver   <- disponible(false).
        processes
            libro  <- alta_libro & prestar & libro0.
            libro0 <- (prestar or devolver) &
                      libro0 or libro1
            libro1 <- baja_libro
    end_class
```

Figura 6 : Especificación OASIS de la clase "libro".

Si en estos momentos se realizara una instantánea sobre el estado de la clase *libro*, el valor de sus atributos sería el que aparece en la *figura 7*

```
nombreClase("libro")
lista_atr_ctes( ("cod_libro","nat"), (" titulo","string") (" autor" "string"))
lista_atr_vbles( ("disponible","bool"))
lista_atr_deriv(vacia_L)
lista_deriv(vacia_L)
new("alta_libro")
destroy("baja_libro")
lista_ev_prop(vacia_L)
lista_ev_comp( ("prestar","socio"), (" devolver","socio"))
lista_precond( (baja_libro,disponible(true)),(prestar,disponible(true)),(devolver,disponible(false)))
lista_triggers(vacia_L)
RI_estática(cod_libro(N) & N>1 & N<1000)
lista_eval((alta_libro,true,disponible(true)),(prestar,true,disponible(false)),(devolver,true,disponible
(true)))
actualID(0)
poblacion(vacia_L).
```

Figura 7 : Estado de la clase "*libro*"

La herramienta permite el acceso al estado de las clases. La *figura 8* muestra ejemplos de consulta del estado de la clase *libro* creada anteriormente.

```
?.- clase_elemental00001??lista_atr_ctes(L)
Proved   clase_elemental00001??lista_atr_ctes([["cod_libro","nat"],["titulo" "string"],
                                                ["autor","string"]])
?.- clase_elemental00001??ri_estatica(R)
Proved   clase_elemental00001??ri_estatica("cod_libro(N)  & N>1 & N<1000")
```

Figura 8 : Ejemplos de consulta del estado de la clase "*libro*"

La *figura 5* indica cómo definir cualquier clase simple, por lo tanto no consideramos relevante mostrar la definición de la clase *socio* ya que sería afín a la anterior. Por otra parte sí que es relevante exponer la definición de una clase compleja (*préstamo*). La *figura 9* muestra la definición de dicha clase compuesta *préstamo*.

DEFINICIÓN DE LA CLASE *'préstamo'*

- **Creación:**

La clase 'préstamo' es una agregación. Por tanto para poder definirla se creará una instancia de la clase 'clase_agregacion'. Para tal fin se seguirá el proceso de creación de dicho tipo de clases.

```
clase_elemental::crear_clase(Oid,"préstamo","compleja","27 -Abril-97")
Proved .
clase_elemental : crear_clase(clase_elemental00003,"préstamo",ep(clase_compleja00001),
                                "27-Abril-97")

clase_compleja00001::crear_agregacion(Oid2)
Proved . clase_compleja00001 : crear_agregacion(clase_agregacion00001)
```

- new y destroy:

```
? - clase_agregacion00001::anyadir_ev_new('prestar')
? - clase_agregacion00001::anyadir_ev_destroy('devolver')
```

- clases componentes:

```
? - clase_agregacion00001::anyadir_clase_componente(['libro',false,true,true,false,true  ])
? - clase_agregacion00001::anyadir_clase_componente([' socio' false true true true true ' '])
```

- atributos constantes:

```
? - clase_agregacion00001::anyadir_atr_cte("cod_libro , nat )
? - clase_agregacion00001::anyadir_atr_cte("num_socio' 'nat' )
? - clase_agregacion00001::anyadir_atr_cte ("fecha" 'string')
```

- **atributos variables:**

```
? - clase_agregacion00001::anyadir_atr_vble("dias_prestado","nat")
?.-
clase_agregacion00001::anyadir_evaluacion("dias_prestado","true",["prestar",[]],"dias_prestado(1)
")
? - clase_agregacion00001::anyadir_evaluacion("dias_prestado","true",["devolver",[]],
                                              "dias_prestado(0)")
?.-
clase_agregacion00001::anyadir_evaluacion("dias_prestado","dias_prestado(N1)",["pasa_dia",[]],
                                          "N is N+1 & dias_prestado(N)")
```

- **eventos propios:**

```
? - clase_agregacion00001::anyadir_ev_prop("pasa_dia",[])
```

- **procesos:**

```
? - clase_agregacion00001::anyadir_proces("préstamo ← prestar & préstamo0
                                          préstamo0 ← (pasa_dia & préstamo0) or devolver ")
```

Figura 9 : Eventos lanzados a la Metaclase que permiten definir la clase *préstamo*

Ya que en OASIS el propio EC es una clase más, faltaría definir la clase *biblioteca* como una composición paralela de las clases *libro*, *socio* y *préstamo*. En la *figura 10* se muestra dicha definición.

Con esto concluye lo que se ha denominado *sesión de edición*, en la cual se ha generado un EC que representa una biblioteca. Llegado este punto, el siguiente paso es la validación de los requisitos que se habían definido sobre el SI. Meta-KAOS proporciona al usuario facilidades para realizar dicha validación mediante la animación de los EC creados.

DEFINICIÓN DE LA CLASE *'biblioteca'*

- **Creación**:

La clase '*biblioteca*' es una composición paralela Por tanto, para poder definirla se creará una instancia de la clase 'clase_composicion_paralela' Para ello se seguirá el proceso de creación de dicho tipo de clases:

```
?.- clase_elemental::crear_clase(Oid,"biblioteca","compleja","27 -Abril-97")
Proved .
clase_elemental::crear_clase(clase_elemental00004,"biblioteca" ep(clase_compleja00005),
                                          "27-Abril-97")

?.- clase_compleja00005::crear_comp_paralela(Oid2)
Proved.. clase_compleja00005::crear_comp_paralela(clase_composicion_paralela00001)
```

- clases componentes:

```
? - clase_composicion_paralela00001::anyadir_clase("socio")
? - clase_composicion_paralela00001::anyadir_clase("libro")
? - clase_composicion_paralela00001::anyadir_clase("prestamo")
```

Figura 10 : Eventos lanzados a la Metaclase que permiten definir la clase *"biblioteca"*.

Como se ha comentado, Meta-KAOS contiene elementos que lo dotan de capacidad para la animación de los EC construidos. A continuación se mostrará un ejemplo (*figura 11*) de cómo se puede animar un EC especificado con la herramienta[12].

El usuario puede lanzar cualquier tipo de consulta o actualización a los objetos que forman el EC que se quiere animar. Como ejemplo se usará el EC *biblioteca*, presentado en el apartado anterior.

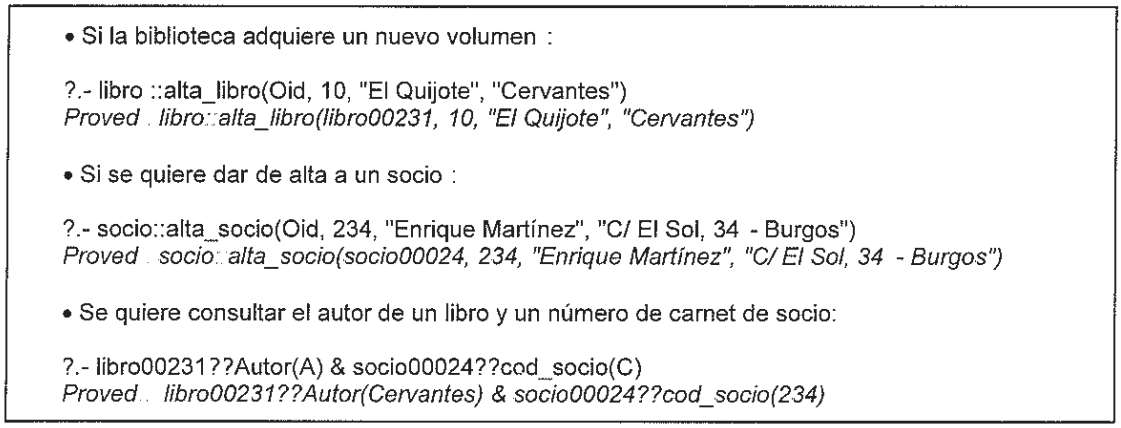


Figura 11 : Ejemplo de animación del EC "*biblioteca*"

5. CONCLUSIONES.

La idea de construir una herramienta CASE que introduzca una visión totalmente objetual y homogénea de lo que sería un Metaentorno de Prototipación Automática, nos ha llevado a la implantación de un primer prototipo operativo denominado Meta-KAOS.

Este prototipo reúne importantes características de las cuales cabría resaltar:

1) La utilización de una única metáfora para toda la interacción hombre-máquina, tanto en la creación (análisis, diseño, programación,...) como en la validación y evolución del software. Dicha metáfora tiene un nombre propio: OASIS. Con esta estrategia se evitan los errores ligados a la falta de uniformidad en el complejo proceso de producción de software.

2) La introducción del concepto de Metaclase, el cual ofrece los servicios necesarios para poder construir clases, que a la vez pueden ser vistas como objetos, los cuales poseen atributos y eventos propios.

3) La especificación del modelo de objetos OASIS usando al propio OASIS como herramienta[13]

Meta-KAOS no pretende ser el último grito en software pero si que intenta, o

[12] Por motivos obvios de espacio, no incluimos más información sobre la animación de EC. Si el lector desea profundizar en las capacidades de animación que posee la herramienta le remitimos a (Carsí et al 1996).

[13] De aquí la aparente proximidad al pattern "reflection"

como mínimo debería hacerlo, romper una pequeña lanza en pro del que debería ser la producción de software a las puertas de una nueva era informática.

BIBLIOGRAFÍA.

Balzer, R.; Cheatman, T.E.; Green, C. 1983. *"Software Technology in the 1990's: Using a New Paradigm"*, IEEE Computer, Nov., pp 39-45

Bonet, B.; Pelechano, V.; Penadés, M.C.; Ramos, I.; Pastor, O.; Carsí, J.A. 1996. *Descripción de la Interfaz Gráfica correspondiente a un Entorno de Prototipación Automática basado en OASIS*, Tech. Report, DSIC, Universidad Politécnica de Valencia.

Buschmann, et al. 1996. *Pattern Oriented Software Architecture*, A System of Patterns, pp 193-221 John Wisley & Sons

Canós, J.H; Penadés, M.C.; Ramos, I.; Pastor, O. 1995. *A Knowledge-Base Architecture for Objects Societies*, Proceedings of DEXA-95, London.

Canós, J.H. 1996. OASIS: *Un lenguaje único para BDOO*, Tesis Doctoral dirigida por I.Ramos. DSIC-UPV (Valencia).

Carsí, J.A.; Penadés, M.C.; Canós, J.H.; Bonet, B. 1996. *Manual de usuario de KAOS*. Tech. Report, DSIC, Universidad Politécnica de Valencia.

Ferrer, J. 1997. *Incorporación del concepto de Metaclase a un SGBDDAOO. Implementación del nivel Clase*, PFC, Facultad de Informática, UPV

Manley J.,Cox A.,Harrison K.,Syrett M.,Wells D. 1990. *KBMS1: A User Manual*, Information System Centre, HP-Labs, Bristol UK.

Pastor, O. 1992. *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos*. Tesis Doctoral dirigida por Isidro Ramos, DSIC-UPV (Valencia).

Pastor, O.; Ramos, I. 1994. *Oasis 2.1.1. A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, Febrero 94 (1ª ed.) Marzo 95 (2ª ed.), Octubre 95 (3ª ed.)

Penadés, M.C.; Ramos, I.; Canós, J.H.; Tolsá, J.M.; Ferrer, J. 1996. *Análisis, Diseño e Implementación OO de un Entorno de Prototipación Automática*, I Jornadas de Trabajo en Ingeniería del Software, Sevilla.

Ramos,I.:Pastor,O.:Cuevas,J.:Devesa,J. 1993. *Objects as Observable Processes*, Proceedings of the 3rd Workshop on the Deductive Approach to Information System Design, Roses, Costa Brava.

Ramos, I.; Pelechano, V.; Penadés, M.C.; Bonet, B.; Canós, J.H.; Pastor, O. 1995. *Análisis y Diseño Orientado a Objetos de un Entorno de Prototipación Automática basado en OASIS*. Tech. Report, DSIC, Universidad Politécnica de Valencia.

Ramos, I.; Pelechano, V.; Penadés, M.C.; Bonet, B.; Canós, J.H.; Pastor, O. 1995. *Especificación en R-OASIS de un Entorno de Prototipación Automática*. Tech. Report. DSIC. Universidad Politécnica de Valencia

Tolsá, J. 1996. *Incorporación del concepto de Metaclase a un SGBDDAOO. Implementación del nivel Meta*. PFC. Facultad de Informática. UPV

Tolsá, J.M.; Ferrer, J.; Penadés, M.C.; Ramos, I. 1996. *Manual de usuario de Meta-KAOS*. Tech. Report, DSIC. Universidad Politécnica de Valencia.

13

MENHIR
Modelos, Entornos y Nuevas Herramientas para la Ingeniería de Requisitos

Grupo Menhir*

Resumen

El proyecto consiste en el diseño y desarrollo de un entorno de producción de software de calidad (correcto, reutilizable y mantenible) que abarque todo el ciclo de vida (desde el análisis al mantenimiento) haciendo uso de los formalismos necesarios para soportar los sucesivos modelos que constituyen el producto software en sus tres aspectos: estado, comportamiento y comunicación. El entorno usará el modelo orientado a objetos (MOO) en todas sus fases, será homogéneo e incorporará intrínsecamente los aspectos de evolución del software, y estará complementado por un soporte metodológico apropiado al modelo. La metodología planteada tendrá un doble apoyo: epistemológico, en la Teoría General de Sistemas, y tecnológico, teniendo en cuenta los métodos industriales más aceptados. Será posible realizar controles de calidad (validación) por prototipación automática (disponibilidad de una semántica operacional que permita la animación de las especificaciones). Los productos generados serán de dos tipos: por una parte, un método específico de especificación de requisitos, incluyendo tanto técnicas diagramáticas como un modelo de proceso adecuado, y por otra, software en entornos de programación visual y de programación orientada a objetos actuales y prototipos en ambientes declarativos que permitan comprobar propiedades del software producido. El entorno ofrecerá una metáfora de trabajo única para producción, validación, reutilización, evolución, y explotación del software soportada sobre mecanismos de interacción intuitivos y uniformes basados en las últimas tendencias de programación visual. Varias empresas de producción de software participan como Entes Promotores/Observadores con el objetivo de generar un prototipo industrializable.

Palabras clave

Ingeniería del Software, Ingeniería de Requisitos, Métodos formales, Modelo Orientado a Objetos, Componentes software, Evolución y reutilización de software

---

* Ver componentes en Tabla 1. Contacto: Prof. Dr. Isidro Ramos Salavert, Departament de Sistemes Informàtics i Computació. Universitat Politècnica de València. Camí de Vera s/n (46071) València. Tel: [illegible]

Ramos, I.,Canós, J.H., Pastor,O.+ investigadores proyecto MENHIR. MENHIR: Modelos, Entornos y Nuevas Herramientas para la Ingeniería de Requisitos. Actas de las III Jornadas sobre Tecnología de Objetos. 1997.

INTRODUCCIÓN

El principal objetivo de la industria del software es obtener productos de calidad con una elevada productividad. La calidad del software es sólo alcanzable usando herramientas precisas —es decir, formales— en su construcción. Dichas herramientas dan contenido a palabras como *corrección*, *validez*, *completitud*, etc., y posibilitan la automatización de tareas. La idea es que el producto software, en sus distintos niveles de desarrollo, sea un producto de precisión y de "alto valor añadido" —como ocurre en otras tecnologías. La productividad, por su parte, está íntimamente relacionada con la disminución de costes de producción y explotación de los productos. Cerca del 80% del coste del software se sitúa en el mantenimiento, y aún partiendo de un proceso de producción altamente formalizado (y, por ende, automatizado) que posibilite controles de calidad a lo largo de dicho proceso (no sólo al final) el mantenimiento evolutivo y perfectivo sigue acaparando la mayor parte de los recursos. Dicho mantenimiento es por otra parte insoslayable dada la dinámica de las "reglas de negocio" que el software modeliza.

El software debe, por lo tanto, ser construido para ser modificado y posteriormente reutilizado sin que ello aporte un deterioro en su calidad, un incremento de costes significativo ni una dificultad superior a la de su producción

Una de las tareas más críticas en Ingeniería del Software, y que tiene una mayor repercusión en cuanto a éxito o fracaso, calidad y productividad del desarrollo de software, es la especificación de requisitos, y con ella el prototipado y modelado conceptual del sistema de información. Esto ha quedado patente en numerosos estudios realizados en el sector y en las numerosas situaciones de fracaso que se han producido; casos conocidos como el famoso del aeropuerto de Denver, el ruinoso contrato entre las FAA americanas e IBM de principios de los noventa, etc., o los informes del "Standish Group" publicados en *IEEE COMPUTER*, Agosto 1995 (Industry Trends Section) sobre la previsión de índices de fracaso de los proyectos software aluden a estas tareas como muy importantes en el desarrollo. Otros estudios recientes sobre desarrollo de software Cliente/Servidor (C/S) y Orientado a Objetos (OO) demuestran que las tareas más críticas siguen estando en el análisis y la especificación de requisitos[1].

Igualmente se observa cómo la mayoría de los métodos de desarrollo presentes en el mercado, y en la literatura especializada, suponen la realización de una cantidad enorme de etapas, tareas, pasos, subetapas, etc., que en la práctica nunca se usan, precisamente por lo tedioso de las mismas. Existe todo un proceso de desarrollo en la construcción de cualquier sistema de información de cierta magnitud y complejidad que debe ser abordado con rigor y de forma sistemática, pero también de manera práctica y productiva: aquí el prototipado —y la ayuda de buenas herramientas de soporte— juegan un papel fundamental (Gary Ford en [Ford 1991] proporciona una visión muy completa de todas las actividades y aspectos implicados).

El origen del problema

Debido a imperativos de mercado y a un importante *gap tecnológico*, los métodos formales, en su proyección industrial, arrastran una larga historia de desencuentros. A lo sumo son tema compartido en una mesa redonda a la que se invita a empresas (como en la conferencia *ICALP*). Los intentos de llevar la industria del software a conferencias de alta tecnología (como *ECOOP*, *DOOD*, *EDBT*, y otras) suelen terminar, en el mejor de los casos, en sesiones totalmente paralelas con públicos que se ignoran.

[1] Patricia J. Douglas George M. Alliger. Robert Goldberg. "Client-Server and Object-Oriented Training" *Computer*. June 1 996 pp 80-84

También es verdad, sin embargo, que herramientas como *lex, yacc, LOTOS, VDM,* lenguajes como *Z , OBJ3* y *AXIS*, el modelo relacional, y otros son productos industriales obtenidos partiendo de planteamientos formales y constituyen la punta de lanza del único futuro posible del software de calidad. Y todo ello a pesar de la magnífica salud de los métodos formales y de la importancia de la industria del software.

En cualquier otra tecnología, nadie cuestiona la construcción de modelos matemáticos (usando ecuaciones diferenciales, teoría de grafos, ...) como el único camino posible para obtener productividad y calidad La industria del software es la triste excepción.

Nuestros grupos de trabajo, partiendo de una sólida actividad en métodos formales (Programación Lógica, Especificaciones Algebraicas, Álgebras de Procesos), Bases de Datos (Deductivas, Activas, Orientadas a Objetos) e Ingeniería del Software (Ingeniería de Requisitos, Especificación OO, Prototipación, etc.) lleva años y proyectos intentando trasladar dichos métodos a la industria. Y ello por una razón fundamental: creemos que la Ingeniería del Software es una ingeniería más, que no debe renunciar al rigor, dadas las consecuencias conocidas que ello tiene (falta de calidad y productividad, que se plasman en la célebre *crisis del software*)

En este momento son numerosos los proyectos europeos relacionados con la Ingeniería de Requisitos y la investigación de nuevos métodos de desarrollo en ambientes OO. Podemos citar entre los más relacionados con nuestra propuesta, *IS-CORE* (Information Systems-COrrectness and REusability), o el proyecto *IDEA* [Ceri 1996], centrado en el Sistema Gestor de Bases de Datos Orientadas a Objetos (SGBDOO) deductivo y activo *CHIMERA*, este último de particular interés tanto por sus objetivos como por las relaciones existentes con algunos miembros de los grupos que lo desarrollan, entre ellos E. Bertino del Politécnico de Milán. También haremos uso de *EUROWARE*, diseñado bajo el programa ESPRIT-III, con los objetivos de facilitar la reutilización, en sentido amplio. El proyecto *ESSI FRUIT* n 21456 es una continuación de los trabajos hechos en *EUROWARE*.

OBJETIVOS DEL PROYECTO COORDINADO

El proyecto que presentamos surge después de una larga experiencia en proyectos de investigación relacionados con la formalización y el desarrollo de software en entornos de *Programación Automática*. Varios de ellos con un contenido principalmente científico (como los proyectos CICYT *SINTESIS, PROTESIS, OASIS, PDR*, y el CYTED *IDEAS*) y otros con una vocación tecnológica, realizados con empresas y trabajando con ellas en la producción de prototipos comerciales de entornos de desarrollo de software (como el proyecto PASO OOAP) o en el asesoramiento técnico en la construcción de nuevas herramientas. El proyecto *MENHIR* aglutina así aspectos teóricos y tecnológicos. La confluencia en el proyecto de grupos de trabajo de cinco universidades españolas (ver Tabla 1), cuyas líneas de trabajo —aunque diversificadas— están en el ámbito de la Ingeniería del Software en general y la Ingeniería de Requisitos en particular, hace particularmente atractivo el proyecto por cuanto "anuncia" debates muy interesantes durante el desarrollo del mismo.

UNIVERSIDAD Y TÍTULO DEL SUBPROYECTO	RESPONSABLE	PARTICIPANTES
Politècnica de València *ESPILL (Evolució del Software, Programació visual Imperativa i Llenguatges Lògics)*	Dr Isidro Ramos Salavert◆	Dr. José Hilario Canós Cerdá Dr. Óscar Pastor López José Ángel Carsí Cubel Patricio Letelier Torres Vicente Pelechano Ferragud María del Carmen Penadés Gramaje Juan Sánchez Díaz Pedro Sánchez Palma
Murcia *Método para la construcción de sistemas de información a partir de especificaciones en OASIS (OM: "OASIS Method").*	Dr Ambrosio Toval Álvarez	Dr Jesús García Molina Dr Javier Zapata* José Luis Fernández Alemán Mariano Flores Gutiérrez Joaquín Nicolás Ros María José Ortín
Sevilla *Técnicas Formales Orientadas a Objetos para la Especificación de Sistemas*	Dr. Miguel Toro Bonilla	Rafael Corchuelo Gil Juan Manuel Cordero Valle Amador Durán Toro Ramón Gómez Labrador Manuel Mejías Risoto José Antonio Pérez Castellanos Miguel Rueda Barranco José Antonio Troyano Jiménez
Granada *HEDES: una herramienta CASE de Especificación, Diseño y Evolución de sistemas software para prototipado de sistemas de información*	Dr José Pareis Llorca	Ana Anaya Morito Patricia Paderweski Rodríguez María José Rodríguez Fortiz Germán Sánchez Mariscal Juan Jesús Torres Carbonell**
Valladolid *Definición y diseño de "mecanos" reutilizables como soporte a la construcción rápida de aplicaciones*	Dr. José Manuel Marqués Corral	Francisco José García Peñalvo*** Carmen Hernández Díez Miguel Ángel Laguna Serrano Esperanza Manso Martínez Jesús Manuel Maudes Raedo**** Juan José Rodríguez Díez

◆ Investigador principal del Proyecto

* Dirección General de Informática. Comunidad Autónoma de Murcia

** Delegación del Gobierno en Telefónica (Madrid)

*** Universidad de Burgos

Tabla 1. Relación de participantes en el proyecto *MENHIR*

Productos a obtener

El proyecto consiste en el diseño y desarrollo de un entorno de producción de software de calidad (correcto, reutilizable y mantenible) que abarque desde la especificación de requisitos al mantenimiento haciendo uso de los formalismos necesarios (lógicos,

algébraicos, categóricos) para soportar los sucesivos modelos que constituyen el producto software en sus tres aspectos: estado, comportamiento y comunicación. Será posible realizar controles de calidad (validación de requisitos funcionales) por *prototipación automática* (garantizada por la disponibilidad de una semántica operacional que permita la animación de las especificaciones). Dicho entorno usará el *modelo orientado a objetos* (MOO) en todas sus fases, será homogéneo e incorporará intrínsecamente los aspectos de *evolución* del software.

Todo proceso de desarrollo en este entorno deberá estar soportado por la aplicación sistemática de una serie de acciones descritas en lo que se ha dado en llamar una *metodología de desarrollo*; en el caso de *MENHIR*, la metodología planteada tendrá un doble apoyo: epistemológico, en la *Teoría General de Sistemas*, y tecnológico, teniendo en cuenta los métodos "industriales" (*OMT, Fusion*, y otros).

Los productos generados serán de varios tipos: software en entornos de *programación visual* actuales (*Delphi, Powerbuilder, Cosmos*, etc.) y aplicaciones de Bases de Datos Orientadas a Objetos, aportando con ello un solución al vacío industrial existente en análisis y diseño en dichos entornos; además de prototipos en ambientes declarativos que permitan comprobar propiedades del software producido. El entorno ofrecerá una *metáfora de trabajo* única para producción, validación, reutilización, evolución, y explotación del software soportada sobre mecanismos de interacción intuitivos y uniformes basados en las últimas tendencias de programación visual sobre entornos Windows y de programación en *X-Window*. Varias empresas de producción de software (*Transtools, Consoft, BASE*, y otras) y de la Administración Autonómica (caso de la Dirección General de Informática de la Comunidad Autónoma de Murcia) participan como *Entes Promotores/Observadores (EPO)* con el objetivo de generar un prototipo industrializable. Además, varios investigadores internacionales de prestigio actuarán como supervisores del trabajo realizado.

Otros objetivos básicos

Además de la obtención de un producto concreto, el interés del proyecto se centra igualmente en otros aspectos de gran relevancia desde nuestro punto de vista:

- la realización de una *transferencia* de saber hacer y resultados a y desde las empresas participantes
- la consolidación de una red de varios grupos de trabajo nacionales en estos temas tan relevantes de Ingeniería del Software
- el último objetivo (en este caso indirecto) es económico: incrementar la calidad y la productividad del sector informático en sus distintos niveles.

OASIS 3.0: UN MODELO FORMAL DE OBJETOS

La definición precisa del modelo de objetos a utilizar en el entorno debe de constituir el primer paso del proyecto. La ausencia de un marco formal adecuado ha producido continuas discusiones y ambigüedades sobre conceptos fundamentales en el Modelo Orientado a Objetos.

La industria, desconcertada, ve sucederse manifiestos (como [Atkinson 1989]) que desde una perspectiva pragmática intentan acotar las propiedades del modelo. Aparecen objetos en programación visual que nada tienen que ver con los objetos de *Smalltalk*, éstos nada que ver con los de *Eiffel*, y así sucesivamente. Se asiste a verdaderas "guerras de religión" en las que gurús del pensamiento y telepredicadores del software introducen una todavía mayor confusión.

La situación es análoga a la producida en el mundo de las bases de datos a principios de los años setenta. Las continuas discusiones entre los partidarios del modelo CODASYL y los del relacional sumieron a la industria en el desconcierto. La memoria histórica nos enseña que sólo aquello bien soportado formalmente (el modelo relacional en este caso) ha sobrevivido, y ello a pesar de la gran cantidad de reuniones de comités internacionales (?), documentos con normas, etc. que se produjeron apoyando el modelo CODASYL.

El modelo de objetos OASIS es similar al modelo propuesto por lenguajes como Troll [Hartman 1994], CMSL [Wieringa 1991], y otros. en ellos, los bloques básicos de construcción de los sistemas de información son los *objetos*. Un objeto es una entidad que encapsula estado y comportamiento. El *estado* está caracterizado por el conjunto de valores de las propiedades estáticas del objeto, llamadas *atributos*. Un objeto tiene un *identificador*, que es independiente del estado del objeto y permanece inalterado a lo largo de la vida del objeto. El valor de los atributos de un objeto puede variar debido a la ocurrencia de *eventos*, que son la abstracción de los cambios de estado.

En nuestro modelo utilizamos la lógica para representar el estado de los objetos; un objeto en un estado dado es un teoría en alguna lógica; fórmulas atómicas caracterizan el valor de los atributos y pueden existir reglas que definan los valores de los *atributos derivados* (atributos cuyo valor depende del valor de otros atributos). Un conjunto de fórmulas representan la forma en que cambian los valores de los atributos debido a la ocurrencia de los eventos. En [Pastor 1994, Canós 1996] se usa la lógica dinámica para representar dichas fórmulas. Otras aproximaciones se podrían seguir usando diferentes lógicas como lógica ecuacional [Toval 1994] la *Transaction Frame Logic* ([Kifer 1996]), la *Linear Logic* ([Ehrich 1996]), etc.

Objetos similares (que comparten las mismas propiedades o tipo) son dinámicamente agrupados en *clases*. La especificación de una clase es la descripción de las propiedades que una colección de objetos (la *extensión* o *población* de la clase) comparten. Las clases proporcionan mecanismos para la creación y destrucción de objetos, así como facilidades para la generación de identificadores de objetos.

Finalmente, el *interfaz* público de un objeto está compuesto por el conjunto de atributos observables y/o eventos que el objeto ofrece a la sociedad. El proceso de exportar propiedades se soporta en OASIS a través de la noción de vista o *interfaz*. La definición de vistas establece un modelo de protección que permite a los diseñadores especificar el aspecto que las clases ofrecen al resto del sistema.

La funcionalidad de las clases es formalizada a través de la noción de *metaobjetos*. Las clases son objetos con unos atributos (consultables) y eventos (que manipulan los atributos). Los atributos contienen fórmulas que reflejan, por un lado, las propiedades que comparten el conjunto de sus instancias, y, por otro, los aspectos de factoría y almacén de objetos. Los eventos proporcionados por estos objetos/clases incluyen la creación y destrucción de objetos, así como, otros que permiten modificar el tipo de la clase reflejado en su estado (incluyendo así la evolución del esquema dentro del propio modelo). Como objetos OASIS, las clases son instancias de lo que llamamos la *metaclase OASIS*; esto es, una clase que tiene como posibles instancias todos los metaobjetos (todas las clases) que pueden componer cualquier esquema conceptual concordante con nuestro modelo ([Canós 1996]).

LOS FORMALISMOS A USAR

Una de las primeras actividades del proyecto consistirá en formalizar el MOO; dada la imposibilidad de establecer cálculo alguno que se derive de la semántica categórica y

que implemente el modelo, procede bajar el nivel de abstracción y elegir los sistemas formales que permitan animar las especificaciones realizadas en los lenguajes formales asociados:

- lenguaje de definición de objetos (*object definition language*, ODL)
- lenguaje de interrogación de objetos (*object query language*, OQL)
- lenguaje de manipulación de objetos (*object manipulation language*, OML)
- lenguaje de evolución del esquema (*object schema language*, OSL)

a efectos de validación por prototipado automático (primer control de calidad), tratamiento de la evolución del esquema conceptual (a efectos de posibilitar la mantenibilidad), recuperación y reutilización de especificaciones y definición de dichos lenguajes. En nuestra aproximación, un lenguaje único, OASIS 3.0, engloba como sublenguajes a los anteriores sin por ello dejar un contexto formal unificado (lógica clausal dinámica en el sistema *KAOS* [Canós 1996], TF lógica en [Carsí 1996]) y manteniéndose en una complejidad razonable.

Dicho lenguaje, soporte de nuestras fases de diseño, sólo será usado en casos excepcionales como herramienta de construcción del diseño o de su modificación. Los lenguajes de especificación visual que soportan el análisis, asistidos metodológicamente en su uso, facilitan la operación del entorno generándose automáticamente a partir de ellos especificaciones textuales en OASIS 3.0.

EL MÉTODO A PROPONER

El CASE es probablemente uno de los más importantes esfuerzos tecnológicos en la ingeniería del software de la última década. Si a este esfuerzo le añadimos la fuerza con la que el paradigma objetual ha entrado en todos los campos de la producción de software (análisis, diseño, implementación, prueba, mantenimiento) podremos comprender el interés industrial y científico que se tiene sobre el CASE OO.

El gran interés comercial y las esperanzas de que esta tecnología basada en el paradigma OO se conviertan en la bala de plata que consiga producir software de calidad en un tiempo récord, ha hecho que surjan una gran cantidad de metodologías OO que proporcionan métodos[2] y herramientas CASE dando soporte a la metodología, con el fin de modelar e implementar "correctamente" sistemas de información. Dentro de estas metodologías las más utilizadas a nivel industrial son las metodologías llamadas convencionales, como las mencionadas anteriormente, desarrolladas a partir de la experiencia adquirida en el desarrollo industrial de software, aunque sin ninguna base formal que garantice la correcta modelización e implementación de los sistemas. Ultimamente se está intentando crear un marco unificado (el *"método unificado"* [Booch 1997]), que proporcione todos los métodos necesarios para modelar y producir software, con la intención de que todo el "mundo" utilice dicha metodología, aunque con esta propuesta se corre el peligro de proporcionar a los usuarios potenciales un excesivo número de métodos, algunos de ellos redundantes e innecesarios para gran cantidad de proyectos y sin una metodología de trabajo bien clara. En el marco de toda esta "actividad metodológica", se sigue no obstante encontrando lagunas. Algunas de las más importantes afectan a la integración del proceso de mantenimiento y evolución dentro de la propia metodología: evolución de esquemas, propagación del cambio, cambio de requisitos, etc. Otras derivan de la falta de integración del proceso de desarrollo en los

[2] respetando los estándares de notación en el mundo metodológico a nivel industrial nos referiremos a *metodología* como el conjunto de métodos que la integran siendo conscientes de que por ese término deberíamos referirnos a la ciencia que estudia los métodos

métodos y herramientas: soporte a las decisiones de diseño, gestión de la configuración, reutilización, etc

Dentro del mundo científico la preocupación por las metodologías de producción de software es evidente, aunque el principal interés a este nivel se centra en el desarrollo de lenguajes de especificación formales OO, con una sólida base matemática. Además, se están desarrollando propuestas metodológicas (*OMTroll, Oblog CASE, OO-Method, OM* y otros), que permiten acercar estos lenguajes de especificación formales al mundo industrial con las ventajas que el uso de éstos proporcionan.

Solucionar cada uno de estos problemas puede ser un largo camino si se hace sin ninguna base formal que soporte la construcción de herramientas CASE. La resolución "ad-hoc" de estos problemas nos puede llevar a la aparición de nuevas lagunas que provoquen en los usuarios de las metodologías y las herramientas que las soportan una continua insatisfación.

Para conseguir un salto cualitativo en la producción del software hace falta un cambio de concepción en la forma de producir software que nos permita abordar de forma "fácil y clara" la resolución de los problemas que van arrastrando las metodologías actuales. El cambio más significativo en el CASE deberá ser un cambio filosófico. Un cambio en la cultura de la ingeniería del software de las organizaciones. El desarrollo del software utilizando herramientas CASE será un proceso dirigido por la especificación en vez de un proceso dirigido por la codificación, y además se pondrá mucho más énfasis en la construcción de sistemas a partir de componentes reutilizables con un chequeo automático y continuo. El ingeniero de software trabajará en entornos interactivos con interfaces de usuario más sofisticadas que las de las herramientas actuales. Los entornos CASE serán sistemas inteligentes que enseñarán y corregirán al analista, y que además serán capaces de adaptarse a sus necesidades. El cambio cultural nos proporcionará unas herramientas CASE que permitirán poner un mayor énfasis en las fases de análisis y diseño, prototipado rápido, generación automática de código, documentación automática, prueba de especificaciones, mantenimiento y reutilización de especificaciones. Así cualquier cambio en el sistema se hará modificando la especificación y regenerando el código. Estas herramientas serán entornos de producción mejorados con una interfaz de usuario a medida (diseñada específicamente para facilitar el trabajo del analista y adaptadas a su forma de trabajo), homogéneos, correctivos y directivos para que, ante cualquier inconsistencia, guíen al analista hacia la solución adecuada y que además intenten educarlo.

En definitiva, debe tratarse de entornos que permitan la generación de sistemas software evolutivos y que, como sistemas software que son, puedan evolucionar en función de las necesidades de los equipos de desarrollo. De este modo el proceso de mantenimiento, el más costoso del ciclo de vida, se transforma en un proceso de adaptación y transformación asistido automáticamente ([Parets 1996a]).

Nuestra contribución se basa en la idea de integrar las metodologías convencionales con los lenguajes de especificación formales para ofrecer entornos de producción que proporcionen más ventajas que las herramientas CASE convencionales (los lenguajes de especificación formales OO ayudan a los diseñadores a detectar y eliminar ambigüedades, y elementos de dudosa utilidad) El trabajo de investigación a desarrollar irá dirigido hacia la concepción de un soporte metodológico basado en el modelo objetual OASIS que nos permita obtener modelos conceptuales orientados a objetos, y que a partir de estos modelos nos permite obtener software de calidad a través de un proceso basado en el paradigma de la programación automática, paradigma que no sólo

debe permitir la realización de sistemas software consistentes y no ambiguos, sino que debe aportar la posibilidad de su transformación en el tiempo. Además diseñaremos e implementaremos entornos de producción de software OO que soporten la propuesta metodológica. Dichos entornos aportarán toda la potencia que el modelo de objetos en el que nos basamos ofrece (evolución del esquema, reutilización de especificaciones, generación de prototipos,...).

La aproximación propuesta ofrecerá entornos gráficos de modelización conceptual OO que nos servirán para recopilar las propiedades relevantes del sistema, siguiendo las pautas ofrecidas por la metodología a proponer. A partir de la información obtenida podremos obtener automáticamente una especificación formal. Esta especificación formal OO constituirá un repositorio del sistema de alto nivel. Mediante la definición de un modelo de ejecución y de una estrategia de traducción entre el lenguaje de especificación y el modelo de ejecución generaremos automáticamente prototipos completos del sistema en el entorno de desarrollo que se quiera (declarativo o imperativo).

No se trata de ofrecer un método aislado más frente a los ya existentes, sino que, aprovechando las investigaciones y experiencias relacionadas, ya desarrolladas por los grupos de investigación participantes, se cuidará la compatibilidad e integración con los métodos de desarrollo OO y estándares "de facto" más prometedores en el mercado internacional (*"Unified Modeling Language -UML-"*, método *OPEN* y los modelos procedentes del *"Object Management Group -OMG-"*) y nacional (actual y nueva versión de *MÉTRICA*, *OO-Method*). El método se particularizará, y probará sobre, al menos, un caso de estudio real, en el ámbito de las BDOO comerciales, ofreciendo guías para su diseño e implementación, a partir de las especificaciones realizadas.

LAS HERRAMIENTAS A USAR

El proyecto combina aspectos de investigación básica con aspectos de desarrollo de prototipos donde se puedan contrastar experimentalmente las propiedades de los modelos elaborados Tratándose de modelos formales, la programación declarativa proporciona ambientes de razonamiento automático que permiten la implementación *a bajo coste* de prototipos basados en los formalismos del modelo (en concreto, lógicas dinámicas, lineales, TF, Especificaciones Algebraicas, etc.). En esta línea, los participantes poseen amplia experiencia en la construcción de entornos de prototipado rápido a partir de especificaciones formales, así como de sistemas de bases de datos deductivas y orientadas a objetos.

Sin embargo, la definición de notaciones gráficas para la fase de análisis, por un lado, y la evolución de la tecnología y la industria del software por otro plantean la necesidad de abordar desarrollos en ambientes diferentes. El hecho de que gran parte del software actual sea producido utilizando lenguajes de programación imperativa (y, en los últimos años, *visual*) supone un reto para el trabajo de desarrollo del grupo, que se traduce en la posibilidad de estudiar la viabilidad de atravesar el *gap* existente entre la Academia y la Industria ofreciendo a los Ingenieros de Software un conjunto de herramientas basadas en modelos formales que permitan validar requisitos en las primeras fases del ciclo de vida, y que además sean capaces de generar a partir de los requisitos, prototipos de aplicaciones en entornos imperativos

El salto al paradigma imperativo hace perder las buenas propiedades de las especificaciones formales; sin embargo, un proceso bien definido de transformación de conceptos de un modelo a otro puede permitir que las aplicaciones generadas sean equivalentes en su mayor parte a las especificaciones de partida. Si a ello unimos la

introducción de representaciones formales del proceso evolutivo en entornos imperativos (lenguajes y bases de datos orientadas a objetos) basados en modelos conceptuales de evolución y clasificación evolutiva, se puede garantizar una mayor adaptabilidad de los productos obtenidos ([Parets 1996b]).

Las herramientas a emplear en el proyecto pertenecen por tanto a ambos ámbitos; de una parte, entornos lógicos declarativos para ser usados como soporte a la prototipación automática, probablemente ocultos por aplicaciones gráficas, construidas usando herramientas industriales, que permitan el uso de notaciones gráficas (*VISUAL-OASIS, OO-METHOD* y otras) para el análisis y la validación de requisitos. Además, dado que se genera un prototipo de aplicación visual, el conocimiento de la filosofía subyacente a los paradigmas RAD (Rapid Aplication Development), programación de interfaces en X-window (como MOTIF), cliente/servidor, programación por componentes, etc. es esencial.

TRANSFERENCIA DE CONOCIMIENTOS Y RESULTADOS

En la medida en que las disquisiciones formales estén orientadas a aclarar conceptos y a hacer que las implementaciones sigan patrones precisos, el matrimonio entre métodos formales y productos de calidad industrial será posible y permitirá que la transferencia de conocimientos a/desde las empresas participantes sea exitosa.

Ese objetivo está en la base de este proyecto. Las EPOs dispondrán como resultado de su participación en el proyecto de un prototipo operativo que les permita aplicar a la resolución de sus problemas cotidianos los principios metodológicos elaborados. Este tipo de herramientas serán concebidas con un claro soporte formal, pero al mismo tiempo con una también decidida vocación de uso industrial, plasmada en la generación de una ilusión de simplicidad real en su uso. Si la herramienta es fácil y cómoda de usar, y mejora la productividad de una manera sustancial, es obvio que será utilizada. El hecho de que en su concepción se empleen nociones definidas de forma precisa y sin ambigüedad, como debe de ser en un contexto de ingeniería, es el que permitirá trabajar en un entorno de trabajo en el que esos objetivos sean realmente alcanzables.

BIBLIOGRAFÍA

[Atkinson 1989] Atkinson, M. et. al., *The Object-Oriented Database Management System Manifesto*, in Proc. of the DOOD-89, Kyoto, Springer-Verlag, 1989.

[Booch 1997] Booch, G. et al., *Unified Modeling Language for Object-Oriented Development*. Documentation Set, Version 1.0, Rational Software Corporation, 1997.

[Canós 1996] Canós, J. H., *OASIS: un lenguaje único para Bases de Datos Orientadas a Objetos*, Tesis Doctoral, Universitat Politècnica de València, 1996.

[Carsí 1996] Carsí, J.A.; Ramos, I.; Penadés, M.C.; Pelechano, V; *Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic*, I Jornadas de Trabajo en Ingeniería del Software, Sevilla, Noviembre 1996

[Ceri 1996] Ceri, S. and P. Fraternali, Designing Database Applications with Objects and Rules (The IDEA Methodology), Addison-Wesley, 1997

[Ehrich 1996] Ehrich, H.D; Object Specification. In Astesiano, E.; Kreowski, H.J. and Krieg-Brückuer, B (ed.), IFIP WG14.3 Book on Algebraic Foundation of Systems Specification, Springer, 1996.

[Ford 1991] Ford, G., 1991 SEI Report on Graduate Software Engineering Education, CMU/SEI-91-TR-2, April, 1991.

[Hartmann 1994] Hartmann, T. et al., *Revised Version of the Conceptual Modeling and Design language TROLL (Version 2.0)*, Informatik-Beritch 94-03, Technische Universität Braunschweig, 1994.

[Kifer 1996] Kifer M.; Deductive and Object Data Languages: A Quest for Integration. Proceedings of the DOOD-96 conference, Springer Verlag, 1996.

[Marques 1994] Marqués, J.M., R Cuesta, and P. de la Fuente, BLC++: A logical browser for C++. in Proceedings of the Conference on Modelling and Simulation, Guasch and Huber, editors, pages 362-366, 1994.

[Parets 1996a] Parets,J, Torres, J.C., Software Maintenance versus Software Evolution: an approach to Software Systems Evolution.1996 IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS 96), Friedrichshafen (Germany), March 1996

[Parets 1996b] Parets,J, Torres, J.C., A Language for Describing Complex-Evolutive Software Systems. in: 'Computer Aided Systems Theory - EUROCAST'95'. Pichler,F.; Moreno-Diaz,R. (Eds.). Lecture Notes in Computer Science. 1030 Springer-Verlag. p. 181-197. 1996.

[Pastor 1994] Pastor, O., OASIS 2.1: An Object Definition Language for Object Oriented Databases, Draft, DSIC, Universitat Politècnica de València, 1994.

[Torres 95] Torres, J., J.A. Troyano, M. Toro. Software Quality through Formal OO Specification. In 4th. International Conference on Software Quality Management, pp. 47-58, Sevilla (España). Abril 1995.

[Toval 1994] A. Toval, Formalización algebraica de un entorno de producción automática de prototipos orientados a objetos, Tesis Doctoral. Departamento de Sistemas Informáticos y Computación Universidad Politécnica de Valencia Mayo 1994.

[Wieringa 1991] Wieringa, R A Formalization of Objects using Equational Dynamic Logic, in Proc of the DOOD 91, LNCS, Springer-Verlag, 1991

KAOS: An Object-Oriented Software Tool for the Objects Definition, Updating, Querying and Programming in an Object-Oriented Environment

Jose H. Canós[1], María D. Lozano[2], Isidro Ramos[1]

[1] Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València -Spain-
{jhcanos, iramos}@dsic.upv.es

[2] Departamento de Informática
Universidad de Castilla-La Mancha -Spain -
mlozano@info-ab.uclm.es

Abstract

The main purpose of this paper is to present KAOS[1], an Object-Oriented Software Tool built as an implementation of the Object-Oriented Model OASIS developed at the Information Systems and Computing Department of the Technical University of Valencia and based on the Object-Oriented Specification Language OASIS[2]. KAOS is a first prototype that offers the functionality of an Object-Oriented, Deductive and Active Database Management System, built over KBMS1, a Knowledge-Base Management System.

1. Introduction.

OASIS [9] is a proposal of an Object-Oriented Model, i.e., a representation of the reality which uses the object notion as starting point, and which is supported by the definition of a formal specification language that allows the rapid prototyping of Conceptual Schemata, within the model of a life-cycle based on automatic programming.

The goal is to build Information Systems prototypes, process which is also supported by an Object-Oriented Methodology called *OO-Method* [8].

Within this environment, KAOS is the implementation of the model mentioned above. It offers the functionality of an Object-Oriented, Deductive and Active Database Management System and has been built with the aim of solving the impedance mismatch existing in the conventional OODBMS's. In [2] is shown how KAOS solves this problem by using a unique language ([3],[4]), R-OASIS, as an extension of the language OASIS mentioned above. Therefore, KAOS uses it as Object Definition Language (ODL), which allows the definition of the Conceptual schemata, as Object Query Language (OQL), which allows the access to the data for prototyping, and finally as Object Management Language (OML), which allows to build DBMS applications using the notion of active objects [4].

2. The KBMS1 System.

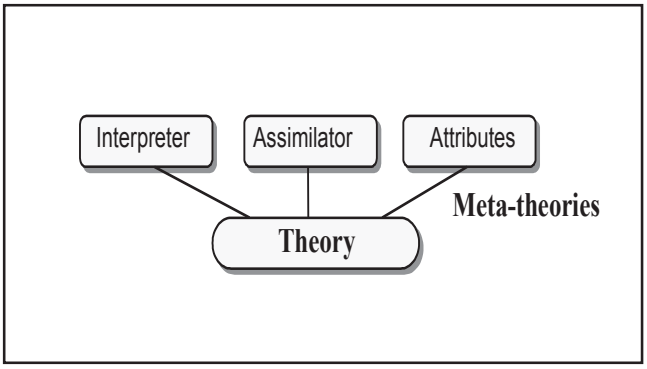
The system used for the implementation of KAOS is KBMS1 [7], a Knowledge Base Management System developed at the Hewlett-Packard laboratories in Bristol; it is a system that extends logic programming environments with persistence and modularity. The tool has been implemented on Hewlett-Packard 9000 workstations, under the HP-UX Operating System, and allowing the client/server architecture. The KBMS1 system allows storing, recovering and updating large quantities of knowledge in form of sets of Horn clauses, written in kbProlog, that is a version of the Prolog language extended with theories and primitives to handle these theories.
Let see briefly the main features of KBMS1 used for the implementation of the tool.

[1] *a Knowledge-base Architecture for Object Societies*
[2] *Open and Active Specification of Information Systems*

Canós, J.H., Lozano, M.D., Ramos, I. KAOS: An Object-Oriented Software Tool for the Objects Definition, Updating, Querying and Programming in an Object-Oriented Environment. Proc. of the IEEE Canadian Conference on Electrical & Computer Engineering.1998.

In first place, a *theory* is an unordered set of procedures, this is, an ordered set of Horn clauses with the same functor and arity of the atom on their heads. They can be handled in execution time and updated in a declarative way. The system theories are a special kind of theories that contain all the other theories defined in the knowledge system. They provide the logical organisation of theories and their meta-theories, and store the system's state at a given moment.

Each theory has three meta-theories associated to it: the *interpreter* theory, the *assimilator* theory and the *attribute* theory. In next figure we can see the internal representation of a kbProlog theory:

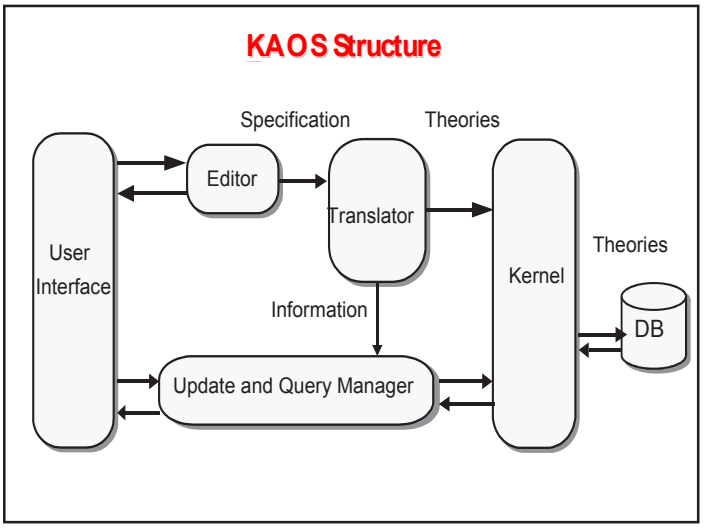


The *interpreter theories* are the query interface for theories, i.e. every query to a theory is handle by its interpreter.

In the same way, *the assimilator theories* are the updating interface for theories. And finally, *the attribute theories* handle meta-information about the theories used by the interpreters or assimilators to carry out special inferences or simply to have a special control over the updates.

3. The KAOS System.

The KAOS system architecture is shown in the following figure:



The system works as follows:

(1) Thanks to a graphical or textual **Editor**, users build the OASIS specification that defines the Conceptual Schema of the Information System being modelled.

(2) A specialised **Translator** builds a model of the defined schema in *kbprolog*, an extension of the Prolog language.

(3) Once the schema has been generated, the **Kernel** builds a *persistent* version of it. As a matter of fact, the persistence is one of the tool's principal features.

(4) From this moment on, we are in conditions to prototype the Information System we have just created. Interactively, it is possible to observe the objects, activate events in them, write and run programs, everything within the same environment.

In order to offer all these services, the system besides consists of:

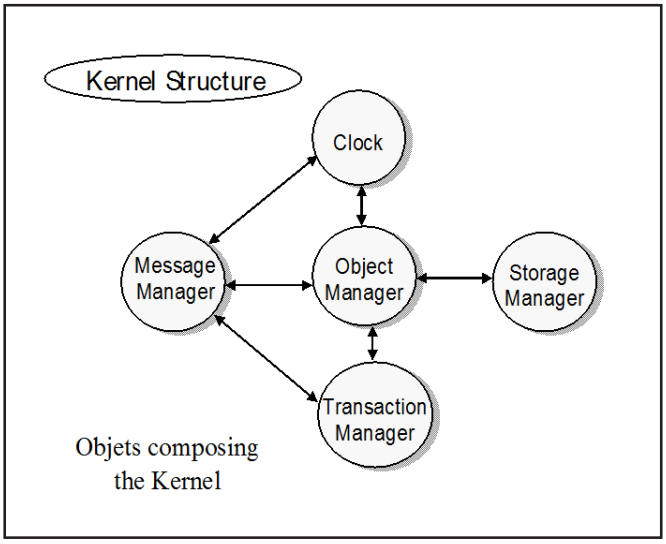
* A **User Interface** which converts the events coming from the keyboard or mouse into character strings which are passed to the Update & Query Manager.
* The **Update & Query Manager** (UQM) receives the character strings generated by the graphic interface and decomposes them into individual messages which are syntactically and semantically checked. Then, the UQM passes these messages to the Kernel of the system. Besides, this module is in charge of running the programs written to manipulate the data. So, the main task of the UQM is to manage the communication between the user and the Database by means of the Kernel.

* The **Kernel of the System** receives the individual messages from the UQM and solves them in case they are queries or executes them in case they are triggers, creates and animates the objects, makes the objects persistent, controls the transactions, the system clock, manages the parallel composition of the classes composing the system, restores the system after crashes, and so on. More information about this can be found in [6].

Now, we describe the main elements of the system:

1) The translator: It converts the OASIS specifications into KbProlog theories. This is, every object is represented by means of a KBMS1 theory. The Object Identification (Oid) is used as the theory name. The queries to the objects will be solved by the interpreter of the theory that represent the object. And the updates will be solved by the assimilator of the theory. And finally, the attribute theory handles the integrity constraints and triggering relationships.

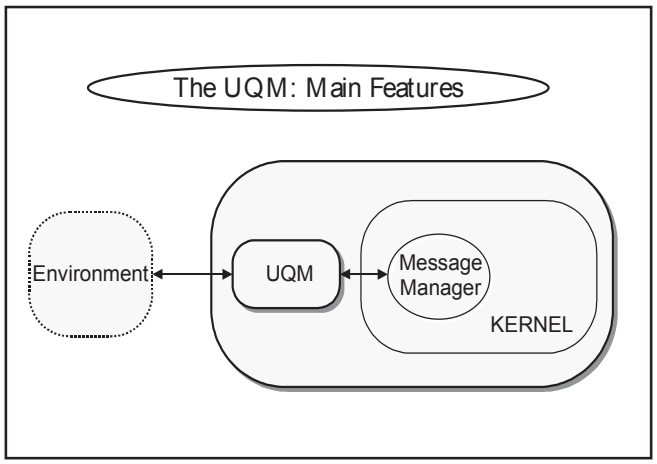
2) The Kernel: It is the lowest level layer of the system. It handles the creation and management of the database. Its structure is the following:



- **The Message Manager** handles the treatment of the received messages, its synchronism, and so on.
- **The Object Manager** deals with everything regarding objects, this is, queries, triggers, memory management, integrity constraints, triggering relationships, and so on.
- **The Storage Manager** solves the accesses to the files composing the database. It is responsible of error recovery and so on.
- **The Transaction Manager** is the responsible of opening, closing or aborting the transactions.
- And finally **the Clock** controls the time and the way of working. We distinguish two ways of working: The current-way in which the time runs forward and where the same queries than updates can be performed. And the past-way in which the time stands in a past position and only the objects' state in that moment can be queried.

3) The Update and Query Manager whose main task is to handle the communication between the user and the database through the kernel. It can be seen as a kernel refinement, as

it is in charge of handling the queries and updates, the new event, the shared events, the specialised objects, the time, transactions and so on. And on the other hand, it acts as an interpreter of the Object Management Language (OML) of the database and it supports the applications defined by the users. It is depicted in the following figure:



4. Conclusions.

We have presented an object-oriented software tool, KAOS, for the object definition, updating, querying and Programming in an object-oriented environment. It has been built over KBMS1 and uses its main features to represent the OASIS objects as theories. As we can see, KAOS is faithful to the OODB Manifesto [1] and regarding other standards like ODMG-93 [5], it follows the proposals exposed there. Therefore, KAOS can be considered a deductive, active and object-oriented database management system.

5. Bibliography.

[1] Atkinson, M. et. al., *The Object-Oriented Database Management System Manifesto*, in Proc. of the DOOD-89, LNCS, Springer-Verlag, 1989.

[2] Canós, J. H., Penadés, M.C., Ramos, I. And Pastor, O., *A Knowledge Base Architecture for Object Societies*, Proc. of the DEXA'95 Workshop, OmniPress, London, 1995.

[3] Canós, J. H. et. al., *A Unified Language for Deductive and Object-Oriented Databases*, Tech. Report, DSIC, Universitat Politècnica de València, 1995.

[4] Canós, J. H., *OASIS: A Unique Language for Object-Oriented Databases*, Ph. D. Thesis, Universitat Politècnica de València, 1996.

[5] Cattell, R. (ed.), *The Object Database Standard: ODMG-93, Release 1.1,* Morgan Kaufmann Publishers, 1994.

[6] Carsí, J. *Diseño e Implementación del Núcleo de un SGBDDAOO Basado en OASIS,* M. Sc. Thesis, Facultat d'Informàtica, UPV, 1994.

[7] Manley, John et. al., *KBMS1: A User Manual (Version 3),* HP-Labs. Internal Report HPL-91-178, 1991.

[8] Pastor, O., Insfrán, E., Pelechano, V., Romero, J., Merseguer, J. *OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods,* Advanced Information System Engineering. CaiSE97. Springer-Verlag. June.1997.

[9] Pastor, O. *OASIS 2.1: An Object Definition Language for Object Oriented Databases,* Draft, DSIC (UPV), 1994.

Especificaciones de proceso para objetos y su representación en Lógica Dinámica*

Patricio Letelier **Pedro Sánchez** **Isidro Ramos**

Departamento Sistemas Informáticos y Computación

Universidad Politécnica de Valencia

Camino de Vera, s/n, 46071 Valencia

email {letelier | ppalma | iramos}@dsic.upv.es

Palabras clave: Ingeniería de requisitos, métodos orientados a objetos, animación automática, álgebra de procesos.

Resumen

Las propuestas más populares para modelado del comportamiento están basadas en los *statecharts* o en alguna de sus derivaciones, por ejemplo los *objectcharts*. OASIS es un modelo OO y formal usado para especificación de requisitos. En OASIS el comportamiento puede ser especificado desde dos perspectivas complementarias: a) considerando el estado actual del objeto se especifican disparos y precondiciones, b) considerando las acciones ya acontecidas se especifican procesos: operaciones y protocolos (en ambos se utiliza álgebra de procesos). Esta capacidad expresiva permite adecuadamente establecer secuencias de acciones que deben ocurrir (operaciones) o secuencias de acciones cuya ocurrencia está permitida (protocolos). En OASIS el formalismo utilizado para disparos y precondiciones es una variante de Lógica Dinámica que incluye los operadores deónticos de permiso, prohibición y obligación. Adicionalmente, la noción de cambio de estado se expresa también en Lógica Dinámica. El interés por animar especificaciones OASIS ha impulsado la búsqueda de un formalismo homogéneo, a partir del cual pueda definirse e implantarse un modelo de ejecución. Este trabajo presenta un procedimiento automatizable para interpretar las especificaciones de proceso como fórmulas en la variante de Lógica Dinámica utilizada.

1 Introducción

La animación automática de especificaciones formales es un aporte interesante en la captación incremental de requisitos y su validación temprana. Sin embargo, los desafíos

*Este trabajo ha sido parcialmente financiado por la "Comisión Interministerial de Ciencia y Tecnología" (CICYT) a través del proyecto con referencia TIC 97-0593-C05-01.

Letelier, P., Sánchez, P., Ramos, I. Especificaciones de proceso para objetos y su representación en Lógica Dinámica. Actas de las III Jornadas de Ingeniería de Software. 1998.

incluidos son considerables. Es necesario disponer de un modelo formal, suficientemente expresivo y factible de ser incorporado en una herramienta que de soporte a la animación de especificaciones de requisitos.

OASIS [10] provee un enfoque OO y formal para especificación de requisitos. OO-Method [11] es la propuesta metodológica basada en OASIS y llevada a la práctica con la asistencia de un entorno CASE. La animación de especificaciones OASIS, fiel a la semántica del modelo e integrada al entorno CASE nos ha impulsado la búsqueda de un formalismo homogéneo, a partir del cual pueda definirse e implantarse un modelo de ejecución [5].

En OASIS, el modelado del comportamiento puede realizarse desde dos perspectivas complementarias en cuanto a expresividad. En cada instante, para un objeto, las acciones permitidas y las acciones obligatorias pueden ser establecidas de la siguiente forma:

- Considerando el estado del objeto en dicho instante. Las precondiciones son fórmulas que permiten la ocurrencia de acciones sólo en ciertos estados del objeto. De forma análoga, los disparos son fórmulas que obligan la ocurrencia de acciones en determinados estados del objeto. El formalismo utilizado en este caso es una variante de Lógica Dinámica que incluye los operadores deónticos de permiso, prohibición y obligación.

- Considerando las acciones ya acontecidas. Una especificación de proceso determina un patrón de comportamiento para el objeto, desde el punto de vista de secuencias de pasos acontecidos, es decir, determina trazas posibles. En OASIS se utilizan dos tipos de especificaciones de proceso: *protocolos* y *operaciones*. Un protocolo establece secuencias permitidas de acciones. Una operación establece secuencias obligatorias de acciones. Las especificaciones de proceso se expresan usando álgebra de procesos. En esta perspectiva, las propuestas más populares están basadas en los *statecharts* [4] o alguna de sus derivaciones, tal como los *objectcharts* [2]. En comparación a los *objectcharts*, la expresividad lograda por las especificaciones de procesos en OASIS es similar.

Este trabajo presenta una procedimiento automatizable para interpretar las especificaciones de proceso como fórmulas en la variante de Lógica Dinámica utilizada. Así, desde una perspectiva global, una especificación de clase OASIS puede ser vista como un conjunto de fórmulas dicha Lógica Dinámica. Estas fórmulas determinan permisos, obligaciones y cambios de estado para los objetos de la clase.

A continuación se presentan algunas definiciones que permiten fijar conceptos esenciales del modelo OO subyacente en OASIS.

Definición 1 ***Plantilla o tipo.*** *Una plantilla de clase está representada por una tupla* $\langle Atributos, Eventos, Fórmulas, Procesos \rangle$.

- $Atributos$ es el alfabeto de atributos. $\forall att \in Atributos$, se tiene la función att : *sort de nombrado* $\rightarrow$ *sort de evaluación.*

- *Eventos* es el alfabeto genérico de eventos. $\forall e \in Eventos$ podemos obtener $\underline{e} = \theta e$ siendo θ una substitución básica de los posibles parámetros del evento.
- *Fórmulas* es un conjunto de fórmulas en diversas lógicas según sea la sección del lenguaje donde se utilizan.
- *Procesos* es el conjunto de especificaciones de proceso.

En la vida de un objeto todo evento es solicitado por un cliente a un determinado servidor, pudiendo, para cada acción, ser el objeto cliente, servidor o ambos.

Definición 2 ***Acción***. *Una acción es una tupla* $\langle Cliente, Servidor, Servicio \rangle$*, que representa tanto la acción asociada a requerir el servicio en el objeto cliente como la acción asociada a proveer el servicio en el objeto servidor.*

Definición 3 ***Servicio***. *Un servicio es un evento o una operación. En el primer caso se trata de un servicio atómico e instantáneo. Una operación es un servicio no atómico y que, en general, tiene duración.*

Para cada clase, asumiremos la existencia implícita de un conjunto A de acciones obtenido a partir de los servicios que los objetos de la clase pueden requerir (cuando son clientes) o proveer (cuando son servidores).

Definición 4 ***Estado del objeto***. *Llamaremos estado del objeto al conjunto de sus atributos evaluados.*

El estado permite caracterizar a un objeto en un instante determinado. El estado de un objeto cambia por la ocurrencia de acciones[1].

Definición 5 ***Conjunto consistente de acciones***. *Un conjunto de acciones* $M \subseteq \underline{A}$ *es consistente si a partir de un estado* σ *dado, el estado* σ' *alcanzado por la ocurrencia de las acciones en* M *es siempre el mismo, independiente del orden en el cual se ejecuten*[2] *las acciones.*

Definición 6 ***Paso***. *Llamaremos paso a un conjunto consistente de acciones que ocurren en un mismo instante de la vida del objeto.*

Definición 7 ***Vida o traza de un objeto***. *Llamaremos vida o traza de un objeto a un prefijo finito de pasos que le acontecen al objeto.*

[1] Utilizaremos la *frame assumption*, es decir, supondremos que si no se especifica que la modificación del valor de un atributo se debe a la ocurrencia de una determinada acción, entonces dicho atributo no es modificado por dicha acción.

[2] Utilizaremos "ejecución de acción" como sinónimo de "ocurrencia de acción" para hacer más intuitiva la presentación. Sin embargo, desde el punto de vista declarativo y formal, "ocurrencia" es el término más adecuado.

2 OASIS expresado en Lógica Dinámica

Globalmente y considerando su expresividad, OASIS está basado en Lógica Deóntica [1]. La Lógica Deóntica es la lógica de obligaciones, prohibiciones y permisos. Siendo $a \in A$, en OASIS se utilizan los siguientes operadores de la Lógica Deóntica:

$O(a)$	obligación de ocurrencia de a.
$F(a)$	prohibición de ocurrencia de a.
$P(a)$	permiso o habilitación de ocurrencia de a.

En [8] la Lógica Deóntica es descrita como una variante de Lógica Dinámica [3]. La definición de los operadores deónticos en Lógica Dinámica es la siguiente:

$O(a) \Leftrightarrow [\neg a]\ false$	"la ocurrencia de a es obligatoria".
$F(a) \Leftrightarrow [a]\ false$	"la ocurrencia de a está prohibida".
$P(a) \Leftrightarrow \neg F(a)$	"a está permitida si y sólo si a no está prohibida".

Donde $\neg a$ representa la no-ocurrencia de la acción a (es decir, que no ocurre nada o que ocurre otra acción distinta de a). En estas fórmulas, $false$ es un átomo tal que no hay un estado que pueda hacerlo verdadero. Esta situación podría representar un estado de violación del sistema. El significado intuitivo de éstas fórmulas es: una acción está prohibida si su ocurrencia conduce a un estado de violación. Una acción es obligatoria si su no-ocurrencia conduce a un estado de violación.

La Lógica Dinámica es un formalismo natural para estudiar de forma simple y directa ciertas aserciones sobre programas. En Lógica Dinámica, la fórmula $[a]\phi$ se interpreta intuitivamente como "después de la ocurrencia de la acción a, ϕ debe satisfacerse", siendo a una acción y ϕ una Fbf de la Lógica de Primer Orden utilizada.

Sea ψ una Fbf que caracteriza el estado del objeto inmediatamente antes de la ocurrencia de la acción a. Utilizaremos los siguientes tipos de fórmulas en Lógica Dinámica:

$\psi \rightarrow [a]false$	"la ocurrencia de a está prohibida en estados que satisfacen ψ".
$\psi \rightarrow [\neg a]false$	"la ocurrencia de a es obligatoria en estados que satisfacen ψ".
$\psi \rightarrow [a]\phi$	"en estados que satisfacen ψ, inmediatamente después de la ocurrencia de la acción a, ϕ debe satisfacerse".

Estas fórmulas constituyen un sublenguaje del lenguaje de Lógica Dinámica propuesto y formalizado en [12]. Para OASIS los tres tipos de fórmulas se corresponden directamente con precondiciones, disparos y evaluaciones, respectivamente. Las especificaciones de proceso expresadas en álgebra de procesos quedan aparentemente fuera de este marco formal. Sin embargo, en la siguiente sección se establece cómo una especificación de proceso puede representarse usando la misma Lógica Dinámica esbozada antes.

3 Especificación de procesos

Existen tres semánticas posibles que pueden ser asociadas a una especificación de proceso en OASIS:

(a) La semántica del lenguaje utilizado para su especificación. En OASIS, el lenguaje utilizado para especificar procesos es un subconjunto de CCS [9], compartiendo así una semántica basada en la noción de sistema de transición etiquetado y todas las propiedades formales establecidas en CCS.

(b) La semántica de permisos, es decir, una especificación de proceso establece secuencias de acciones que pueden ocurrir.

(c) La semántica de obligaciones, es decir, una especificación de proceso establece secuencias de acciones que deben ocurrir.

Dichas secuencias de acciones vienen dadas por la relación de alcanzabilidad entre mundos en la estructura Kripke.

Un proceso en OASIS tendrá siempre la semántica (a). Adicionalmente, asociaremos la semántica (b) ó (c) a un proceso y será declrada de forma explícita. Así, la sección *operations* se utiliza para especificar secuencias obligatorias de acciones y la sección *protocols* para especificar secuencias permitidas de acciones. Un caso particular de proceso de obligación es el que actúa como una transacción, es decir, con la política del "todo o nada". En este caso se puede declarar con el calificativo de *transaction* para el proceso. Asumiremos siempre por defecto que el proceso no actúa como transacción.

Siguiendo el planteamiento y notación propuestos en CCS, la especificación de un proceso R puede ser vista como un sistema de transiciones etiquetado representado por $(\kappa_R, A_R, \{\xrightarrow{a}: a \in A_R\})$, donde:

- κ_R es el conjunto de constantes agente[3] utilizados para definir el proceso R.
- A_R es el conjunto de acciones usadas en la especificación del proceso R, $A_R \subseteq A$, siendo A el conjunto de acciones[4] en la signatura de la clase.
- $\xrightarrow{a} \subseteq \kappa_R \times \kappa_R$ es la relación de transición.

Para cada constante agente $E \in \kappa_R$ se tiene una ecuación de definición $E \stackrel{def}{\equiv} Q$, siendo Q una expresión construida usando los siguientes operadores (siendo $S \in \kappa_R$ y a, a_1 y a_2 acciones pertenecientes al conjunto A_R):

1. $a.S$, es un prefijo.

[3] Un proceso de OASIS en CCS corresponde al concepto de agente. Un agente está representado por una constante agente (su nombre). En la definición de una constante agente se pueden utilizar otras constantes agente, es decir, otros agentes.

[4] Incluye tanto las acciones generadas desde otros objetos (o el mismo objeto) como las acciones generadas por el propio objeto. Es decir, el objeto visto como cliente y servidor.

2. $if\ C\ then\ a.S$, es un condicional, siendo C una expresión *boolean*.

   Extensiones del tipo $if\ C\ then\ a_1.S_1\ else\ a_2.S_2$ pueden ser definidas reescribiéndolas como $(if\ C\ then\ a_1.S_1) + (if\ \neg C\ then\ a_2.S_2)$.

3. $\sum_{i\in I} Q'_i$, donde Q'_i puede obtenerse por alguno de los operadores anteriores y $\sum_{i\in I}$ es la suma con I como conjunto de indexación.

Existe una constante adicional e implícita denotada por $\mathbf{0}$ y definida como $\mathbf{0} = \sum_{i\in I} a_i.S_i$, con $I = \emptyset$. Por lo tanto, desde $\mathbf{0}$ no existen transiciones posibles.

Ejemplo 1 *Un proceso P se especifica a continuación junto al sistema de transiciones asociado.*

$$\begin{array}{lcl} P & \stackrel{def}{\equiv} & a.P_1 \\ P_1 & \stackrel{def}{\equiv} & b.P_2 + e.P_3 \\ P_2 & \stackrel{def}{\equiv} & c.P_2 + b.P_2 + d.P_1 \\ P_3 & \stackrel{def}{\equiv} & 0 \end{array} \qquad \begin{array}{lcl} \kappa_P & = & \{P, P_1, P_2, P_3\} \\ A_P & = & \{a, b, c, d, e\} \\ \stackrel{a}{\rightarrow} & = & \{(P, P_1)\} \\ \stackrel{b}{\rightarrow} & = & \{(P_1, P_2), (P_2, P_2)\} \\ \stackrel{c}{\rightarrow} & = & \{(P_2, P_2)\} \\ \stackrel{d}{\rightarrow} & = & \{(P_2, P_1)\} \\ \stackrel{e}{\rightarrow} & = & \{(P_1, P_3)\} \end{array}$$

Definición 8 ***Estado del proceso***. *Llamaremos estado del proceso a la constante agente que en un determinado instante representa al proceso. De acuerdo con el lenguaje que se ha definido para especificar procesos, un estado de proceso coincide con una constante agente del proceso, y a su vez, se corresponde con un nodo del grafo de transiciones del proceso.*

En la plantilla de clase OASIS, para cada proceso R existe implícitamente un atributo variable de *sort* κ_R con el mismo nombre del proceso. Dicho atributo representa el estado del proceso (la constante agente o nodo del grafo de transiciones asociado) en el cual se encuentra el proceso. Como veremos, cuando el objeto está en el estado $\mathbf{0}$ en un determinado proceso, el comportamiento del objeto no está siendo afectado por el proceso (el proceso no determina ni permisos ni obligaciones para el objeto).

Nótese que un mismo estado del proceso puede estar asociado a más de un estado del objeto (un σ distinto).

3.1 Procesos como fórmulas en Lógica Dinámica

Como se ha expuesto, los operadores de la Lógica Deóntica que expresan obligaciones, prohibiciones y permisos pueden ser vistos como una variante de Lógica Dinámica. Esto sugiere que los procesos de obligación y prohibición se corresponden del mismo modo con fórmulas en Lógica Dinámica.

Una especificación de proceso determina una conducta particular del objeto. Diremos que *un proceso afecta a un objeto* si el objeto debe comportarse según la conducta establecida por dicho proceso. Comúnmente, el objeto seguirá la conducta del proceso en partes de su vida. Un caso particular es un proceso que afecta al objeto durante toda su existencia.

Un proceso que se establece por permisos se puede expresar como un conjunto de fórmulas en Lógica Dinámica de la forma $\neg\phi \rightarrow [a]false$, del mismo modo que se indicó para las fórmulas de precondiciones.

De forma análoga, un proceso que se establece por obligaciones se puede expresar como un conjunto de fórmulas en Lógica Dinámica de la forma $\psi \rightarrow [\neg a]false$, tal modo como se hace para las fórmulas de disparos.

Como veremos, en ambos casos se obtendrán adicionalmente fórmulas por transición de estado del proceso. Estas fórmulas son del mismo tipo usado para evaluaciones, es decir, $\psi \rightarrow [a]\phi'$.

Sea R un proceso definido por el sistema de transiciones $(\kappa_R, A_R, \{\xrightarrow{a}: a \in A_R\})$, E,$E' \in \kappa_R$. Si $(E, E') \in \xrightarrow{a}$, esto suele escribirse como $E \xrightarrow{a} E'$ y se dice que a **es una acción desde** E.

Ejemplo 2 *Utilizando el proceso P presentado en el ejemplo 1.*

> De acuerdo con el sistema de transiciones de P, se tiene que: a es una acción desde P, b es una acción desde P_1 y desde P_2, c es una acción desde P_2, d es una acción desde P_2 y e es una acción desde P_1.

Definición 9 ***Predicado de ocurrencia de una acción en un estado del proceso.*** *Sea r el atributo variable asociado al estado del proceso R. Si a es una acción desde un estado E del proceso, llamaremos* ***predicado de ocurrencia de*** *a* ***en*** *E al predicado $((r = E) \wedge C)$ y lo denotaremos por Ω_{Ea}. Siendo C la expresión boolean cuando (E, E') tiene definida una condición C, sino el predicado de ocurrencia de a en E es sólo $(r = E)$. Por defecto, si a no es una acción desde E, entonces $\Omega_{Ea} \equiv false$.*

Ejemplo 3 *Los predicados de ocurrencia de cada acción en cada estado del proceso P son:*

$$\begin{array}{lcllcl} \Omega_{Pa} & \equiv & (p = P) & \Omega_{P_2c} & \equiv & (p = P_2) \\ \Omega_{P_1b} & \equiv & (p = P_1) & \Omega_{P_2d} & \equiv & (p = P_2) \\ \Omega_{P_2b} & \equiv & (p = P_2) & \Omega_{P_1e} & \equiv & (p = P_1) \end{array}$$

Para el resto de combinaciones entre estado del proceso y acciones, el predicado de ocurrencia es $false$.

Definición 10 ***Predicado de ocurrencia de una acción en un proceso.*** *Ampliando el concepto anterior al proceso como un todo, llamaremos* ***predicado de ocurrencia de*** *a* ***en*** *R, denotado por Ω_{R^*a}, a la disyunción de predicados de ocurrencia de a desde E, para cada constante agente E que define al proceso R.*

Ejemplo 4 *Los predicados de ocurrencia en P para cada una de sus acciones, son:*

$$\begin{array}{lcl} \Omega_{P^*a} & \equiv & (p = P) \\ \Omega_{P^*b} & \equiv & ((p = P_1) \vee (p = P_2)) \\ \Omega_{P^*c} & \equiv & (p = P_2) \end{array} \qquad \begin{array}{lcl} \Omega_{P^*d} & \equiv & (p = P_2) \\ \Omega_{P^*e} & \equiv & (p = P_1) \end{array}$$

El significado de Ω_{R^*a} es el siguiente: "Ω_{R^*a} se satisface si y sólo si el proceso R está en algún estado de proceso E y algún Ω_{Ea} se satisface, es decir, a es una acción desde E y además se cumple la condición C asociada (si ésta existe)".

Así, un proceso R, definido por el sistema de transiciones $(\kappa_R, A_R, \{\xrightarrow{a}: a \in A_R\})$, se corresponde con las fórmulas en Lógica Dinámica presentadas a continuación.

3.1.1 Fórmulas por transiciones.

Efectúan la evolución del proceso de acuerdo a las transiciones definidas y a la ocurrencia de acciones. Para cada $a \in A_R$, y para cada $(E, E') \in \xrightarrow{a}$ se obtiene la siguiente fórmula: $\Omega_{Ea} \rightarrow [a](r = E')$.

Ejemplo 5 *Las fórmulas por transiciones para el proceso P son:*

$$\begin{array}{lcl} (p = P) & \rightarrow & [a](p = P_1) \\ (p = P_1) & \rightarrow & [b](p = P_2) \\ (p = P_2) & \rightarrow & [b](p = P_2) \end{array} \qquad \begin{array}{lcl} (p = P_2) & \rightarrow & [c](p = P_2) \\ (p = P_2) & \rightarrow & [d](p = P_1) \\ (p = P_1) & \rightarrow & [e](p = P_3) \end{array}$$

Además, siendo a_{new} la acción de creación para instancias de la clase, entonces:

a) Si R es un proceso que establece obligaciones entonces, además, existirá la fórmula:

$$[a_{new}](r = \mathbf{0})$$

Es decir, asumimos que un proceso de obligación no afecta al objeto desde el comienzo de su existencia.

b) Si R es un proceso que establece permisos, la fórmula adicional es:

$$[a_{new}](r = E_1)$$

$E_1 \in \kappa_R$ es la constante agente de inicio del proceso.

Es decir, asumimos que un proceso de permiso afecta al objeto desde el comienzo de su existencia.

3.1.2 Fórmulas por Permisos (sólo si R es un proceso que establece permisos).

De acuerdo con el estado del proceso, una fórmula de permiso establece la situación en la cual la ocurrencia de una acción se permite. Para cada $a \in A_R$ se obtiene la siguiente fórmula: $\neg\Omega_{R^*a} \rightarrow [a]false$.

Ejemplo 6 *Si en el ejemplo tratado, el proceso P fuera un proceso que establece secuencias de acciones permitidas, las fórmulas de permiso asociadas serían:*

$$\begin{array}{lcl} \neg(p = P) & \rightarrow & [a]false \\ \neg((p = P_1) \vee (p = P_2)) & \rightarrow & [b]false \\ \neg(p = P_2) & \rightarrow & [c]false \end{array} \qquad \begin{array}{lcl} \neg(p = P_2) & \rightarrow & [d]false \\ \neg(p = P_1) & \rightarrow & [e]false \end{array}$$

3.1.3 Fórmulas por Obligaciones (sólo si R es un proceso que establece obligaciones).

De acuerdo al estado del proceso, una fórmula de obligación establece la situación en la cual la ocurrencia de una acción es obligatoria. Para cada $a \in A_R$ se obtiene la siguiente fórmula: $\Omega_{R^*a} \rightarrow [\neg a]false$.

Ejemplo 7 *Si en el ejemplo tratado, el proceso P fuera un proceso que establece secuencias de acciones obligatorias, las fórmulas de obligación asociadas serían:*

$$\begin{array}{lcl} (p = P) & \rightarrow & [\neg a]false \\ ((p = P_1) \vee (p = P_2)) & \rightarrow & [\neg b]false \\ (p = P_2) & \rightarrow & [\neg c]false \end{array} \qquad \begin{array}{lcl} (p = P_2) & \rightarrow & [\neg d]false \\ (p = P_1) & \rightarrow & [\neg e]false \end{array}$$

4 Conclusiones y trabajo futuro

En [5] se estableció un modelo básico de ejecución para especificaciones OASIS. Dicho modelo incorporaba el tratamiento de precondiciones, disparos y evaluaciones. Esto dio lugar a la construcción de un un prototipo de animador automático para especificaciones OASIS, definido en [6] y presentado en [7]. Gracias a las correspondencias establecidas en este trabajo, tanto el modelo de ejecución como el animador automático pueden ser fácilmente extendidos para cubrir la expresividad de las especificaciones de procesos. De esta forma, podemos decir que el nuevo modelo de ejecución permite animar permisos, prohibiciones, obligaciones y cambios de estado, consiguiendo la misma semántica asociada al modelo OASIS.

Actualmente nuestro interés es conseguir una adecuada integración del animador de especificaciones en el entorno CASE de OO-Method. Además, estamos determinando pautas metológicas que apoyen la actividad de captura y validación de requisitos asistida por animación automática.

Referencias

[1] Åqvist L. Deontic logic. In D.M. Gabbay and F.Guenthner, editors, Handbook of Philosophical Logic II, pp.605-714. Reidel, 1984.

[2] Coleman D., Hayes F., Bear S. Introducing Objectcharts or how to use statecharts in object-oriented design, IEEE Transactions on Software Engineering, Vol.18, No.1, January 1992.

[3] Harel,D. Dynamic Logic in Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pags. 497-694. Reidel 1984.

[4] Harel D. On visual formalisms, Communications of the ACM, Vol.31, No.5, May1988.

[5] Letelier P., Sánchez P., Ramos I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitos, ASOO'97, Buenos Aires, Argentina, Agosto 1997.

[6] Letelier P., Sánchez P., Ramos I. Guías para la Generación Automática de un Programa Lógico Concurrente que sirve de Prototipo para una Especificación OASIS, Informe Técnico DSIC-II/26/97, Universidad Politécnica de Valencia, 1997.

[7] Letelier P., Sánchez P., Ramos I. Animación de especificaciones OASIS en entornos concurrentes, Actas de las II Jornadas de Trabajo de Menhir, DSIC-Universidad Politécnica de Valencia, Febrero 1998.

[8] Meyer J.-J.Ch. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.

[9] Milner R. Communication and Concurrency. Prentice Hall Series in Computer Science, C.A.R. Hoare, Series Editor. 1989.

[10] Pastor O., Ramos I. OASIS versión 2 (2.2) : A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.

[11] Pastor O., Insfrán E., Pelechano V.,Romero J.,Merseguer J. OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods. Conference on Advanced Information Systems Engineering (CAiSE '97). Barcelona, Spain. June 1997.

[12] Wieringa R.J., Meyer J.-J.Ch. Actors, Actions and Initiative in Normative System Specification Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

Una aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objetos[=]

Pedro Sánchez, Patricio Letelier, Isidro Ramos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, Valencia, España
email {ppalma | letelier | iramos}@dsic.upv.es

Abstract

OASIS es un lenguaje formal para la especificación de modelos conceptuales que sigue un enfoque orientado a objeto. Estamos interesados es representar la semántica de OASIS en el modelo de concurrencia de las redes de Petri. A partir de una representación gráfica para OASIS podemos derivar el equivalente en redes de Petri y animar posteriormente la especificación. La existencia de simuladores para este tipo de redes hacen atractiva la consideración de estos formalismos en las fases tempranas de la ingeniería de requisitos.

1 Introducción

OASIS [Pas95] es un lenguaje formal para la especificación de modelos conceptuales siguiendo un enfoque orientado a objetos. La tarea de verificación de modelos conceptuales en OASIS exige disponer de algún formalismo preciso y con toda la capacidad expresiva para capturar los aspectos de interés del problema. Estamos interesados en la interpretación de OASIS en modelos de concurrencia como son las Redes de Petri Objetuales. La existencia de herramientas que permitan animar modelos especificados con esta semántica proveen un camino hacia la obtención de prototipos ejecutables equivalentes a la especificación original.

Este artículo ofrece una traducción de los elementos fundamentales del lenguaje gráfico de OASIS a las redes OO. La Figura 1 representa la relación existente entre el lenguaje, su equivalente gráfico y por último, las Redes de Petri OO y Temporales (OOTPN) que constituyen nuestro núcleo operativo. La semántica de una especificación OASIS es representable mediante un conjunto de diagramas.

Definiciones previas

Un objeto estará representado por una serie de subprocesos concurrentes. Cada subproceso concurrente representa una parte del todo que es el proceso más general que representa al

[=] Este trabajo está financiado por el proyecto *MENHIR* de la Comisión Interministerial de Ciencia y Tecnología con referencia TIC97-0593-C05-01.

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto. Actas de las III Jornadas de Ingeniería del Software. 1998.

objeto. Las siguientes son definiciones que consideramos necesarias, además de la funcionalidad básica descrita en los *statechars* de [Har88] y los *objectcharts* de [Col92].

Un **subproceso** es cada una de las partes concurrentes que forman y definen la semántica de un objeto. La definición de **estado** de un subproceso es equivalente a la que se presenta en los statecharts. Cada subproceso se encuentra siempre en un único estado. Los **arcos** representan transiciones entre estados. La transición entre dos estados puede deberse a la ocurrencia de una acción o al simple cumplimiento de una precondición. Una transición entre estados implica un cambio en el estado del objeto y los estados de llegada se caracterizan mediante postcondiciones. Un arco es del tipo "**input/output**" si su sintaxis es de la forma:

[precondition] input_action / output_action [postcondition]

Un arco es tipo "**input**" si su sintaxis es de la forma:

[precondition] input_action [postcondition]

Un arco es de tipo "**output**" si su sintaxis es de la forma:

[precondition] / output_action [postcondition]

Las precondiciones y postcondiciones son fórmulas en lógica de primer orden ecuacional sobre los atributos del objeto. Las precondiciones pueden incluir además el predicado **In**(*subproc.estado_subproc*) que se evalúa a true si el subproceso llamado *subproc* está actualmente en el estado llamado *estado_subproc*.

Un **subproceso afectado** es el que cumple una de las siguientes condiciones:

1. Llega una acción (act) y el subproceso está en un estado con al menos un arco (que llamamos **arco afectado**) conteniendo a la acción act como input_action.
2. No llega ninguna acción pero el subproceso está en un estado con un arco tipo **output** (que llamamos **arco afectado**).

Un subproceso se dice que está **sensibilizado** si es un subproceso afectado cuyo arco afectado cumple su precondición. Una transición puede ocurrir para un objeto si y sólo si todos los subprocesos afectados están sensibilizados.

Partiremos del hecho de que una acción conlleva: (1) el envío, (2) la recepción, (3) la atención, y (3) la ejecución. Distinguiremos tres tipos de arcos dependiendo de la sincronía existente[1].

1) Arcos cuya output_action se envía y ejecuta de forma síncrona. Ésta es la interpretación de [Har88] y usamos el mismo tipo de arco, es decir:



2) Arcos cuya output_action se atiende **exactamente un paso después** de la recepción:



3) Arcos cuya output_action será atendida **algún paso después**:



2 Elementos básicos

De entre todas las combinaciones posibles a partir de los elementos anteriores sólo un subconjunto de subprocesos, que consideramos **básicos**, serán necesarios para representar los conceptos fundamentales del lenguaje OASIS (atributos, evaluaciones, precondiciones, triggers, operaciones y protocolos).

Subprocesos tipo I. Aquellos que poseen un único estado (que llamaremos Idle) con un único arco tipo **input** etiquetado con precondición true y postcondición post. La Figura 2 muestra un subproceso representativo de este primer tipo.

Subprocesos tipo II. Aquellos que poseen un único estado (que llamaremos Idle) con un único arco tipo **input** etiquetado con precondición pre y postcondición true. Aunque son equivalentes, excluimos de este tipo II a aquellas precondiciones que contengan el predicado **In()** definido anteriormente. Dedicaremos un tipo particular a los subprocesos que contengan dichos predicados y una implementación en OOTPN específica (véase la Figura 3).

Subproceso tipo III. Aquellos que poseen un único estado (que llamaremos Idle) con un único arco tipo **input** etiquetado con precondición pre que incluye el predicado **In()** y una postcondición true (véase la Figura 4).

Subproceso tipo IV. Aquellos que poseen un único estado (que llamaremos Idle) con un único arco tipo **output asíncrono** para **exactamente un paso después** y etiquetado con precondición pre y postcondición true (véase la Figura 5).

Subproceso tipo V. Poseen un único estado (que llamaremos Idle) con un único arco tipo **output asíncrono** para **algún un paso después** y etiquetado con precondición pre y postcondición true. La Figura 6 muestra un subproceso representativo de este quinto tipo.

Subproceso tipo VI. Poseen un único estado (que llamaremos Idle) con n arcos de tipo **input** donde cada $arco_i$ está etiquetado con la precondición pre_i y la misma acción en todos los $arco_i$ (véase la Figura 7).

---

[1] Asumiremos que el envío es siempre síncrono con la recepción.

Subprocesos tipo VII. Aquellos representados por un grafo dirigido de estados donde Idle es el estado de partida y cada $arco_i$ entre dos estados del subproceso es de tipo **input** y posee una precondición pre_i y la postcondición true. La única restricción que posee dicho grafo es que del mismo estado nunca salen dos o más arcos con precondiciones solapadas para la misma acción. Todas las demás combinaciones son posibles. La Figura 8 muestra un ejemplo.

Subproceso tipo VIII. Aquellos representados por un grafo dirigido de estados que cumple las siguientes restricciones:

i) Idle es el estado de partida y de él sale uno o más arcos tipo **input/output asíncrono** para **exactamente un paso después** con postcondición true.

ii) Del resto de estados distintos de Idle los arcos (arc_i) son del tipo **output asíncrono** para **exactamente un paso después** con precondición pre_i y postcondición true.

iii) Del mismo estado no salen dos o más arcos con precondiciones solapadas para la misma acción.

La Figura 9 es un ejemplo de este donde vemos que el arco de salida de Idle es del tipo input/output, que las postcondiciones son true en todos los arco, y que tanto las acciones como las precondiciones pueden usarse en cualquier arco.

3 Plataforma

Una especificación OASIS da lugar a varios subprocesos de alguno de los tipos anteriores. Cada tipo posee alguna peculiaridad que lo hace distinguible del resto y requiere una implementación concreta en OOTPN. Además de la red OOTPN obtenida para cada subproceso aparece un nivel superior encargado de implementar la funcionalidad asociada a la transición entre estados. La definición de transición para un objeto hace necesario un protocolo de confirmación de dos fases. En la primera fase se detecta qué subprocesos están sensibilizados. En la segunda fase se confirma a todos los subprocesos si la transición tiene o no lugar (será afirmativo si todos los subprocesos afectados son también subprocesos sensibilizados). Según esto, cada subproceso recibe en la primera fase del protocolo la acción. El subproceso puede responder de tres formas:

1. **Se ve afectado** por la acción y **está sensibilizado** a la transición.
2. **Se ve afectado** por la acción y **no está sensibilizado** a la transición.
3. **No se ve afectado** por la acción.

Si todos los procesos están sensibilizados o no se ven afectados entonces el protocolo confirmará en su segunda fase a todos los subprocesos. Aquellos subprocesos sensibles a la

acción transitarán en el arco específico. Los subprocesos que no se veían afectados no transitarán. Si alguno de los subprocesos responde al protocolo como que no está sensibilizado entonces la segunda fase no informa a los subprocesos.

La Figura 10 presenta la plataforma en OOTPN para representar la semántica de un objeto. El nivel inferior es la red asociada a cada subproceso. El nivel superior representa la red encargada de la contienendo lugares con los valores de los atributos del objeto. Además encamina la llegada de acciones a todos los subprocesos. Es necesaria una subred dentro de esta de nivel superior para la gestión ordenada de las acciones que el objeto va a atender (tanto provenientes del exterior como del propio objeto).

4 OOTPN por subproceso.

Consideraremos la sintaxis y semántica definidas en las OOTPN de [Ess96]. La posibilidad de especificar la comunicación entre distintas OOTPN mediante **interfaces** de entrada y salida, el poder encapsular en **componentes** reusables, y la existencia de un simulador en entorno gráfico hacen adecuado tal elección. Cada componente posee interfaces que pueden ser de lectura, escritura o ambos (véase la Figura 11). La Figura 12 muestra un ejemplo de red OOTPN con cinco componentes. La propiedad de herencia y el polimorfismo confieren a estas redes el atributo de orientadas a objeto.

Red OOTPN subprocesos tipo I. Describiremos la red para los procesos más representativos. La Figura 13 muestra parte de la red para subprocesos de este tipo. Notas aclaratorias:

1) El lugar ev es el lugar que recoge la acción que va ejecutar el subproceso.
2) El interfaz End marca la finalización de la operación. Los atributos Atr_1 hasta Atr_n son los que aparecen en la postcondición post del arco y van a ser modificados por la transición. En algún nivel superior de la jerarquía de la red hay un lugar para cada atributo y mediante estos interfaces se accede al valor. La modificación pasa por extraer y posteriormente añadir un nuevo token con el valor calculado.
3) Siendo act la acción del arco del subproceso, la transición Trans tiene por condición que ev=act y como efectos hacer cierta la postcondición post.
4) Idle es un lugar que almacena un token cuando el subproceso está en dicho estado. En este tipo de subprocesos siempre hay un token en dicho lugar por lo que la transición recoge y deposita el mismo token. A pesar de ser innecesario se mantiene por llevar uniformidad en la forma de traducción de todos los subprocesos.

5) El interfaz Is_Idle permite que desde el nivel superior se detecte si el subproceso está en dicho estado.

La Figura 14 muestra la totalidad de la red OOTPN para subprocesos tipo I. El componente encapsulado es la red de la Figura 13. Notas aclaratorias:

1) El subproceso termina de procesar una acción cuando acaba el componente. Las interfaces indicando el fin están conectadas y dicho fin se propaga a un nivel superior donde se implementa el protocolo de dos fases mencionado anteriormente.

2) Los atributos que maneja el componente no están en este nivel sino en alguno superior.

3) La operatoria es la siguiente:

   a) En la primera fase del protocolo llega una acción por el interfaz test_ev.

   b) El evento se deposita en el lugar event_place.

   c) La transición Process no posee condición alguna. El efecto de dicha transición es responder en el lugar ans_prev y en el interfaz ans_test alguna de las siguientes: *está sensibilizado* a la acción; *no le afecta* o le afecta pero *no está sensibilizado* (es decir, no cumple las precondiciones pre del arco). Dado que en este tipo de subprocesos la precondición es true, nunca se va a responder que no está sensibilizado. El guardar el valor en ans_prev permite que en la segunda fase del protocolo se transite o no dentro de este subproceso.

   d) Si el protocolo de dos fases acepta la acción entonces llegará por el interfaz in_ev y se depositará en el lugar in_place. Si en el lugar ans_prev se guardó *le afecta*, entonces se procesará la acción (tal como indicaba la Figura 13). Caso contrario, se indica el fin y la acción no se procesa. Si algún subproceso respondiera que *no está sensibilizado*, entonces a este subproceso nunca hubiese llegado confirmación alguna, lo que equivale a que la acción se descarte.

Red OOTPN subprocesos tipo III. La Figura 15 muestra parte de la red para subprocesos de este tipo. Nota aclaratoria:

1) Siendo act la acción del arco del subproceso, la transición Trans tiene por condición que ev=act y ningún efecto.

La Figura 16 muestra la totalidad de la red OOTPN para subprocesos tipo III. El componente encapsulado es la red de la Figura 15. Notas aclaratorias:

1. Si la acción que hay en event_place es la del arco del subproceso que implementa y además se cumple la precondición pre de dicho arco, expresada en predicados del tipo **In()**, entonces en ans_test y ans_prev responderá que *está sensibilizado*.

2. Si la acción que hay en event_place es la del arco del subproceso que implementa y además **no** se cumple la precondición pre de dicho arco expresada en predicados del tipo **In()**, entonces en ans_test y ans_prev responderá que *no está sensibilizado*.

3. Si la acción que hay en event_place **no** es la del arco del subproceso que implementa entonces en ans_test y ans_prev responderá que *no le afecta*.

4. Se han incluido varios interfaces Test_in_i para el mismo subproceso pues la precondición pre puede ser la disyunción de predicados In() indicando que la acción se acepta si está en alguno de los estados explicitados.

Red OOTPN subproceso tipo IV. La Figura 17 muestra parte de la red para subprocesos de este tipo. Notas aclaratorias:

1) La transición Trans no tiene condición de disparo y tiene por efecto la producción de la acción especificada de salida en el arco del subproceso (asíncrona para el siguiente paso de ejecución).
2) El interfaz evnext representa la acción generada de salida por parte de Trans y flagnext es un indicador para detectar que dicha acción va al propio objeto y que se va a procesar en el siguiente paso.

La Figura 18 muestra la totalidad de la red para subprocesos tipo IV. El componente encapsulado es la red de la Figura 17. Notas aclaratorias:

1) La diferencia radica en que por implementar un arco tipo output del subproceso no es necesaria la llegada de acción alguna para que se dispare la transición. Es necesario entonces considerar la existencia de un evento, que llamaremos tick y que llegará a todos[2] los subprocesos (cuando no haya acciones pendientes de procesar pues si las hay, dichas acciones desempeñan el papel del evento tick. Dicho esto, el protocolo de nivel superior debe proveer el control adecuado para la generación automática del evento tick si no hay acciones pendientes de procesar).

La llegada de tick al lugar event_place conlleva la evaluación de la precondición pre del arco (en función de los atributos desde Atr_1 hasta Atr_n) y el disparo, tal como sigue:

a) Si cumple la precondición pre del arco evaluada sobre los atributos, en ans_test y ans_prev responderá que *está sensibilizado*.
b) Si **no** se cumple la precondición pre de dicho arco, entonces en ans_test y ans_prev responderá que *no le afecta*.

[2] Para los otros tipos de subproceso la llegada del evento tick implica responder que *no le afecta*.

2) El interfaz evnext representa la acción generada desde el componente y el interfaz flagnext sirve de indicador de que dicha acción es asíncrona para el siguiente paso.

3) Es responsabilidad de la red OOTPN del protocolo hacer que la acción generada tenga preferecia sobre el resto de acciones y que se atienda en el siguiente paso.

Red OOTPN subprocesos tipo V. La Figura 19 muestra parte de la red para la subprocesos de este tipo. Notas aclaratorias:

1) La transición Trans no tiene condición de disparo y tiene por efecto la producción de la acción especificada de salida en el arco del subproceso.

2) El interfaz Ev_asyn representa la acción generada de salida por parte de Trans.

La Figura 20 muestra la totalidad de la red para subprocesos tipo V. El componente encapsulado es la red de la Figura 19. Notas aclaratorias:

1) La transición Process tiene una operativa totalmente similar a la de los subprocesos tipo IV:

a) Si cumple la precondición pre del arco en ans_test y ans_prev responderá que *está sensibilizado*.

b) Si **no** se cumple la precondición pre de dicho arco, entonces en ans_test y ans_prev responderá que *no le afecta*.

2) El interfaz Ev_asyn representa la acción generada desde el componente que es de salida al nivel superior.

Red OOTPN subprocesos tipo VII. Para simplificar, consideraremos el ejemplo que se muestra en la Figura 21 y la red de la Figura 22. Ampliar el estudio a cualquier subproceso de este tipo hace más extensa la exposición pero no más compleja. Notas aclaratorias:

1) La red de la Figura 22 tiene la misma estructura que el grafo de la Figura 21. Los arcos del grafo están asociados a las transiciones de la red. El disparo de la transición es la transición por el arco. Las precondiciones de los arcos se comprueban a través de los interfaces de sólo lectura de la red. Los estados del grafo vienen representados por lugares en la red.

2) La transición To_After_Act_1 representa el paso del estado Idle al estado After_Act_1. La condición de disparo es que se cumpla la precondición pre_1 del arco y que la acción sea la indicada en dicho arco. El interfaz de lectura/escritura llamado st es actualizado al valor del estado que se alcanza (es decir, After_Act_1). Dicho valor se propaga a niveles superiores de forma que los subprocesos del tipo III (con predicados In()) puedan evaluar sus precondiciones.

3) La transición To_Idle representa el paso del estado After_Act_1al estado Idle. La condición de disparo es que se cumpla la precondición pre_2 del arco y que la acción sea la de dicho arco. El interfaz de lectura/escritura llamado st es actualizado al valor del estado que se alcanza (es decir, Idle). Al igual que antes, dicho valor se propaga a niveles superiores de forma que los subprocesos del tipo III (con predicados In()) puedan evaluar sus precondiciones. La Figura 24 muestra la totalidad de la red OOTPN para subprocesos tipo VII. El componente encapsulado es la red de la Figura 22. Notas aclaratorias:

1) El lugar State mantiene información del valor actual de estado del subproceso. Su valor inicial es Idle. A través del interfaz test_subproc otros subprocesos podrán entonces comprobar el valor del estado del subproceso.

2) Hay una transición Process por cada estado del subproceso. En el ejemplo de la Figura 23 hay dos estados lo que implica dos transiciones, Process_1 y Process_2 que cumplen lo siguiente:

   a) La transición Process_1 se disparará si el subproceso está en el estado Idle (lo indica el interfaz del componente). Si la acción de entrada de event_place es act_1, entonces en ans_test y ans_prev responderá *que está sensibilizado* responderá que *no le afecta* en caso contrario.

   b) La transición Process_2 se disparará si el subproceso está en el estado After_act_1 (lo indica el interfaz del componente). Si la acción de entrada de event_place es act_2, entonces en ans_test y ans_prev responderá que *está sensibilizado* y responderá que *no le afecta* en caso contrario.

3) Esta red no cancela el protocolo de dos fases pues lo único que gestiona es la evolución del estado actual en el grafo. Los subprocesos del tipo III son los que complementan a éste introduciendo la prohibición de los protocolos de OASIS.

4) Es posible generalizar a cualquier tipo de subproceso a partir de las pautas del ejemplo.

5 Conclusiones

La justificación de que la semántica de un objeto OASIS es equivalente a otra en redes de Petri puede hacerse en demostrando: (1) que la semántica de OASIS definida en lógica dinámica es equivalente a la de los diagramas de subprocesos, (2) que la semántica de subprocesos es equivalente a la semántica asociada a las redes de Petri *OOTPN* para cada tipo de subproceso viendo que los marcados alcanzables por la red son los correctos, y (3), que la equivalencia semántica se preserva tras el cambio de estado del objeto [Sán97b].

6 Referencias

[Col92] Coleman D., Hayes F., Bear S., *Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*, IEEE Transactions on Software Engineering, Vol. 18, 1992.

[Ess96] Esser R., *An Object Oriented Petri Net Approach to Embedded System Design*, PhD Thesis, Swiss Federal Institute of Technology Zurich, 1996.

[Har88] Harel D., *On Visual Formalisms*, Communications of the ACM, V.31, 1988.

[Let97] Letelier P., Sánchez P., Ramos I., *Animación de Modelos Conceptuales para ayudar en la validación de requisitos*, Argentine Symposium on Object Orientation, Buenos Aires, pp. 103-110, 1997

[Pas95] Pastor O., Ramos I. OASIS (2.2): *A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, SPUPV-95.788, Valencia, 1995.

[Ram93] Ramos I., Pastor O., Cuevas J., Devesa J., *Objects as Observable Processes*, Actas del 3rd. Workshop on the Deductive Approach to Information System Design, Cataluña, 1993.

[Sán97a] Sánchez P., Letelier P., Ramos I., *Constructs for Prototyping Information Systems with Object Petri Nets*, IEEE SMC'97, Orlando, pp. 4260-4265, 1997.

[Sán97b] Sánchez P., Letelier P., Ramos I., *Representación de Objetos como Teorías y Traducción a un Modelo de Concurrencia*, Revista *Computación y Sistemas*. ISSN 0188-8048, 1997.

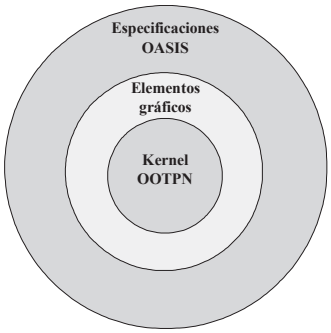


Figura 1: Relación entre formalismos.

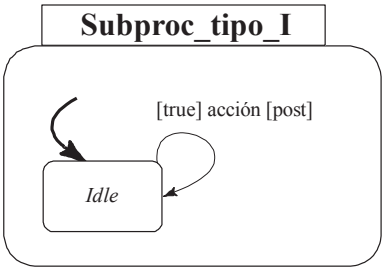


Figura 2: Subproceso tipo I.

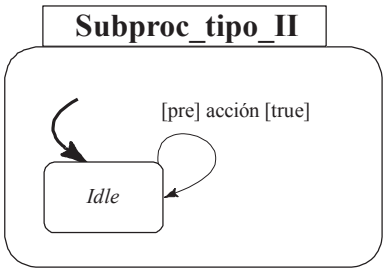


Figura 3: Subproceso tipo II.

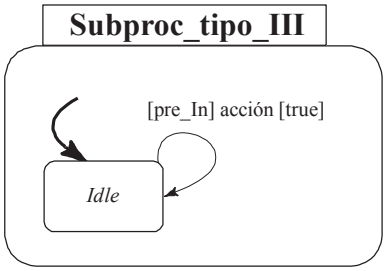


Figura 4: Subproceso tipo III.

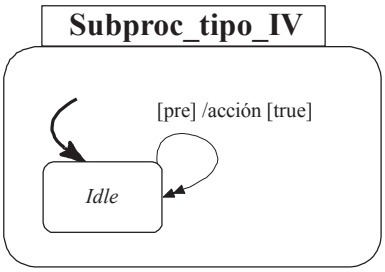


Figura 5: Subproceso tipo IV.

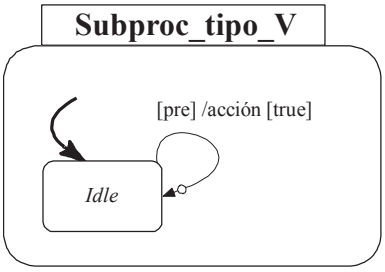


Figura 6: Subproceso tipo V.

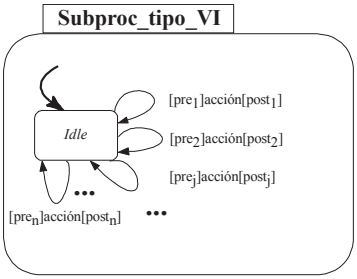


Figura 7: Subproceso tipo VI.

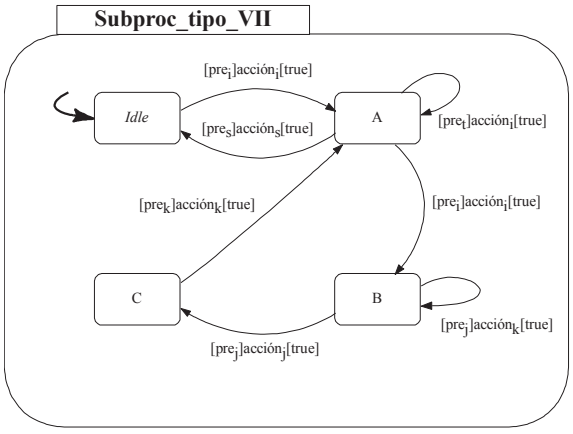


Figura 8: Subproceso tipo VII.

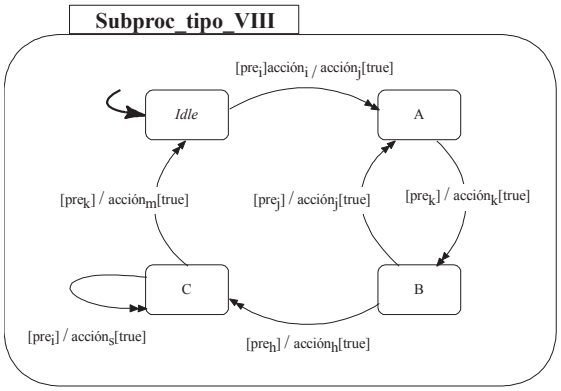


Figura 9: Subproceso tipo VIII.

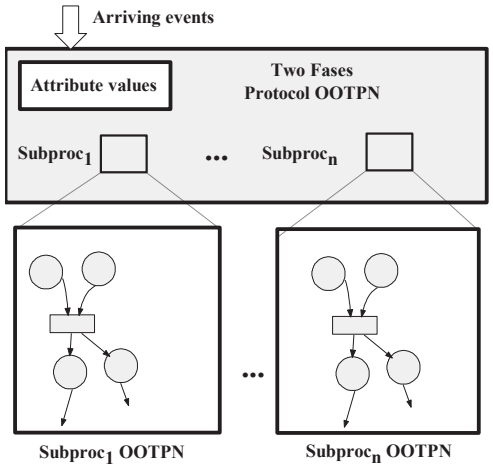


Figura 10: Plataforma de representación.

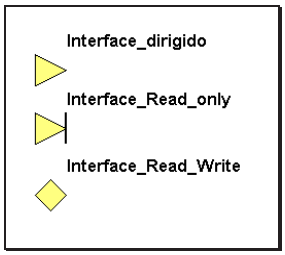


Figura 11: Tipos de Interfaz utilizados.

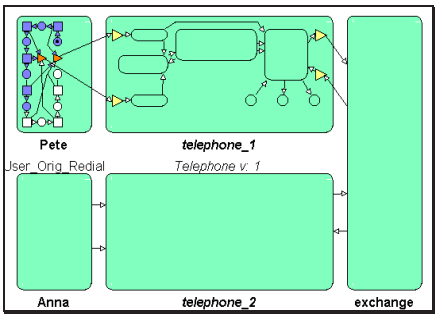


Figura 12: Ejemplo red OOTPN.

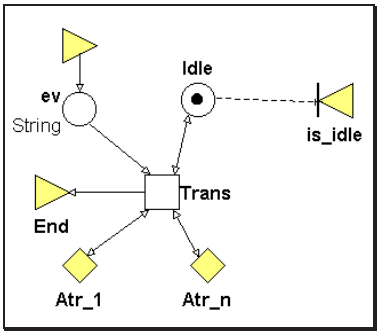


Figura 13: OOTPN subprocesos tipo I.

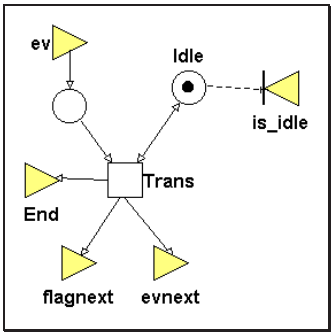


Figura 17: OOTPN subprocesos tipo IV.

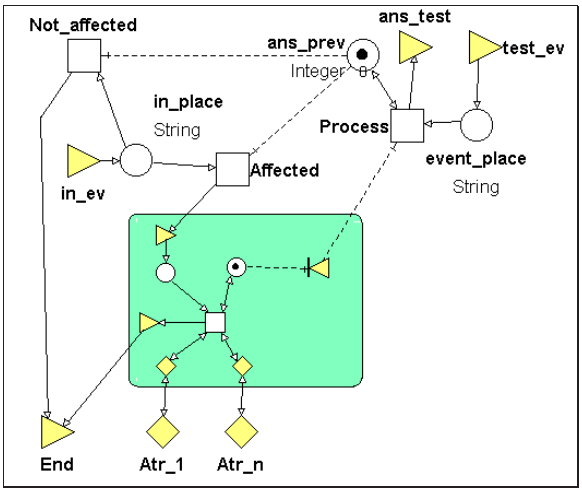


Figura 14: OOTPN total subprocesos I.

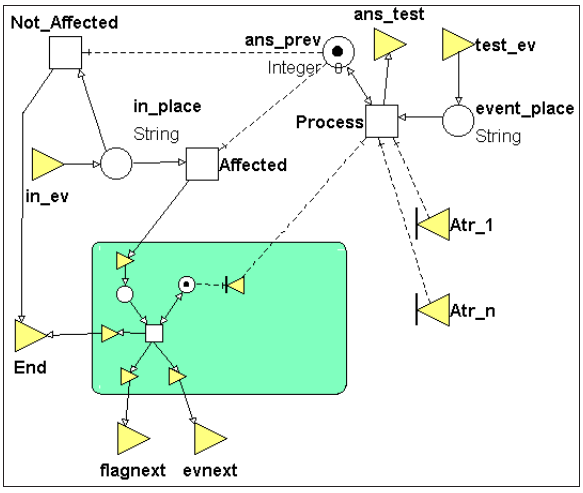


Figura 18: OOTPN total subprocesos IV.

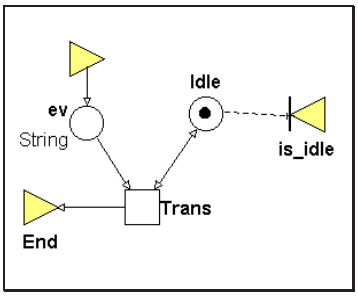


Figura 15: OOTPN subprocesos III.

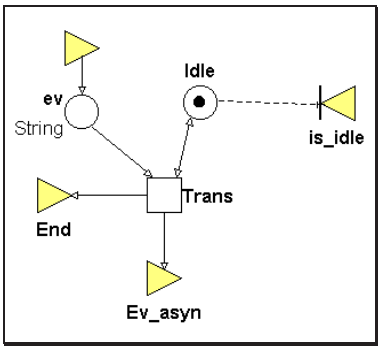


Figura 19: OOTPN subprocesos V.

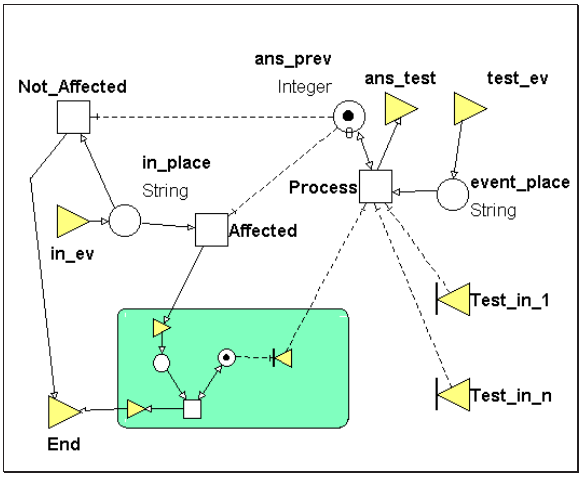


Figura 16: OOTPN total subprocesos III.

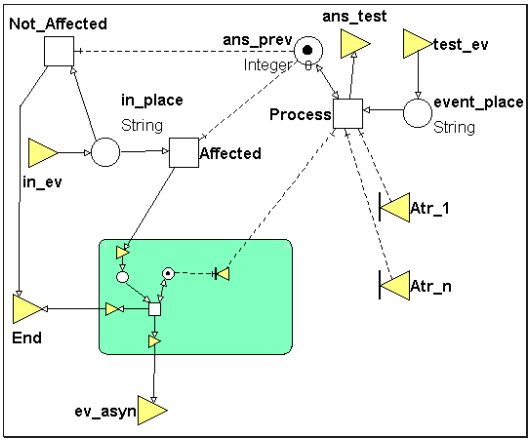


Figura 20: OOTPN total subprocesos V.

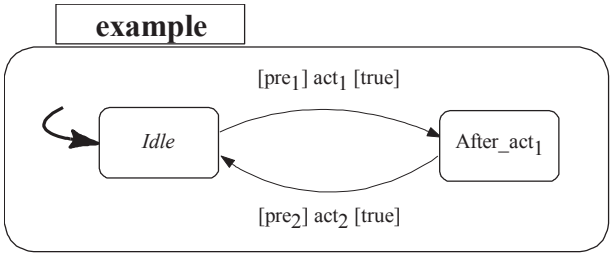


Figura 21: Ejemplo subproceso tipo VII.

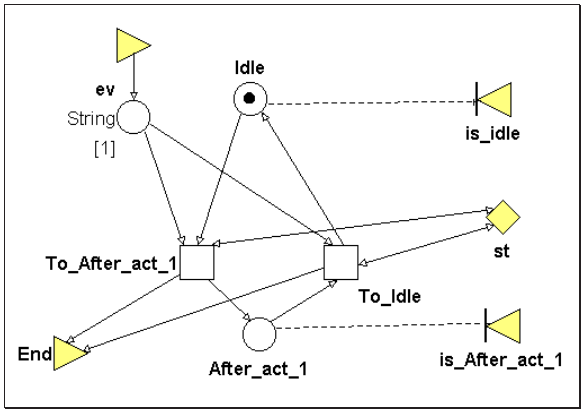


Figura 22: OOTPN parcial para el ejemplo de subprocesos tipo VII.

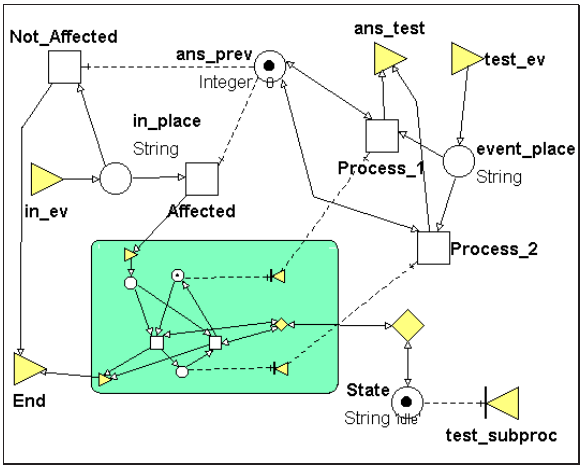


Figura 23: OOTPN total subprocesos VIII.

Una aproximación a las pautas de representación de OASIS en un formalismo gráfico[=]

Pedro Sánchez, Patricio Letelier, Isidro Ramos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n, 46071 Valencia - Spain
☎ 3877350; Fax: 3877357
email {ppalma | letelier | iramos}@dsic.upv.es

Abstract

OASIS es un lenguaje formal para la especificación de modelos conceptuales siguiendo un enfoque orientado a objetos. Existe un interés creciente en poder aprovechar las ventajas de las especificaciones formales (no ambigüedad, consistencia, verificación automática de propiedades, etc.) para resolver los problemas inherentes a los métodos de especificación informales existentes. Este artículo presenta algunas pautas para representar la semántica de OASIS mediante una notación gráfica basada en los StateCharts de Harel y en los ObjectCharts de Coleman, con algunas modificaciones. Entre las ventajas de dicha notación está el poder explicitar de forma rigurosa la implementación que se haga del modelo OASIS en otros modelos de concurrencia, tales como las redes de Petri de alto nivel.

Keywords: Orientación a objetos, Lenguajes formales de especificación, Statecharts

1 Introducción

OASIS (Open and Active Specification of Information Systems) [Pas95] es un lenguaje formal para la especificación de modelos conceptuales siguiendo un enfoque orientado a objetos. Otros lenguajes con una motivación similar a la de OASIS son TROLL [Jun95], Oblog [Ser92] y Gnome [Ser94]. La tarea de verificación de un modelo conceptual en OASIS exige disponer de algún formalismo preciso y, a la vez, con toda la capacidad expresiva necesaria para capturar los aspectos de interés del problema. Estamos interesados en la interpretación de los conceptos OASIS en modelos de concurrencia como pueden ser las Redes de Petri Objetuales [Ess96]. La existencia de herramientas que permiten animar modelos especificados con esta semántica nos provee un camino hacia la obtención de prototipos ejecutables equivalentes a la especificación original.

[=] Este trabajo está financiado por el proyecto *MENHIR* de la Comisión Interministerial de Ciencia y Tecnología con referencia TIC97-0593-C05-01.

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a las pautas de representación de OASIS en un formalismo gráfico. Actas de las III Jornadas de Ingeniería del Software. 1998.

Este artículo presenta una aproximación a pautas que permitan representar OASIS mediante una notación grafica sin perder la semántica asociada al modelo. La animación de OASIS puede verse facilitada si se dispone de una representación grafica con una semántica claramente definida. El poder formalizar el proceso de representación gráfica, a partir del lenguaje textual, nos ofrece un camino para justificar formalmente la representación que del modelo se haga en el modelo de concurrencia escogido. La notación gráfica usada es, en parte, la que presenta D. Coleman [Col92] inspirado a su vez en los trabajos de D. Harel [Har88]. Se aborda la representación de atributos, valuaciones, precondiciones, triggers, protocolos y operaciones de OASIS concluyendo con un ejemplo representativo.

2 Preliminares de OASIS

En OASIS, un **objeto** es un *proceso observable* [Ram93] cuya vida está caracterizada por la ocurrencia de acciones, tanto si son solicitadas como si son recibidas por el objeto, y que determinan las posibles observaciones que del mismo se hagan. Desde el punto de vista estático, llamamos **atributos** al conjunto de propiedades que describen estructuralmente al objeto. Los valores asociados a cada propiedad del objeto determinan el **estado** del objeto en un instante dado. Una **acción** es una tupla formada por el cliente, el servidor y el servicio solicitado. Los servicios que un objeto proporciona a nivel "atómico" se denominan **eventos**. Los eventos pueden ser estructurados como **procesos**. Además de la semántica propia del sublenguaje utilizado para especificar procesos se contempla una semántica adicional para distinguir entre: procesos de obligación (*operations*) y procesos de prohibición (*protocols*). Un proceso de obligación es un servicio de mayor nivel. Por otra parte, un proceso de prohibición define secuencias permitidas de acciones en la vida del objeto.

3 Formalismo gráfico

Los ObjectCharts [Col92] (definidos a partir de los Statecharts [Har88]) extienden la notación de las máquinas de estado finitas produciendo descripciones modulares, jerárquicas y estructuradas de sistemas. Ambos permiten expresar concurrencia, jerarquización y comunicación por difusión global (*broadcast*). Hemos escogido un subconjunto de los aspectos definidos en estos diagramas y se han incluido nuevos que permiten expresar los conceptos de OASIS de forma más directa. La semántica añadida puede ser expresada completamente a partir de la notación original de los statecharts. Un objeto estará

representado por una serie de subprocesos concurrentes donde cada uno representa una parte del todo que es el proceso más general representando al objeto.

Un **subproceso** es cada una de las partes concurrentes que forman y definen la semántica de un objeto. Se representa gráficamente mediante una caja cerrada con una etiqueta superior que indica el nombre de dicho subproceso. La definición de estado de un subproceso es equivalente a la que se presenta en los statecharts. Básicamente es una de las posibles situaciones correspondientes a la máquina de estados finita asociada al subproceso. Cada subproceso se encuentra siempre en un único estado. Un estado se representa gráficamente en un subproceso como un rectángulo etiquetado por su nombre. El estado inicial de cualquier subproceso es el que posee un arco de entrada sin etiquetar.

Los **arcos** representan transiciones entre estados. La transición entre dos estados puede deberse a la ocurrencia de una acción o al simple cumplimiento de una precondición. Una transición entre estados implica un cambio en el estado del objeto y los estados de llegada se caracterizan mediante postcondiciones. Las siguientes tres definiciones presentan los tres tipos de arco relevantes.

Un arco es del tipo "**input/output**" si su sintaxis es de la forma:

[precondition] input_action / output_action [postcondition]

Un arco es tipo "**input**" si su sintaxis es de la forma:

[precondition] input_action [postcondition]

Un arco es de tipo "**output**"si su sintaxis es de la forma[1]:

[precondition] / output_action [postcondition]

Las precondiciones y postcondiciones son fórmulas en lógica de primer orden ecuacional sobre los atributos del objeto. Las precondiciones pueden incluir además el predicado **In**(*subproc*.*estado_subproc*) que se evalúa a true si el subproceso llamado *subproc* está actualmente en el estado llamado *estado_subproc*.

Un **subproceso afectado** es el que cumple una de las siguientes condiciones:

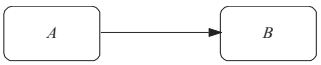
1. Llega una acción (act) y el subproceso está en un estado con al menos un arco (que llamamos **arco afectado**) conteniendo a la acción act como input_action.
2. No llega ninguna acción pero el subproceso está en un estado con un arco tipo **output** (que llamamos **arco afectado**).

Un subproceso se dice que está **sensibilizado** si es un subproceso afectado cuyo arco afectado cumple su precondición.

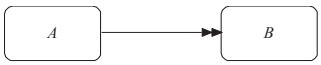
[1] Obsérvese que la barra invertida no puede suprimirse para evitar la confusión con el tipo input.

Una transición puede ocurrir para un objeto si y sólo si todos los subprocesos afectados también son subprocesos sensibilizados. Partiremos del hecho de que una acción conlleva: (1) el envío, (2) la recepción, (3) la atención, y (3) la ejecución. Distinguiremos tres tipos de arcos dependiendo de la sincronía existente[2].

1) Arcos cuya output_action se envía y ejecuta de forma síncrona. Ésta es la interpretación de [Har88] y usamos el mismo tipo de arco, es decir:



2) Arcos cuya output_action se atiende (y se ejecuta si cumple las precondiciones) **exactamente un paso después** de la recepción. Representamos estos arcos de la forma:



3) Arcos cuya output_action será atendida **algún paso después** (y ejecutada si cumple precondiciones). Estos arcos presentan la forma:



Para el ejemplo de la Figura 1 plantearemos distintas situaciones:

- Hay tres subprocesos concurrentes del objeto Z y los estados A, E y C son los de partida en cada subproceso al iniciarse dicho proceso.
- Si el objeto Z está en la situación (A,F,E), entonces la llegada de la acción e implica pasar a la situación (B,C,E). Si el proceso Z está en la situación (B,C,E), entonces la llegada de la acción f implica pasar[3] a (A,F,E). Si el subproceso S está en el estado F y se cumple la precondición atr1=value entonces en el siguiente paso se dará la acción k. Si el proceso Z está en la situación (A,E,D) y se recibe n cumpliéndose atr1=value1 y no cumpliéndose atr3=value3, entonces no se ejecuta la acción n por no aceptarse en todos los subprocesos que poseen un arco etiquetado con dicha acción. Caso de aceptarse, se cumplirá la postcondición atr2=value2 indicada en la transición.

4 Pautas de representación

Para los conceptos básicos de OASIS daremos los diagramas de subprocesos equivalentes.

Atributos. El conjunto de atributos de un objeto no se indica de forma explícita sino que se extraen de los arcos etiquetados. Entre todas las fórmulas de precondiciones y postcondiciones

[2] Asumiremos que el envío es siempre síncrono con la recepción.

[3] Nótese que la acción *f* es de salida y entrada en el mismo paso dentro del mismo objeto.

podemos saber qué atributos son variables. Asumiremos como constantes aquellos que aparezcan únicamente en fórmulas de precondición en la etiqueta de los arcos.

Evaluaciones. Como sabemos, las evaluaciones explicitan los efectos que tienen las distintas acciones sobre el conjunto de atributos del objeto. Una evaluación puede ser tal que se aplique siempre que ocurre una acción. Además, puede especificar las condiciones que deben cumplir los atributos del objeto para que dicha evaluación tenga lugar. Dada una especificación OASIS, si existen distintas valuaciones para la misma acción (variando el estado en el que se aplican) el subproceso asociado a la valuación debe contener un único estado (que llamamos idle) del que salgan tantos arcos como valuaciones de este tipo haya (sólo una de las transiciones se va a lleva a cabo: la que cumpla la condición pre del arco). Veamos con más detalle cada uno de los tipos de valuaciones que se pueden presentar:

1. Una única valuación para una acción sin condición para la aplicación de la misma. Es decir, la línea de especificación OASIS tiene la forma: [acción] post.

La precondición de aplicación de la valuación se considera true. El subproceso correspondiente es el que se muestra en la Figura 2.

2. Una única valuación para una acción con condición para la aplicación de la misma. El subproceso debe incluir las posibilidades de aplicación y no-aplicación de la valuación pero nunca debe impedir que se lleve a cabo dicha acción. Es decir, la línea de especificación OASIS tiene la siguiente forma: pre [acción] post.

El subproceso, en el que destacamos que se incluye el poder transitar frente a Not pre que no tiene efectos sobre los atributos (true), es el que se muestra en la Figura 3.

La condición exigida de que las precondiciones para una misma acción desde el mismo estado fuesen excluyentes es obvio que se cumple en dicho ejemplo.

3. Distintas valuaciones para la misma acción. Se requiere que las precondiciones de aplicación de cada valuación sean disjuntas entre sí para poder representarlo con la semántica que hemos precisado (lo cual siempre se puede conseguir, de forma más o menos directa). La porción de especificación OASIS, en la que se ha simplificado a dos valuaciones, podría ser de la forma:

pre1 [acción] post1
pre2 [acción] post2

El subproceso de la Figura 4 representa las dos valuaciones para la acción (una cuando pre1 es cierto; la otra cuando lo sea pre2), donde destacamos que volvemos a considerar el

caso en el que no se aplica ninguna de ellas (not pre1 **and** not pre2) sin postcondición asociada.

Precondiciones. Una precondición es una condición que debe cumplir una acción para que pueda ejecutarse. Dada la precondición OASIS:

acción **if** condición

El subproceso correspondiente es el de la Figura 5.

La inclusión del subproceso asociado a una precondición supone restringir el número de transiciones posibles pues si pre no se cumple entonces no es posible que ningún otro subproceso que tenga a acción como arco tipo "input" pueda transitar. Es decir, se realiza en todos los subprocesos o en ninguno de ellos.

Triggers. Los triggers de OASIS expresan obligación asociada al estado en el que se encuentra el objeto. La inclusión de actividad por parte de un objeto implica que pueda realizar acciones como consecuencia directa de su estado. Un trigger en OASIS tiene la forma:

acción **when** condición

El servidor de la acción puede ser el propio objeto (caso de que el servidor sea self) o bien cualquier otro objeto de la sociedad. En el objeto que se cumple la condición del trigger debe también acontecer la petición del mismo, pues, tal como se distingue en [Let97], una cosa es la acción que se solicita y otra muy distinta la solicitud de la acción. Distinguimos[4] a nivel sintáctico de OASIS mediante "::" entre ambas acciones. En resumen, dentro del mecanismo asociado a un trigger diferenciamos tres partes:

1. El envío de forma asíncrona de la acción indicada en el trigger.
2. El envío de forma síncrona del evento prefijado con "::" que representa que al objeto le ocurre el lanzamiento del trigger.
3. La valuación (que puede no existir) asociada a la recepción y ejecución de la acción del punto anterior.

Para cada parte se asocia un subproceso distinto. El subproceso para el envío asíncrono es el de la Figura 6. Se ha utilizado un arco tipo "output" con asincronía. La acción es de salida hacia el servidor de dicha acción (que puede ser el mismo objeto, caso de que el trigger sea a self).

El subproceso para el envío síncrono se incluye en la Figura 7. Mediante ::acción nos referimos a la acción enviada al propio objeto y el tipo de arco indica que se atenderá en el siguiente paso.

La Figura 8 representa el subproceso asociado a una posible valuación del evento ::acción. Es importante la utilidad de estas valuaciones pues permiten implementar el mecanismo de desactivación de un trigger garantizando, por ejemplo, que sólo se dispare una vez.

Protocolos. Las precondiciones permiten la ejecución de acciones en determinados estados del objeto (aquellos en los que se cumplía la condición) prohibiendo dicha ejecución en caso contrario. Los protocolos de OASIS nos van a permitir implementar la noción de prohibición para la parte de proceso. Asumimos que dichos protocolos establecen secuencias de ejecución permitidas sólo de las acciones incluidas en dicho protocolo (pero no se dice nada del resto). Un objeto puede tener varios protocolos definidos. Asumimos además que todos los protocolos están activados desde el comienzo de la vida del objeto. Utilizando los operadores del álgebra de procesos, tales como secuencia (".") y selección ("+"), y permitiendo definir protocolos de forma jerárquica podemos establecer protocolos para la vida de un objeto OASIS. Veamos un protocolo en el que se deben alternar las acciones a y b y el proceso no finaliza[5] nunca:

```
example_protocol.
example_protocol_1=a. example_protocol_2;
example_protocol_2=b.example_protocol_1;
```

Este protocolo fija que las ejecuciones de a y b se lleven a cabo según secuencias del tipo, (ab, abab, ...). La Figura 9 muestra un grafo que representa los caminos posibles según el protocolo y cuyo estado inicial es el marcado.

Para cada protocolo que se defina incluiremos dos tipos de subprocesos. Un primer tipo para representar la evolución en el grafo. De este primer tipo sólo es necesario un subproceso en el que los estados y los arcos poseen la misma estructura que el grafo del que provienen. El segundo tipo es para incluir la prohibición implícita que representa el protocolo. Se considera un subproceso de este tipo por cada acción afectada por el protocolo. El primer subproceso para el ejemplo anterior es el de la Figura 10. Las Figuras 11 y 12 establecen las prohibiciones asociadas a las acciones a y b, respectivamente.

La evolución de los tres subprocesos de forma simultánea conlleva que el estado de partida es tal que se puede hacer a y no b (por no cumplirse la precondición de transición). Tras hacerse a, el diagrama example_protocol pasa al estado After_a donde no es posible hacer a sino b. Esta evolución simultánea de los distintos diagramas es fundamental para entender la semántica asociada.

---

[4] Especialmente importante en el caso de que el objeto cliente sea también servidor del evento.

Operaciones. Mediante las operaciones establecemos secuencias obligadas de ejecución de acciones. La sintaxis en OASIS es la misma que para los protocolos pero el significado cambia. A diferencia de los protocolos, las operaciones no se consideran activas desde la creación del objeto. Es necesaria la solicitud del servicio correspondiente para adquirir la obligación. Dicho servicio es el que etiqueta el principio de la operación y forma parte de la signatura de acciones del objeto. La semántica asociada a los protocolos establece secuencias obligadas de acciones que deben ocurrir en el objeto. Consideremos a modo de ejemplo una operación OP que representa la ejecución secuencial y obligada de las acciones a y b.

En OASIS tendríamos:

```
operations
OP.
    OP_1= a . OP_2;
    OP_2= b;
```

Se destaca que sería posible prefijar con fórmulas sobre el estado y discriminar la transición en el grafo de proceso que representa. El diagrama de la Figura 14 muestra la operación OP mediante el uso del arco de sincronía para el siguiente paso. La llegada de OP conlleva el envío a **self** del evento a (si no se especifica servidor asumimos el propio objeto). El estado al que se llega es tal que se produce el envío a self del evento b. Las precondiciones son true en ambos casos indicando que no es necesario discriminar el arco por el que se producirá la transición (situación que se da cuando del mismo estado sale más de un arco para la misma acción).

5 Conclusión, trabajo futuro.

Hemos visto la conveniencia de encontrar una representación gráfica de la funcionalidad de OASIS si queremos expresar formalmente el proceso de representación a otros modelos de concurrencia. Nuestro trabajo consiste en los pasos: (1) formalizar el proceso de representación de OASIS en los diagramas presentados, (2) encontrar las representaciones que trasladen de estos diagramas a redes de Petri de alto nivel tales como las *Object Oriented Timed Petri Nets (OOTPN)* [Ess96], y (3) justificar que la representación de OASIS en dichas redes preserva la semántica definida, tal como se introduce en [Sán97b]. La justificación de que la semántica de un objeto OASIS es equivalente a otra en redes de Petri puede hacerse en tres fases:

[5] La finalización contendría la constante agente **0** indicando que no hay continuación.

1. Demostrando que la semántica de OASIS definida en lógica dinámica es equivalente a la semántica de los diagramas de subprocesos.
2. Demostrando que dicha semántica de subprocesos es equivalente a la semántica asociada a las redes de Petri OOTPN [Ess96] obtenidas. Esto puede hacerse para cada tipo de subproceso viendo que los marcados alcanzables por la red de Petri del subproceso son los correctos. Será necesario además estudiar la equivalencia global de la red de Petri incluyendo la parte relativa al protocolo de dos fases, completándose la semántica asociada a un objeto OASIS.
3. Demostrar que la equivalencia semántica se preserva tras el cambio de estado del objeto, tal como se propone en [Sán97b].

Las ventajas de traducir de OASIS a los diagramas de subprocesos se resumen en las siguientes:

1. Es más fácil demostrar que una red de Petri tiene un comportamiento equivalente a un subproceso que a la especificación completa de una clase OASIS.
2. Si la sintaxis del lenguaje OASIS cambia por nuevas versiones del mismo dichos cambios no tienen efecto sobre el proceso de traducción si la semántica del lenguaje OASIS se preserva.

6 Ejemplo

Una máquina expendedora de chocolatinas (vm) proporciona una chocolatina (choc) por cada moneda introducida (coin_in). El máximo de monedas que puede retener (credit) antes de que se solicite la chocolatina es de tres. Si no hay suficiente stock (num_chocs) como para satisfacer la demanda según monedas (credit), entonces se enciende una luz de vacía (light_empty) y el usuario debe pulsar cancel para que le sean devueltas las monedas (coin_out). La acción cancel también puede ser usada para interrumpir el proceso. Para parte de la especificación OASIS se aporta la representación gráfica. La evolución de cualquier objeto de dicha especificación OASIS es equivalente a la evolución simultánea de los distintos subprocesos definidos para el objeto. Las Figuras 14 a 19 muestran los subprocesos asociados a este ejemplo.

class vm
identification
by_number: (number);
constant_attributes
number: **nat(**0);
variable_attributes
num_chocs: **nat**; credit: **nat**;
switch_on **boolean(false);**
events
set **new**; coin_in; coin_out; choc; light_empty;
operations
CANCEL.
$CANCEL_1$={credit>1}::coin_out.$CANCEL_1$
+ {credit<=1} ::coin_out;
valuations
[coin_in] credit:=credit+1;
[coin_out] credit:=credit-1;
[choc] nchocs:=nchocs-1 **and** credit:=credit-1;
[::light_empty] switch_on:=true;
preconditions
choc **if** credit>0 **and** nchocs>0;
CANCEL **if** credit>0;
triggers
self::light_empty **when** nchocs=0
and switch_on=false;
patterns
GETCHOC.
$GETCHOC_1$ = coin_in.$GETCHOC_2$;
$GETCHOC_2$=choc.$GETCHOC_1$+ coin_in.$GETCHOC_3$ + coin_out.$GETCHOC_1$;
$GETCHOC_3$=choc.$GETCHOC_2$+ coin_in.$GETCHOC_4$ + coin_out.$GETCHOC_2$;
$GETCHOC_4$=choc.$GETCHOC_3$+coin_out.$GETCHOC_3$;
end class

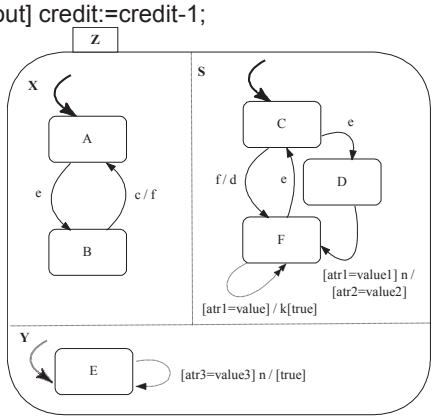


Fig. 1: Ejemplo de subproceso.

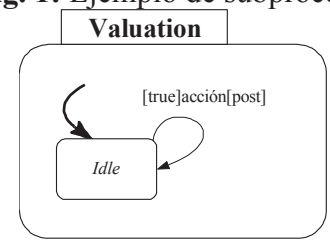


Fig. 2: Valuación Simple

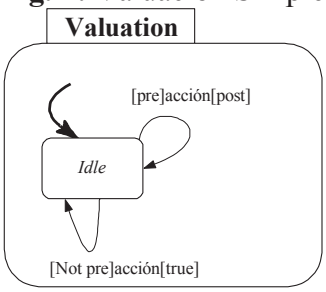


Fig. 3: Valuación simple.

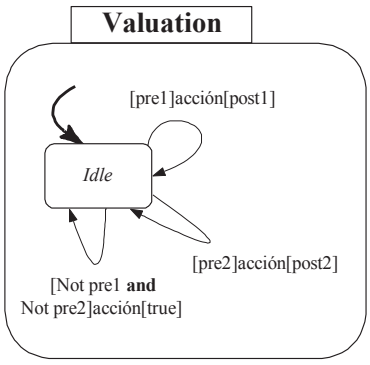


Fig. 4: Valuación múltiple.

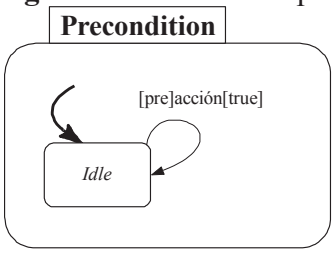


Fig. 5: Precondición.

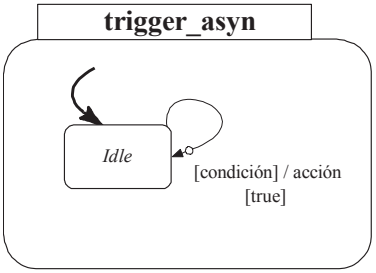


Fig. 6: Envío asíncrono.

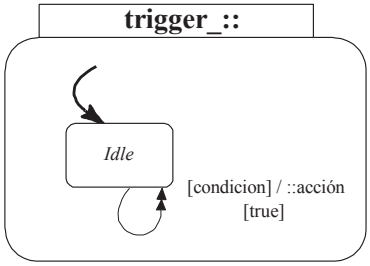


Fig. 7: Envío de ::acción .

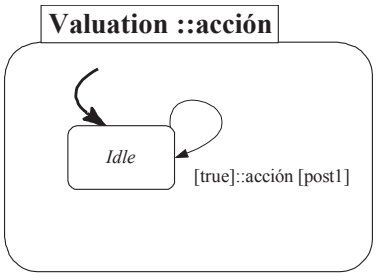


Fig. 8: Valuación de ::acción.

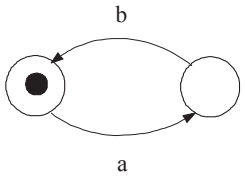


Fig. 9: Grafo asociado al protocolo.

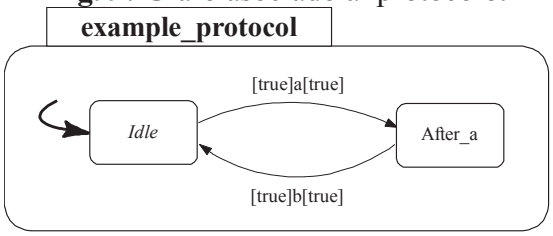


Fig. 10: Evolución protocolo.

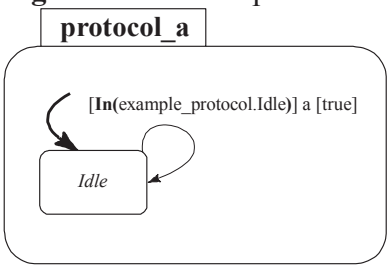


Fig. 11: Prohibición de a.

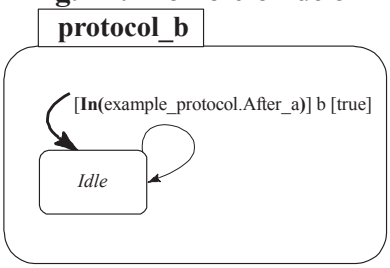


Fig. 12: Prohibición de b.

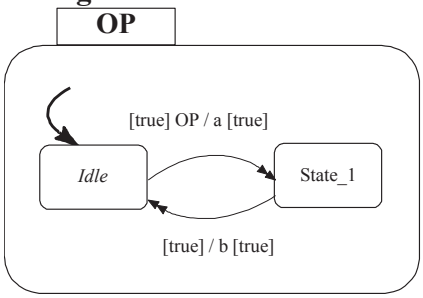


Fig. 13: Diagrama de obligación de OP.

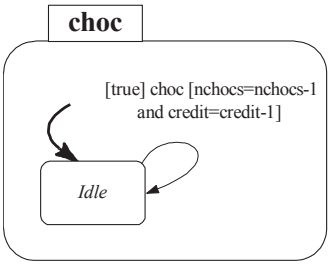


Fig. 14: Evaluación por choc.

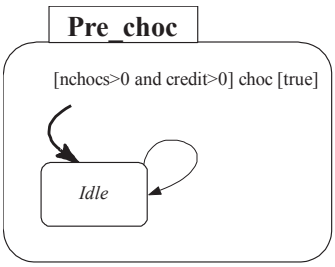


Fig. 15: Precondición de choc.

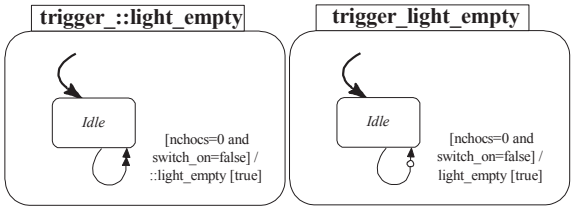


Fig. 16: Trigger light_empty.

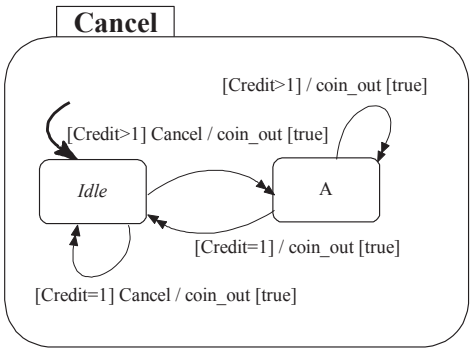


Fig. 17: Obligación por Cancel.

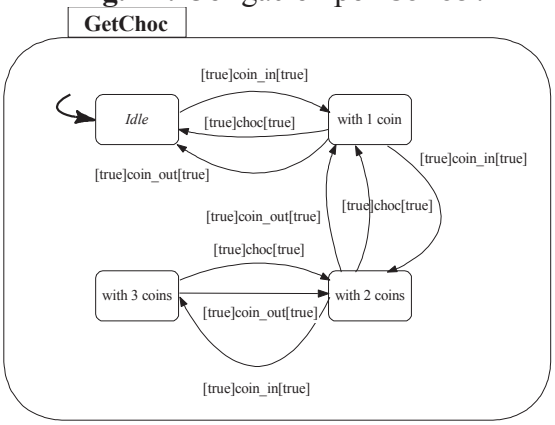


Fig. 18: Protocolo Getchoc.

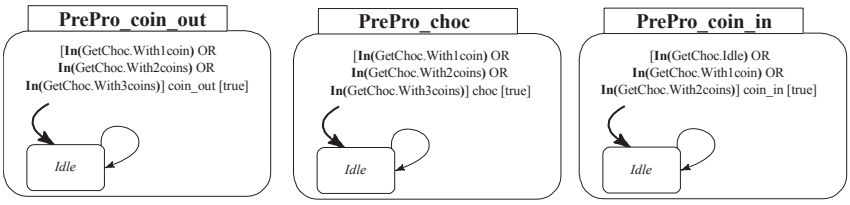


Fig. 19: Protocolo Getchoc.

Referencias

[Col92] Coleman D., *Statecharts in Object-Oriented Design*, IEEE Transactions on Software Engineering Hayes F., Bear S., *Introducing Objectcharts or How to Use*, Vol. 18, 1992.

[Ess96] Esser R., *An Object Oriented Petri Net Approach to Embedded System Design*, PhD Thesis, Swiss Federal Institute of Technology Zurich, 1996.

[Har88] Harel D., *On Visual Formalisms*, Communications of the ACM, Vol 31, N.5, 1988.

[Jun95] Jungclaus R., Saake G., Hartmann T., Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems*, ACM Transactions on Information Systems, Volume 14, Number 2, Pages 175-211, 1995.

[Let97] Letelier P., Sánchez P., Ramos I., *Animación de Modelos Conceptuales para la validación de requisitos*, Argentine Symposium on OO, Buenos Aires, pp.103-110, 1997

[Pas95] Pastor O., Ramos I. OASIS versión 2 (2.2) : *A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.

[Ram93] Ramos I., Pastor O., Cuevas J., Devesa J., *Objects as Observable Processes*, Actas del 3rd. Workshop on the Deductive Approach to Information System Design, Cataluña, 1993.

[Sán97a] Sánchez P., Letelier P., Ramos I., *Constructs for Prototyping Information Systems with Object Petri Nets*, IEEE SMC'97, Orlando, pp. 4260-4265, 1997.

[Sán97b] Sánchez P., Letelier P., Ramos I., *Representación de Objetos como Teorías y Traducción a un Modelo de Concurrencia*, Revista *Computación y Sistemas*, Vol. I, N.2, México, Octubre-Diciembre, 1997.

[Ser92] Sernadas C., Gouveia P., Sernadas A., *Oblog: Object-Oriented, logic-based conceptual modelling*. Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, 1096 Lisboa, 1992.

[Ser94] Sernadas C., Ramos J. *Gnome: Sintaxe, semântica e cálculo*. Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, Lisboa, Portugal, 1994.

Propuesta de Interfaz Gráfica de Usuario homogénea para el diseño y la explotación de Sistemas de Información♦

José A. Carsí, Sergio Camilleri, José H. Canós, Isidro Ramos
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Camí de Vera, s/n, 46071 València
☎ 3877350; Fax: 3877357
e_mail { pcarsi }@dsic.upv.es

Resumen

En este artículo se propone un modelo de interfaz gráfica de usuario para una herramienta CASE que permite modelar y animar sistemas de una forma homogénea gracias a la noción de metaclase que se incluye en el modelo formal OASIS. La interfaz forma parte de un entorno de prototipación automático que da soporte a la especificación, ejecución por animación y evolución de sistemas que está siendo desarrollado dentro del proyecto Menhir.

1 Introducción

En el proceso de construcción de una herramienta CASE que abarque el ciclo completo (desde el análisis al mantenimiento) de la vida de las aplicaciones software, que permita la construcción de aplicaciones con altas cotas de calidad y facilite la reutilización y la evolución de los sistemas, una de las partes más importantes es el diseño de la interfaz gráfica de usuario (IGU) que permita a los usuarios la definición y explotación de las aplicaciones de forma cómoda y amigable.

En [Car96, Car98a, Car98b] se ha presentado la *metaclase OASIS* como el núcleo de un entorno de prototipación automática que proporciona un marco de trabajo, utilizando una metáfora única para desarrollar, validar y modificar modelos de sistemas de información, así como para trabajar con los mismos.

Hasta el momento, el trabajo en el entorno se realizaba de forma textual en el que era difícil trabajar debido a la necesidad de conocer detalles del sistema para poder realizar acciones (hay que conocer los atributos, los eventos, el número y tipo de los argumentos, ...), existe poca o nula retroalimentación (para saber que algo ha pasado es necesario realizar una consulta para comprobar el nuevo estado), etc.

En el presente artículo se presenta una propuesta de IGU para facilitar el uso de la herramienta, coherente con el modelo OO basado en la metaclase OASIS. Centrándonos en la idea de que definir sistemas de información haciendo uso de la metaclase es exactamente lo mismo que trabajar con los esquemas conceptuales que definen los sistemas de información, presentamos un IGU homogéneo que utiliza el diagrama de configuración de clases (DCC)

♦ Este trabajo ha sido financiado con fondos de CICYT TIC97-0593-C05-01

Carsí, J. A., Camilleri, S., Canós, J.H., Ramos, I. Propuesta de Interfaz Gráfica de Usuario homogénea para el diseño y la explotación de Sistemas de Información. Actas de las III Jornadas de Ingeniería del Software. 1998.

como base para la presentación. El ambiente gráfico presentado va a poder ser utilizado tanto para la definición como para la explotación.

Herramientas CASE existentes (*System Architect* [Pop98], *Rational Rose* [Rat98], *OO-Method* [Pas97], etc.), utilizan en la definición de los sistemas, diferentes modelos gráficos para la introducción de la información: modelo de objetos, modelo dinámico, modelo funcional y/o modelos de interacción. El modelo de IGU que presentamos en este artículo puede verse como una alternativa, no excluyente, a dichos modelos. El usuario/diseñador podría trabajar con cualquiera de los modelos gráficos con los que se sienta más cómodo aunque internamente estaría trabajando con los servicios que le proporciona la metaclase.

El documento está estructurado de la siguiente manera: tras la introducción, en el segundo apartado se da un repaso al concepto de metaclase y las ventajas que aporta. En el tercer apartado se presenta la arquitectura del sistema que se está construyendo. En el cuarto apartado se presenta la interfaz gráfica. Por último, en el quinto apartado se presentan las conclusiones y trabajos futuros.

2 La Metaclase OASIS

La metaclase es el corazón de un entorno CASE-OO que permite realizar el análisis y el diseño de los sistemas de información orientados a objetos, siguiendo la visión aportada por OASIS.

La metaclase OASIS es una clase compuesta que recoge todas las características del modelo orientado a objetos OASIS [Pas96] y cuyas instancias, llamadas meta-objetos o meta-instancias, son objetos especiales que, a su vez, son clases de objetos. Todos los objetos son instancias de alguna clase y las clases (vistas como objetos) son a su vez instancia de la metaclase. Para cerrar el ciclo y no tener una jerarquía potencialmente infinita de metaclases se realiza el cierre reflexivo de la metaclase, diciendo que es instancia de sí misma. De esta manera, las clases son objetos de primera clase y pueden ser consultadas y manipuladas de la misma manera que los objetos.

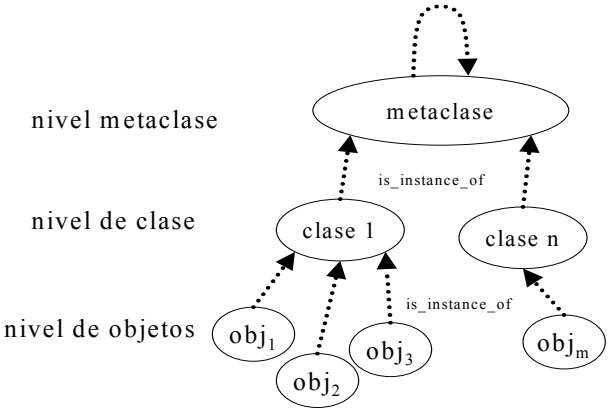


Figura 1: Jerarquía *is_instance_of* en la metaclase.

OASIS proporciona tres unidades de diseño: las clases primitivas, las clases elementales y las clases complejas. La metaclase permite al usuario construir clases de cualquiera de estas tres categorías, es decir, la metaclase ofrece los servicios de un editor de clases primitivas, un editor de clases elementales y un editor de clases complejas (ver Figura 7) a través de cuyos servicios (crear_clase, añadir_atributo, ...) se definen las clases que forman los esquemas conceptuales.

Una de las principales características de la metaclase consiste en que sus instancias no son sólo objetos con un estado definido por el valor que toman determinados atributos y una dinámica ofrecida en forma de servicios, sino que, además, son clases de objetos. En la Figura 2 podemos ver una instancia de la metaclase (la clase libro) en su doble vertiente clase/objeto.

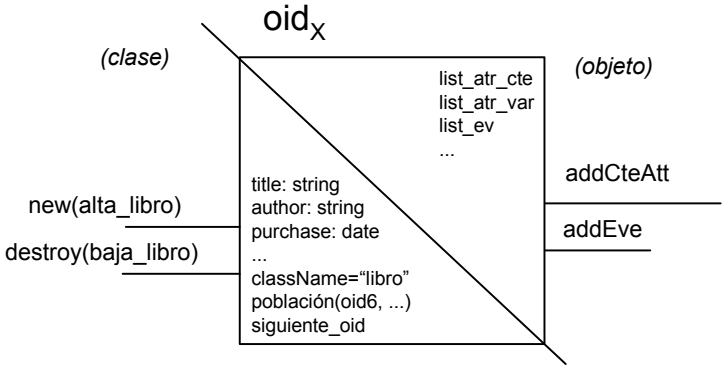


Figura 2: Doble visión de las instancias de la metaclase.

Debido a su doble visión, otra característica importante de la metaclase estriba en que los aspectos de evolución del esquema están totalmente integrados en la misma[1]. Esto es así, porque el disparo de, por ejemplo, el evento que añade un atributo a una clase, debe provocar la adecuación de todos los objetos involucrados (instancias de la clase modificada) para ser consistentes con el nuevo estado (migración de objetos).

El entorno resultante tiene una doble perspectiva. Por un lado, es un entorno de producción de software orientado a objetos y, por lo tanto, soporta entre otras características la facilidad de crear nuevas clases. Por otro lado, está basado en el paradigma de la programación automática y permite el prototipado rápido[2].

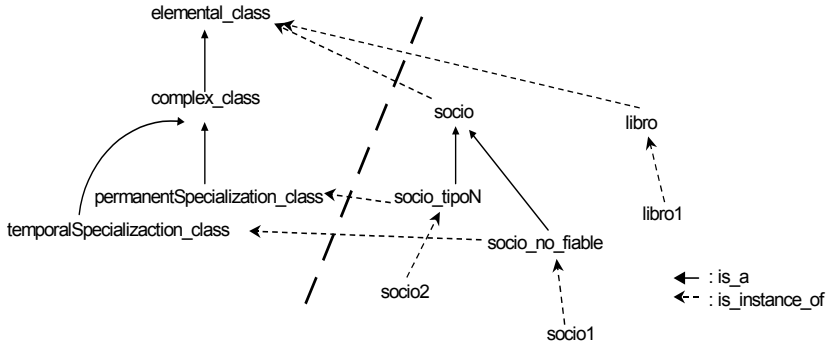


Figura 3: Estado del esquema conceptual de la gestión de una biblioteca en un instante dado.

Una vez el usuario se identifique en el sistema y asuma el papel de alguno de los objetos activos, podrá empezar a definir el esquema conceptual de algún sistema de información haciendo uso de los servicios proporcionados por la metaclase. En la Figura 3 se puede observar un posible estado alcanzado tras haber creado algunas clases del sistema de información de una biblioteca.

[1] Dado que un esquema se representa como el estado de una instancia de la metaclase.

[2] OASIS tiene unas semánticas declarativa y operacional bien definidas.

Una primera implementación del concepto de metaclase se encuentra en avanzado estado de desarrollo y puede ser probada en la dirección: http://www.dsic.upv.es/~pcarsi/tfl.

3 Arquitectura

Las herramientas usadas en la implementación de la IGU de definición y explotación de sistemas de información son básicamente dos: el lenguaje *Tcl&Tk* [Wel97] con el que se ha desarrollado el IG y el intérprete de *Transaction-Frame Logic* [Car97] que se encarga de la animación de la metaclase OASIS.

Tcl (Tool command language) es realmente un lenguaje de *scripts* y un intérprete para ese lenguaje diseñado para ser embebido fácilmente en cualquier aplicación. *Tk* (Toolkit) es una de las extensiones más notables de *Tcl*, que añade al núcleo de éste comandos adicionales para el diseño de interfaces gráficas de usuario de una forma rápida y sencilla. Tanto *Tcl* como *Tk* están desarrollados como librerías de procedimientos en C, pudiendo extender las facilidades básicas de *Tcl&Tk* con nuevos comandos *Tcl* y nuevos comandos gráficos escritos en C. Además, el lenguaje es independiente de plataforma, por lo que puede ser utilizado indistintamente en entornos UNIX, Windows o Macintosh.

La *Transaction-Frame Logic* (TFL) es un formalismo lógico recientemente propuesto que proporciona un modelo formal para los lenguajes y las bases de datos orientadas a objetos. De manera similar a los lenguajes imperativos, en *TFL* se pueden usar operadores secuenciales para especificar transacciones componiendo acciones simples en otras más complejas de forma declarativa.

La arquitectura del programa, se puede observar en la Figura 4. La IG se divide básicamente en dos módulos:

- El módulo gestor de la interfaz
- El módulo gestor de las consultas

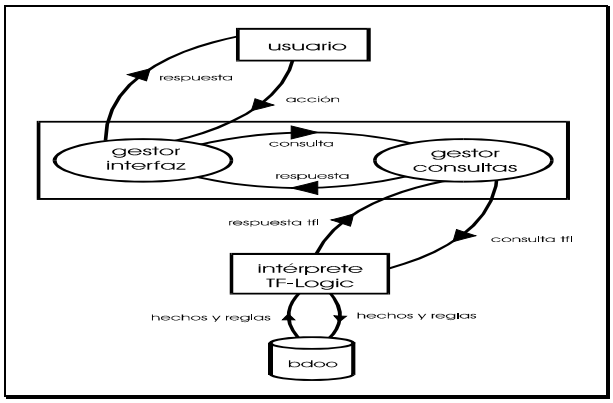


Figura 4: Arquitectura Software

El módulo gestor de la interfaz se comunica con el usuario y con el módulo gestor de consultas. El usuario realiza peticiones al programa a través de su interfaz; estas peticiones pueden requerir información externa o no. Por ejemplo, cuando el usuario quiere ver la información de una clase ha de pedírsela al intérprete a través del gestor de consultas que se encarga de comunicarse con el intérprete y de devolver la información requerida.

El módulo gestor de las consultas recibe peticiones de información del módulo gestor de la interfaz. Estas peticiones son complejas y necesitan ser traducidas a un nivel comprensible por el intérprete. Por ejemplo, la petición de consultar la plantilla de una clase puede dividirse un múltiples peticiones al intérprete acerca del nombre de la clase, sus atributos constantes, variables y derivados, sus eventos propios y compartidos, etc. Este módulo procesa las respuestas del intérprete y les da formato antes de enviarlas al módulo gestor de la interfaz.

En lo que respecta al intérprete, éste se ejecuta en segundo plano utilizando las conocidas *pipe-lines* de UNIX. El intérprete y la aplicación comparten un canal de lectura y escritura a través del cual intercambian información.

4 Ambiente Gráfico

Se va a presentar el ambiente gráfico desarrollado a través de ejemplos que ilustren la forma de trabajar con el mismo. Describiremos una secuencia de acciones que podría desencadenar el usuario una vez inmerso en la sociedad de objetos que aparece ante él.

La primera pantalla que muestra la aplicación presenta el conjunto de esquemas conceptuales disponibles en el estado actual representados en forma de iconos tal y como se puede ver en la Figura 5. Para utilizar un esquema conceptual, bien sea para editarlo o bien para ejecutarlo, no hay más que seleccionarlo con el ratón.

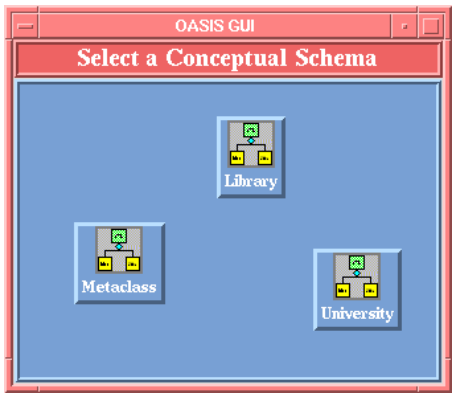


Figura 5: Pantalla Inicial (Esquemas Conceptuales disponibles)

Una vez seleccionado el esquema conceptual aparece un diálogo para la identificación del usuario que desea utilizar dicho esquema. Este diálogo, que aparece representado en la Figura 6, consta de tres campos que ha de rellenar el usuario para poder acceder al esquema. Los usuarios están divididos en distintos grupos, por lo que primeramente se ha de identificar el grupo al que pertenece el usuario, grupo que en nuestro caso se corresponde con una clase OASIS. El control utilizado para la clase de usuario posee una lista de las clases de usuario que pueden acceder al esquema. En segundo lugar, se ha de identificar el usuario con su *login*, que en este caso se corresponde con el identificador del objeto de la clase elegida que representa al usuario. En último lugar, se introduce el *password*, que se compara con el valor del atributo variable *password* del objeto que representa al usuario.

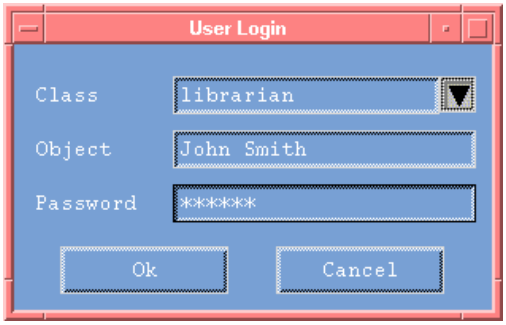


Figura 6: Identificación del usuario

Esta identificación protege a los esquemas conceptuales de usos indebidos por parte de usuarios no autorizados y pone en marcha el mecanismo de vistas que discrimina la visión del usuario conectado sobre la sociedad, es decir, qué clases son visibles para él y qué servicios puede solicitar.

En la sesión de ejemplo, el usuario se propone trabajar sobre la propia metaclase por lo que, tras seleccionar el esquema conceptual iconizado que representa a la metaclase, aparece una nueva ventana como la mostrada en la Figura 7. Esta ventana consta de una barra de herramientas, cuyos servicios pasaremos a describir en breve, un *canvas* sobre el que se representa el DCC correspondiente al esquema conceptual que se está utilizando y una barra de estado que proporciona al usuario información sobre lo que está pasando.

El DCC permite mostrar al usuario de un solo golpe de vista qué clases componen la clase compleja que es el esquema conceptual, así como qué relaciones existen entre ellas.

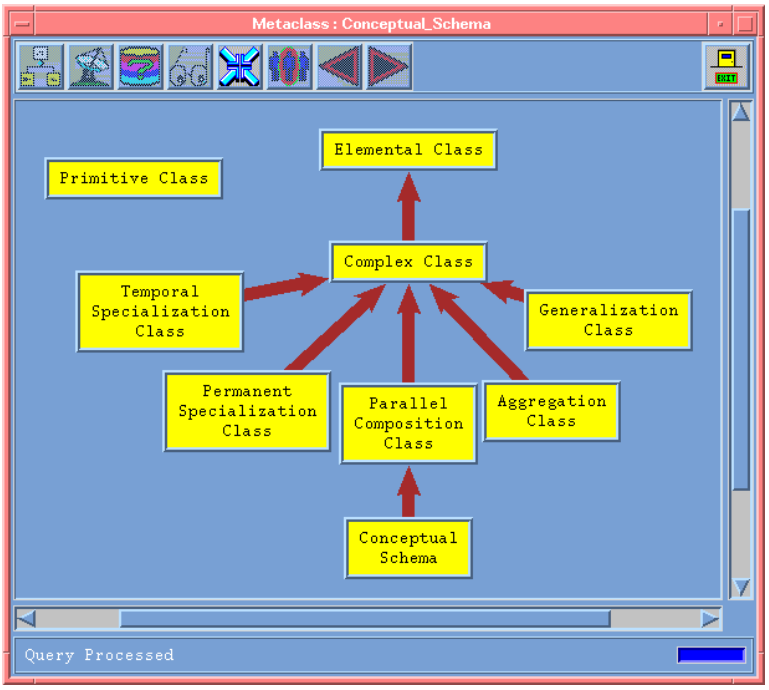


Figura 7: DCC de la metaclase OASIS

El DCC muestra las distintas clases que forman el esquema conceptual de la metaclase OASIS. En él podemos crear nuevas clases y modificar las existentes. Para trabajar con una clase del DCC no tenemos más que hacer doble click sobre el rectángulo que la representa. Las clases del DCC pueden, además, desplazarse por el *canvas* para mayor comodidad del usuario.

Supongamos que el usuario desea trabajar con la clase *Elemental_Class*, para lo cual hará doble click sobre el rectángulo que la representa. Existe otro modo de seleccionar una clase del DCC a través del servicio  presente en la barra de herramientas, el cual muestra una lista con las clases que forman el DCC que se está visualizando. Una vez seleccionada la clase, la pantalla tendrá el aspecto mostrado en la Figura 8.

El usuario puede seleccionar cualquier objeto de la población de la clase seleccionada, por ejemplo, el objeto socio (*Member*), instancia de *Elemental_Class*, para lo cual hará doble click sobre el elemento de la población correspondiente (ver Figura 9). Existe otro servicio en la barra de herramientas que permite ver una instancia de la clase visualizada. Ese servicio viene representado por  y, como el anterior, muestra una lista equivalente a la población de la clase visualizada. La ventaja que presenta este servicio es que permanece accesible aunque estemos visualizando la visión objetual de la instancia.



Figura 8: Clase *Elemental_Class*

La representación gráfica de los metaobjetos viene dada por la doble visión comentada en el punto 2. Los metaobjetos pueden verse como objetos o como clases de objetos. Todas las instancias de la metaclase tienen esa doble visión. Sin embargo, las instancias de los metaobjetos (las clases) sólo poseen la visión objetual.

En la Figura 9 y la Figura 10 se puede observar la visión de clase y la visión objetual de la clase socio (*Member*), instancia de la clase *Elemental_Class*.

Una clase OASIS está formada por un nombre, una plantilla, una función de identificación, una factoría y un conjunto de objetos llamados la población de la clase. La visión de clase de una meta-instancia consiste en un rectángulo dividido en cuatro zonas bien diferenciadas, de las que tres se corresponden con los elementos de la definición anterior.

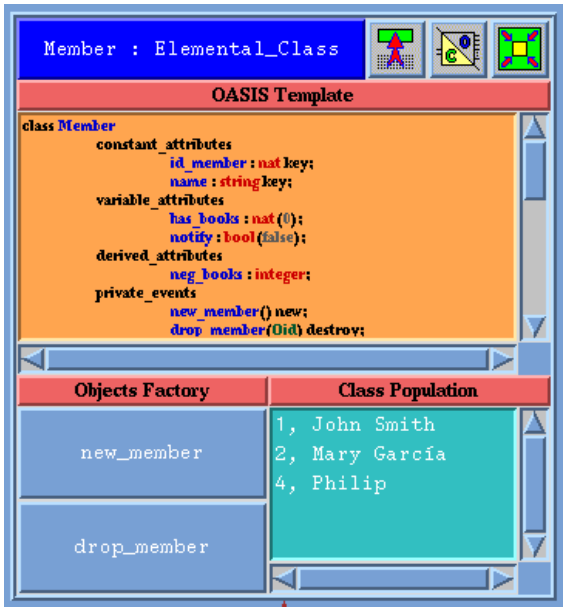


Figura 9: Visión de clase de la instancia *Member*

- En la zona superior hay una etiqueta para el identificador de la instancia y tres botones cuyo funcionamiento detallaremos en breve. Tras el identificador se muestra la clase de la que el objeto es instancia. En el ejemplo de la Figura 9 el identificador de la instancia es *Member*. El primer botón,  , situado a la derecha del nombre permite visualizar la clase de la cual desciende la instancia actual. El segundo botón,  , que indica la visión actual seleccionada[3], sirve para pasar de la visión de clase a la

objetual y viceversa. El icono de este botón es una representación de la Figura 2 con la parte de la clase resaltada. Por último, el tercer botón,  , minimiza el rectángulo.

- En la zona central aparece la plantilla de objetos**,** es decir, la especificación OASIS de la clase que se está visualizando. La plantilla muestra con diferentes colores los distintos elementos presentes en ella (tipos, nombres, palabras clave, fórmulas, …) para una mayor claridad y una asimilación más rápida.
- La zona inferior izquierda muestra la factoría de objetos, es decir, los servicios de creación y destrucción de instancias. Cuando se solicita alguno de estos servicios aparece un diálogo en el que se establece el valor de los parámetros, que para el caso del evento creador consiste en el valor de los atributos constantes de la clase, y para el evento destructor el identificador de la instancia a destruir.
- Por último, la zona inferior derecha muestra el conjunto de objetos llamados la población de la clase. La población de la clase viene dada en forma de lista, mostrando los nombres (identificadores) de las instancias de la clase. La solicitud del evento *destr*oy utiliza el objeto seleccionado en la población. Las acciones destructivas han de ser confirmadas por el usuario, por lo que en este caso se mostrará un diálogo para confirmar su destrucción. Además, la población sirve como enlace para visualizar la instancia seleccionada haciendo doble click sobre ella.

La visión objetual de una instancia consiste al igual que la de clase en un rectángulo dividido en tres zonas:

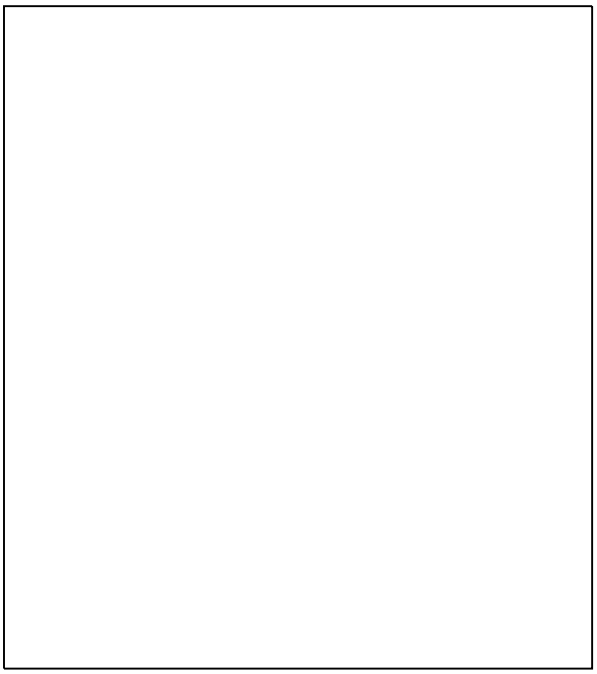


Figura 10: Visión objetual de la instancia *Member*

- La zona superior es exactamente igual a la de la visión de clase exceptuando el hecho de que el icono del botón de cambio de visión resalta en este caso la parte de la instancia. 
- La zona central se utiliza para mostrar el estado de la instancia o, lo que es lo mismo, sus atributos. Los nombres de los atributos aparecen junto a sus valores en una rejilla con dos columnas.
- Por último, en la parte inferior se muestran los servicios que ofrece la instancia. Estos servicios son los eventos y transacciones definidos en la signatura de la clase de la cual es instancia el objeto.

Supongamos que el usuario desea trabajar con el esquema conceptual Biblioteca (*Library*), al que pertenece la clase libro. Para que aparezca dicho esquema conceptual podemos volver a la ventana inicial (Figura 5) y seleccionar el esquema tal y como se hizo con la metaclase. Otra forma de acceder a dicho

esquema consiste en seleccionar la clase *Conceptual_Schema*, tal y como se hizo con *Elemental_Class*, y mostrar la instancia *Library* perteneciente a su población. Una vez en esta situación, mostrada en la Figura 11, sólo queda escoger la biblioteca sobre la que se quiere trabajar (en el ejemplo la *UPV-Library* o la *UM-Library*) y pulsar sobre el botón  para ver el DCC.

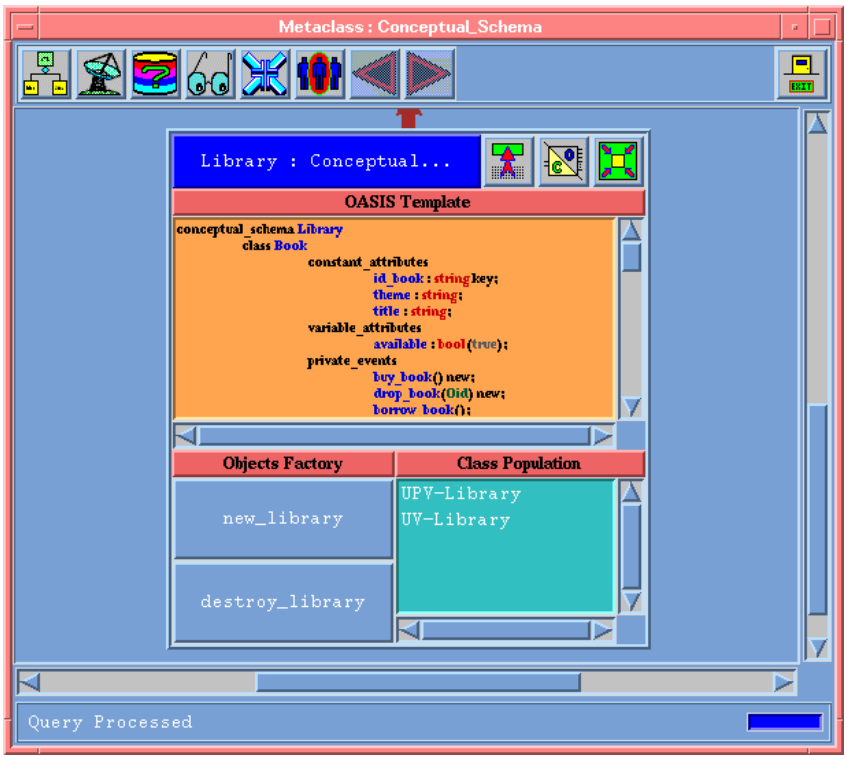


Figura 11: Visión de clase del esquema conceptual *Library*

En la Figura 12 se puede observar el DCC de una biblioteca (*Library*). El aspecto de la ventana es similar al de la Figura 7, exceptuando, claro está, el DCC mostrado en el *canvas* y el título de la ventana. Este esquema está muy simplificado, y consta de libros (*Book*) y socios (*Member*), relacionados entre sí a través de préstamos (*Loan*).

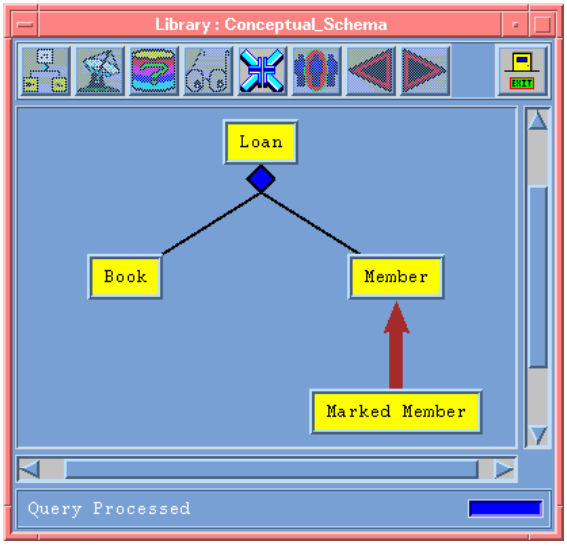


Figura 12: Esquema Conceptual *Library*

Si el usuario selecciona la clase libro, ésta mostrará su visión de clase, tal y como aparece en la Figura 13.

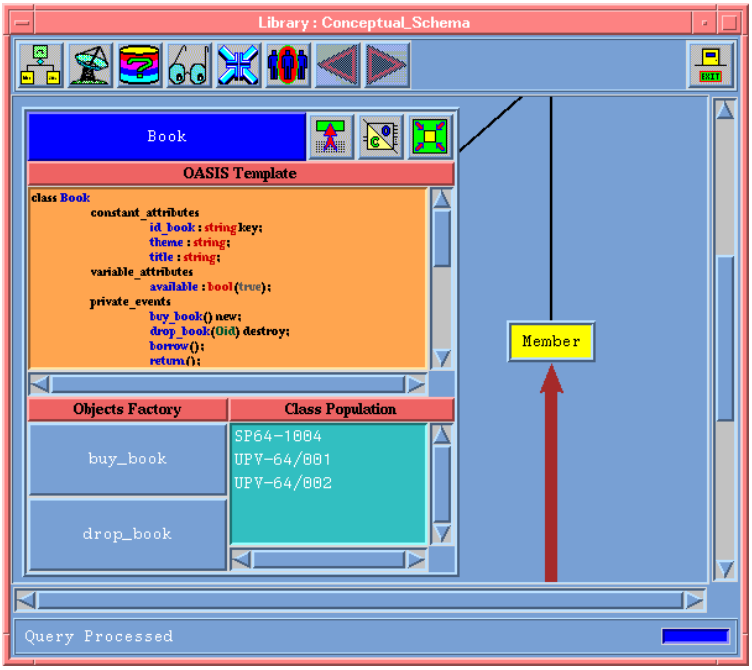


Figura 13: Visión de clase de la instancia *Book* en el esquema *Library*

Supongamos ahora que el usuario quiere consultar la instancia de libro que aparece en primera posición en la lista de la población. Para ello hará doble click sobre dicho elemento, tras lo cual se encontrará en la situación mostrada en la Figura 14.

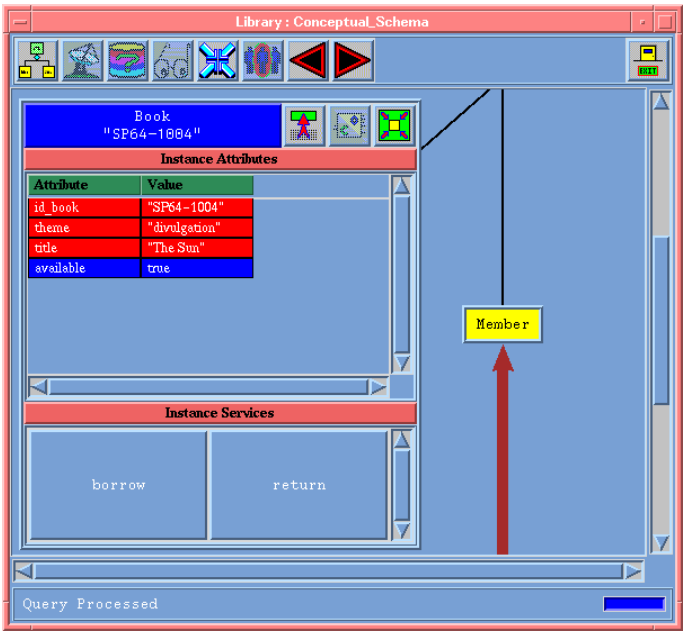


Figura 14: Visión de una instancia de la clase *Book*

Como se puede observar en la figura anterior, los servicios  y  se hallan en estado activo. Estos servicios sirven para ver la instancia anterior y posterior a la instancia visualizada. Como es lógico, estos servicios sólo están activos cuando se ha accedido a una

instancia a través de la población. Si se sitúa el cursor del ratón sobre alguno de estos botones se visualizará el nombre de la instancia, anterior o posterior, en la etiqueta de ayuda flotante, que poseen algunos controles.

Una característica destacable de la herramienta es la posibilidad de definir distintos tipos de vistas sobre las clases: vistas por selección - representadas como el servicio - y vistas por proyección - representadas como el servicio -.

En la Figura 15 se muestra con un ejemplo cómo puede definirse una vista por proyección. Como se puede observar en la figura, una vez expresada la intención de generar una nueva vista por proyección, se despliega ante el usuario una nueva representación de la clase en la que aparecen todos los nombres de los atributos y servicios que forman parte de su signatura. Cada uno de estos elementos podrá seleccionarse o no a partir de esta pantalla de manera que, una vez definida la vista, se ocultarán al usuario:

- la parte del estado correspondiente a los atributos que no hayan sido seleccionados
- los servicios de la clase sobre la que se está definiendo la vista y que no aparecen en dicha definición como marcados (seleccionados).

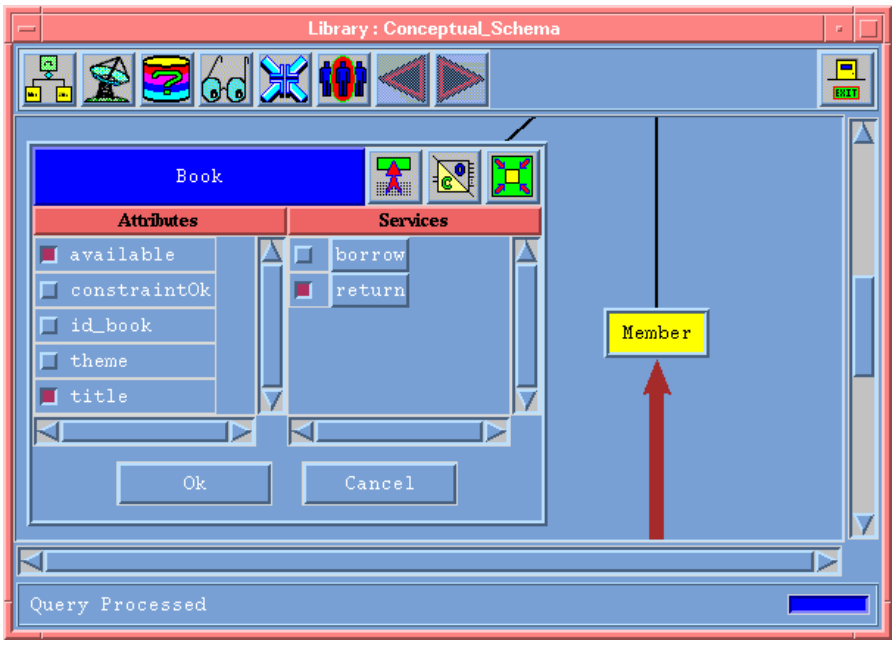


Figura 15: Definición de una vista por proyección sobre la clase *Book*

Tras la definición de la vista, el usuario dejará de percibir la clase libro tal y como la percibía hasta el momento y pasará a trabajar con la nueva vista. Las instancias de la clase libro son ahora objetos virtuales obtenidos a partir de las instancias reales de la clase libro sobre las que se aplica la proyección.

En la Figura 16 se ilustra la definición de una vista por selección sobre la clase libro. En la definición de este tipo de vista se hace uso de un OQL definido de acuerdo con el estándar ODMG-93 (Object Database Management Group, 1993) [Atw94]. Cada una de estas vistas puede ser nombrada, de forma que podrá ser recuperada en cualquier instante una vez definida. En el ejemplo podemos ver que la vista definida es la de todos aquellos *libros*

correspondientes al *tema divulgación* que no se encuentren prestados en el momento de la aplicación de la vista

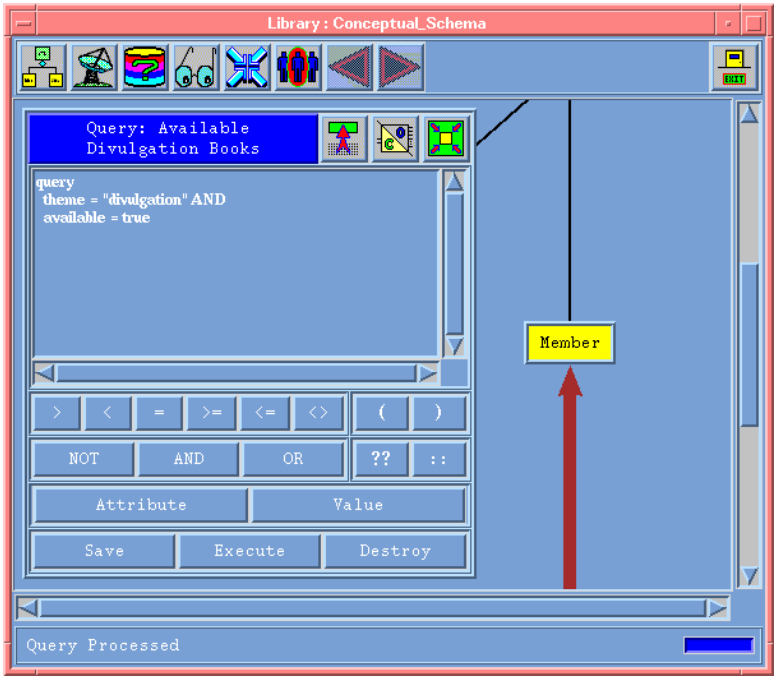


Figura 16: Definición de una vista por selección sobre la clase *Book*

El resultado de aplicar esta vista sobre la clase *libro* proporciona la población de una clase virtual obtenida a partir de la población original de la clase libro aplicando una condición de selección (expresada mediante un OQL).

El servicio representado por  muestra el Diagrama de Estados de Clase (DEC) de la clase sobre la cual se solicita.

5 Conclusiones y Trabajos Futuros

Se ha presentado un posible ambiente gráfico que posibilita la definición y explotación de sistemas de información dentro del marco formal y uniforme que proporciona la metaclase OASIS.

El ambiente gráfico tiene como ventaja el uso de un modelo único para la definición y explotación de los sistemas de información. La forma de trabajar para definir y explotar el esquema conceptual es exactamente la misma y está basada en una metáfora totalmente orientada a objetos.

El ambiente gráfico contempla la posibilidad de que el analista se introduzca en la sociedad como una instancia de cualquier clase que forme parte de cualquier esquema conceptual convirtiéndose en una potente herramienta de depuración de especificaciones al poder observar si los objetos se comportan tal y como se especificaron.

Es interesante observar la homogeneidad del ambiente presentado. El modo de trabajo es siempre el mismo, ya se trate de trabajar (crear, evolucionar o animar) con las clases y sus instancias ya se trate de trabajar (crear, evolucionar o animar) con las especificaciones.

Posibles líneas futuras de actuación:

- Proporcionar una apariencia más comercial del estilo de las últimas aplicaciones Windows.
- Acercar el DCC al propuesto por UML.
- Integrar otros modelos gráficos más conocidos como son los modelos de interacción, modelos funcionales, ...

Bibliografía

[Atw94] Atwodd, T.; Duhl, J.; Ferran, G.; Loomis, M.; Wade, D.; *The Object Database Standar: ODMG-93*, Ed. R.G.G. Cattell, 1994.

[Car96] Carsí, J.A.; Ramos, I.; Penadés, M.C.; Pelechano, V.; *Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic*, I Jornadas de Trabajo en Ingeniería del Software (JIS'96), Sevilla, pp.85-96, Noviembre 1996.

[Car97] Carsí, J.A.; Canós, J.H.; Ramos, I.; *Un intérprete de Transaction Frame Logic*, II Jornadas de Trabajo en Ingeniería del Software (JIS'97), San Sebastian, pp.36-50, Septiembre 1997.

[Car98a] Carsí J.A.; *Metaclase y Evolución del Esquema en Meta-OASIS*, II Jornadas de Trabajo en MENHIR (MENHIR'98), Valencia, pp.59-80, Febrero 1998.

[Car98b] Carsí J.A., Ramos I., Canós J.H.; *La Metaclase OASIS*, Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'98), Torres-BRASIL, pp.181-192, Abril 1998.

[Car98c] Carsí J.A., Letelier P., Sanchez P.; *A DOOD System for Treating the Schema Evolution Problem*, EDBT'98 Demo Session, VI Intl. Conference on Extending Database Technology, Valencia, pp.19-20, Marzo 1998.

[Pas96] Pastor, O. and Ramos, I., *OASIS version 2 (2.2): A Class-Definition Language to Model Information Systems using an Object-Oriented Approach*, SPUPV-95.788, Universitat Politècnica de València, 1996.

[Pas97] Pastor, O., Insfrán, E., Pelechano, V., Romero, J., Merseguer, J., *OO-Method: An OO software Production Environment Combining Conventional and Formal Methods*, Conference on Advanced Information Systems Engineering (CAiSE'97), Barcelona, España, Junio 1997.

[Pop98] Popkin, http://www.popkin.com.

[Rat98] Rational Software, http://www.rational.com.

[Wel97] Welch, B.B., *Practical Programming in Tcl & Tk*, Prentice Hall PTR, 1997.

Un ambiente para especificación incremental y validación de modelos conceptuales*

Patricio Letelier **Pedro Sánchez** **Isidro Ramos**
Departamento Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46071 Valencia
email {letelier | ppalma | iramos}@dsic.upv.es
http://www.dsic.upv.es/users/oom

Resumen

Las dificultades encontradas en la captura, especificación, verificación y validación de requisitos han impulsado variadas propuestas que mejoran dicho proceso. El modelo conceptual establece los requisitos funcionales del software y es uno de los resultados principales de dichas actividades, constituyéndose en una pieza clave para las posteriores tareas en el desarrollo de software. Los métodos formales dotan de precisión y rigor al modelo conceptual, pero presentan dificultades en cuanto a su aplicación y comprensión por el usuario. Por otro lado, las técnicas de prototipación han ganado aceptación por su fácil aplicación y proximidad al usuario, pero en general carecen de exhaustividad y precisión. Actualmente, la animación de modelos conceptuales formales emerge como un enfoque capaz de aprovechar las ventajas de los métodos formales y de la prototipación. OASIS es un enfoque formal para el modelado conceptual OO. El comportamiento de un objeto en OASIS está determinado por un conjunto de fórmulas expresadas mediante una variante de lógica dinámica. Este trabajo presenta un ambiente integrado para el modelado conceptual basado en OASIS, orientado especialmente a la especificación incremental y validación mediante animación.

1 Introducción

El modelo conceptual, representando los requisitos funcionales de un sistema de información, es la pieza clave para establecer el vínculo entre el espacio del problema y el espacio de la solución. Las deficiencias del modelo conceptual tienen un impacto considerable en las posteriores actividades del proceso de desarrollo de software.

La construcción del modelo conceptual es un proceso de descubrimiento, no sólo para el analista, sino también para el usuario. La estrategia que más se ajusta a esta situación es construir el modelo conceptual de forma iterativa e incremental, mediante interacción entre el usuario y el analista. Durante el modelado conceptual se llevan a cabo esencialmente cuatro tareas: captura de requisitos, modelado o especificación, verificación de criterios de calidad y consistencia, y finalmente validación. Una vez alcanzada la conformidad del analista y del usuario, el modelo conceptual es utilizado como entrada para las fases posteriores del desarrollo. El grado de exhaustividad en las validaciones con el usuario es una variable determinante para conseguir que el producto final se ajuste a lo esperado. La necesidad de validación del modelo conceptual está presente cada vez que se efectúa alguna modificación en él, sea ésta durante su proceso incremental de construcción, durante el diseño e implementación o durante el mantenimiento.

Los métodos formales [27] para el desarrollo de software (y en particular para el modelado conceptual) prometen una mejora en la calidad del resultado pues su aplicación permite expresar y determinar con rigor las propiedades del software. Sin embargo, la introducción de métodos formales no ha sido inmediata y sólo en la actualidad comienzan tímidamente a ser utilizados en el entorno industrial de

*Este trabajo ha sido financiado por el proyecto MENHIR de la Comisión Interministerial de Ciencia y Tecnología, con referencia TIC97-0593-C05-01.

Letelier, P., Sánchez, P., Ramos, I. Un ambiente para especificación incremental y validación de modelos conceptuales. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99).1999.

desarrollo de software [2, 25]. Cuatro razones explican esta situación [9]: falsas expectativas o desconocimiento, carencia de estándares para métodos y su aplicación, falta de herramientas integradas en entornos CASE y finalmente una precaria preparación entre sus usuarios potenciales.

En particular, para validar los requisitos del sistema se ha extendido la utilización de técnicas de prototipación. Un prototipo permite al usuario y al analista determinar las propiedades del sistema sobre una representación más cómoda para el usuario. Generalmente los prototipos son bosquejos de la interfaz del producto, concentrándose principalmente en aspectos de entrada y salida del sistema. Indudablemente la prototipación es una ayuda para la validación del modelo conceptual, pero no es suficiente. Un tipo particular de prototipación está orientada a simulación o animación del sistema. Este tipo de prototipación resulta atractiva para cubrir de forma exploratoria la validación del modelo conceptual, analizando su comportamiento ante situaciones de funcionamiento más complejas. Sin embargo, en esta estrategia se presentan dos importantes obstáculos: conseguir una representación ejecutable del modelo conceptual e interactuar con el usuario usando dicha representación.

De acuerdo a lo anteriormente expuesto y centrándose en las dificultades presentes en la construcción de modelos conceptuales, la utilización de enfoques formales aporta mejoras principalmente en aspectos de rigor y precisión de la especificación, facilitando su verificación. Sin embargo, respecto de la captura y validación de requisitos, las técnicas de prototipación han ganado más aceptación. De esto se deduce que los métodos formales y las técnicas de prototipación son complementarios, por lo cual resulta interesante una combinación de ambos. En los últimos años se ha registrado gran interés en torno a la validación de especificaciones formales mediante su animación [24]. En este sentido, este trabajo propone una enfoque para la animación de especificaciones formales mediante la ejecución de la especificación y orientada a la exploración interactiva del comportamiento del modelo conceptual y su validación respecto de escenarios [21].

Se utilizará OASIS 3.0 (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [12] como enfoque formal para la construcción de modelos conceptuales. Otras propuestas con una motivación similar a la de OASIS son: TROLL [10], LCM [5], Albert [4] y Oblog [23]. Algunas propuestas relacionadas con validación mediante animación en dichos trabajos son: la presentada en [8] que utiliza TROLL, y en [7] basada en Albert. Respecto de la animación establecida, las diferencias entre otras propuestas y este trabajo radican principalmente en las características formales del enfoque utilizado y la expresividad ofrecida. Además, considerando los resultados alcanzados, el estado del arte es similar y se limita a versiones preliminares de entornos para animación.

El asegurar la fidelidad de la animación respecto del modelo conceptual presenta los mismos problemas que se tienen en la fase de pruebas para verificar la corrección de la implementación respecto del modelo conceptual. Sin embargo, en este trabajo dicho problema es atenuado por los siguientes factores: a) el modelo conceptual está basado en un enfoque formal y se ha establecido un modelo abstracto de ejecución [16] que está inspirado en la semántica de dicho enfoque formal y b) la animación se implementa ([13, 15, 22]) en entornos de programación concurrentes que ofrecen la posibilidad de justificar formalmente las equivalencias entre constructores de la especificación OASIS del sistema y el programa obtenido para animación.

En este trabajo se presenta un ambiente para especificación incremental y validación de requisitos basado en OASIS. Se describen las características generales del entorno que, como es de esperar, se enmarcan dentro de las funcionalidades normalmente provistas por los entornos CASE actuales, sin embargo, en este trabajo se pone énfasis en aspectos de validación lo cual constituye la excepción y el aporte más significativo.

En el apartado 2 se exponen brevemente los aspectos básicos de OASIS. El apartado 3 presenta una propuesta para la construcción de un ambiente para especificación incremental y validación de requisitos. Los conceptos básicos del modelo de ejecución utilizado para animar especificaciones OASIS son introducidos en el apartado 4. El entorno gráfico para animación de especificaciones OASIS es descrito en el apartado 5. Finalmente, se presentan las conclusiones en el apartado 6.

2 OASIS

OASIS [12] es un enfoque formal para la especificación de modelos conceptuales usando el paradigma orientado a objeto. En OASIS un sistema de información es visto como una sociedad de objetos autónomos y concurrentes que se comunican entre sí a través de acciones.

OASIS está formalizado a través de una variante de Lógica Dinámica [6] que permite representar

los operadores de obligación, prohibición y permiso usados en Lógica Deóntica [1]. Una especificación OASIS es una presentación de una teoría en el sistema formal usado, expresada como un conjunto estructurado de definiciones de clase. Las clases pueden ser simples o complejas. Una clase compleja es aquella que en su definición utiliza la definición de otras clases (simples o complejas). Las clases complejas se definen estableciendo relaciones entre clases. Estas son agregación y herencia.

Una clase se compone de un nombre de clase, uno o más mecanismos de identificación para instancias de la clase (objetos) y un tipo o plantilla que comparten todas las instancias. La plantilla recoge las propiedades de estructura y de comportamiento compartidas por todos los objetos potenciales de la clase considerada. Para una clase simple todas las propiedades quedan establecidas en su plantilla. Para una clase compleja, además de las propiedades establecidas por su plantilla, existirán propiedades adicionales determinadas por los operadores de clase que la definen. A continuación se presentan los conceptos básicos de OASIS.

Definición 1 ***Plantilla o tipo.*** *Una plantilla de clase está representada por una tupla* $\langle Atributos, Eventos, Fórmulas, Procesos\rangle$.

- *Atributos* es el alfabeto de atributos. $\forall att \in Atributos$, se tiene la función att : *sort de nombrado* $\rightarrow$ *sort de evaluación*
- *Eventos* es el alfabeto genérico de eventos. $\forall e \in Eventos$ podemos obtener $\underline{e} = \theta e$ siendo θ una substitución básica de los posibles parámetros del evento.
- *Fórmulas* es un conjunto de fórmulas en diversas lógicas según sea la sección del lenguaje donde se utilizan.
- *Procesos* es el conjunto de especificaciones de proceso. Cada proceso se especificará usando un cálculo de procesos basado en un subconjunto del lenguaje propuesto en CCS [18]. Las especificaciones de proceso serán divididas en operaciones y protocolos.

Definición 2 ***Servicio.*** *Un servicio es un evento o una operación. En el primer caso se trata de un servicio atómico e instantáneo. Una operación es un servicio no atómico y que, en general, tiene duración.*

Definición 3 ***Acción.*** *Una acción es una tupla* $\langle Cliente, Servidor, Servicio\rangle$, *que representa tanto la acción asociada a requerir el servicio en el objeto cliente como la acción asociada a proveer el servicio en el objeto servidor.*

Para cada clase, asumiremos la existencia implícita de un conjunto A de acciones obtenido a partir de los servicios que los objetos de la clase pueden requerir (cuando son clientes) o proveer (cuando son servidores). Al igual que para los eventos, $\forall a \in A$ podemos obtener $\underline{a} = \theta a$ siendo θ una substitución básica de los posibles cliente, servidor y servicio.

Definición 4 ***Atributo.*** *Un atributo es un par* $\langle$*nombre, sort de evaluación*$\rangle$, *siendo nombre el identificador del atributo y sort de evaluación el nombre del conjunto soporte (carrier) con los valores que puede tomar dicho atributo.*

Definición 5 ***Estado del objeto.*** *Llamaremos estado del objeto al conjunto de sus atributos evaluados. Se expresa normalmente mediante Fórmulas bien formadas (Fbfs) de una lógica de primer orden.*

El estado permite caracterizar a un objeto en un instante determinado. Sea Σ el conjunto de estados alcanzables por un objeto y $\sigma \in \Sigma$ un estado en particular. El estado de un objeto cambia por la ocurrencia de acciones. Los cambios de estado son modelados por la siguiente relación:

$$efecto : \underline{A} \rightarrow (\Sigma \rightarrow \Sigma)$$

Donde $\underline{A}$ es el conjunto de acciones instanciadas ($\underline{a} \in \underline{A}$) que le pueden acontecer al objeto y $efecto$ es la función que asocia a cada acción instanciada una función de cambio de estado, siendo $(\Sigma \rightarrow \Sigma)$ el conjunto de todas las funciones de cambio de estado.

Definición 6 ***Conjunto consistente de acciones.*** *Un conjunto de acciones* $M \subseteq \underline{A}$ *es consistente si a partir de un estado* σ *dado, el estado* σ' *alcanzado por la ocurrencia de las acciones en* M *es siempre el mismo, independiente del orden en el cual se ejecuten*[1] *las acciones.*

Definición 7 ***Paso.*** *Llamaremos paso a un conjunto consistente de acciones que ocurren en un mismo instante de la vida del objeto.*

[1]Se utilizará "ejecución de acción" como sinónimo de "ocurrencia de acción" para hacer más intuitiva la presentación. Sin embargo, desde el punto de vista declarativo y formal, "ocurrencia" es el término más adecuado.

Sea $\mu = \{a_1, ..., a_n\} \subseteq \underline{A}$ un paso. La relación de accesibilidad entre estados está determinada por μ, $R_\mu \subseteq \Sigma \times \Sigma$ y se define como (siendo "$\circ$" el operador de composición funcional):

$$R_\mu(\sigma) \stackrel{def}{\equiv} efecto(a_1)(\sigma) \circ ... \circ efecto(a_n)(\sigma) = \sigma'$$

De acuerdo con la definición de conjunto consistente, el orden en el cual se aplica la función $efecto$ a las acciones no altera el estado siguiente alcanzado por el objeto.

Definición 8 ***Vida o traza de un objeto.*** *Llamaremos vida o traza de un objeto a un prefijo finito de pasos que le acontecen al objeto. De esta forma, la traza es una caracterización alternativa del estado del objeto en un instante determinado.*

2.1 OASIS expresado en Lógica Dinámica

Globalmente y considerando su expresividad, OASIS está basado en Lógica Deóntica. La Lógica Deóntica es la lógica de obligaciones, prohibiciones y permisos. Siendo $a \in A$, en OASIS se utilizan los siguientes operadores de la Lógica Deóntica:

$O(a)$	obligación de ocurrencia de a.
$F(a)$	prohibición de ocurrencia de a.
$P(a)$	permiso o habilitación de ocurrencia de a.

En [17] la Lógica Deóntica es descrita como una variante de Lógica Dinámica. La definición de los operadores deónticos en Lógica Dinámica es la siguiente:

$\psi \rightarrow [a]false$	"la ocurrencia de a está prohibida en los estados que satisfacen ψ".
$\psi \rightarrow [\neg a]false$	"la ocurrencia de a es obligatoria en los estados que satisfacen ψ".
$\psi \rightarrow [a]\phi$	"en los estados que satisfacen ψ, inmediatamente después de la ocurrencia de la acción a, ϕ debe satisfacerse".

Donde ψ es una Fbf que caracteriza el estado del objeto inmediatamente antes de la ocurrencia de la acción a y $\neg a$ representa la no-ocurrencia de la acción a (es decir, que no ocurre nada o que ocurre otra acción distinta de a). Además, $false$ es un átomo tal que no hay un estado que pueda hacerlo verdadero. Esta situación podría representar un estado de violación del sistema. El significado intuitivo de éstas fórmulas es: una acción está prohibida si su ocurrencia conduce a un estado de violación. Una acción es obligatoria si su no-ocurrencia conduce a un estado de violación. La Lógica Dinámica es un formalismo natural para estudiar de forma simple y directa ciertas aserciones sobre programas. En Lógica Dinámica, la fórmula $[a]\phi$ se interpreta como "después de la ocurrencia de la acción a, ϕ debe satisfacerse", siendo a una acción y ϕ una Fbf de la lógica de primer orden utilizada para caracterizar el estado. Estas fórmulas constituyen un sublenguaje del lenguaje de Lógica Dinámica propuesto y formalizado en [26].

En OASIS, el modelado del comportamiento puede realizarse desde dos perspectivas complementarias en cuanto a expresividad. En cada instante, para un objeto, las acciones permitidas y las acciones obligatorias pueden ser establecidas de la siguiente forma:

- Considerando el estado del objeto en dicho instante. Las precondiciones son fórmulas que permiten la ocurrencia de acciones sólo en ciertos estados del objeto. De forma análoga, los disparos son fórmulas que obligan la ocurrencia de acciones en determinados estados del objeto.

- Considerando las acciones ya acontecidas. Una especificación de proceso determina un patrón de comportamiento para el objeto desde el punto de vista de secuencias de pasos acontecidos, es decir, determina trazas posibles. En OASIS se utilizan dos tipos de especificaciones de proceso: *protocols* y *operations*. Un protocolo establece secuencias permitidas de acciones. Una operación establece secuencias obligatorias de acciones. En [14] se explica la interpretación de una especificación de proceso como un conjunto de fórmulas en la variante de lógica dinámica utilizada.

La semántica de OASIS es dada en términos de una estructura de Kripke (W, τ, ρ). W es el conjunto de todos los mundos[2] posibles que un objeto puede alcanzar.

[2] De acuerdo con lo dicho, los estados son aserciones (fórmulas), los mundos son estructuras sobre las que dichas fórmulas son interpretadas.

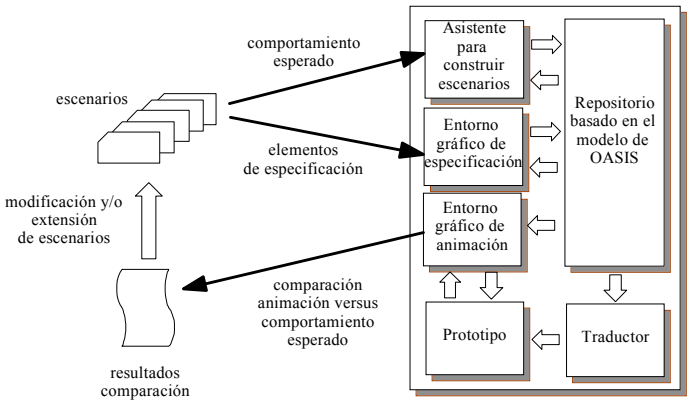


Figura 1: Un ambiente para construcción incremental y validación de modelos conceptuales

Sea F el conjunto de Fbfs evaluadas sobre el estado (mundo asociado) en el cual se encuentra el objeto, A el conjunto de acciones de la signatura del objeto y $2^{\underline{A}}$, el conjunto de pasos posibles instaciados. Las funciones τ y ρ se definen como:

$$\tau : F \rightarrow 2^W$$
$$\rho : 2^{\underline{A}} \rightarrow (W \rightarrow W)$$

La función τ asigna a una fórmula en la lógica de estado utilizada (Lógica de Predicados de Primer Orden) el conjunto de mundos en los cuales se satisface. La función ρ asigna a cada paso una relación binaria entre mundos, la cual es la semántica declarativa del lenguaje. Siendo $\mu \in 2^{\underline{A}}$ y $w, w' \in W$, el significado buscado es: $(w, w') \in \rho(\mu)$ si y sólo si la ocurrencia de μ conduce al objeto desde el mundo w al mundo w'.

3 Un ambiente para especificación y validación

El modelo conceptual debería ser construido incrementalmente mediante interacción entre el usuario y el analista. Como se trata de un proceso de descubrimiento, en cada paso sería recomendable que el modelo conceptual fuera validado respecto de los requisitos del usuario. Normalmente esta tarea de comparación se realiza manualmente y por simple inspección. Cualquier apoyo automatizado para el proceso de validación debe disponer de una representación para los requisitos que pueda contrastarse respecto de la funcionalidad del modelo.

La Figura 1 ilustra el ambiente para la especificación incremental y validación de modelos conceptuales que se está construyendo. En términos generales, se trata de un trabajo que puede ser extendido hacia el desarrollo de una herramienta CASE. Sin embargo, se ha prescindido de características no estrictamente relacionadas con la construcción incremental y validación del modelo conceptual. La programación de las interfaces gráficas se está realizando con Tcl/Tk [19]. Los componentes del entorno (en el rectángulo de la derecha en la Figura 1) son:

- Un asistente para construir escenarios en un formato adecuado para su comparación. La programación de este módulo aún no se ha comenzado. Las técnicas usadas para especificar escenarios varían considerablemente (ver[21]). Como el interés se centra en realizar una comparación de resultados obtenidos por animación respecto de resultados esperados incluidos en los escenarios, tanto la especificación de escenarios como la animación del modelo conceptual estarán caracterizadas por trazas de acciones y estados alcanzados de la vida de los objetos estudiados.

- Un entorno gráfico de especificación, que permita editar el modelo conceptual. Este módulo permite la edición gráfica del modelo, definiendo los elementos de la especificación que se almacenan en el repositorio. Se ha construido una versión preliminar que permite especificar gráficamente la configuración de clases y en forma textual los detalles de la plantilla de una clase. Alternativamente a la introducción textual, para las especificaciones de proceso de una clase (operaciones y protocolos) se está integrando un editor gráfico de diagramas

de transición de estados. La notación gráfica utilizada está adaptándose para hacerla compatible con UML [20].

- El repositorio, que almacena el modelo conceptual y los escenarios. Corresponde a un metamodelo estático para especificaciones OASIS visto como un modelo relacional (detallado en [11]).
- Un traductor, que genera automáticamente un prototipo a partir del modelo conceptual almacenado. Se ha construido un traductor que a partir del repositorio genera un programa lógico concurrente en KL1[3] cuyo ejecutable constituye el prototipo para la animación. El traductor implementa las pautas presentadas en [15].
- El entorno de animación, que interactúa con el prototipo validando cada uno de los escenarios respecto del comportamiento del prototipo. En el contexto de OASIS el comportamiento del sistema se corresponde con el comportamiento de la sociedad de objetos incluidos en él y puede ser observado a través de las acciones acontecidas y los estados alcanzados por cada objeto. Así, la animación de una especificación OASIS permite examinar el comportamiento de un objeto o grupo de objetos, estudiando las acciones acontecidas y los correspondientes estados alcanzados. Esta facilidad exploratoria constituye una mejora para la validación del modelo conceptual. Sin embargo, el aporte más significativo se conseguirá cuando se puedan contrastar con asistencia automática las trazas esperadas respecto de las obtenidas por animación. Se ha construido una versión preliminar para este entorno [15], sin incluir la funcionalidad de comparación de trazas. Posteriormente se explicará con más detalle este módulo.

El proceso de construcción del modelo conceptual, desde la perspectiva del ambiente propuesto, está compuesto por las siguientes actividades:

1. Capturar requisitos del usuario mediante técnicas de escenarios.
2. Especificar los escenarios en una forma representativa para su comparación (objetos, acciones y estados alcanzados).
3. Especificar los elementos de la especificación asociados a cada escenario y que determinan partes del modelo conceptual.
4. Verificación automática de la consistencia del modelo conceptual y de los escenarios introducidos.
5. Obtención automática de un prototipo para animación.
6. Validación del modelo conceptual. Mediante animación de la especificación se contrasta el funcionamiento del sistema respecto de cada escenario introducido. Además, realizando una animación exploratoria pueden buscarse situaciones de funcionamiento que evidencien comportamientos no deseados del modelo conceptual o que determinen ampliaciones.
7. Modificación y/o extensión del conjunto de escenarios.

Estas actividades se repiten en la secuencia establecida hasta conseguir un resultado satisfactorio en lo que respecta a las actividades 4 y 6. Además, las actividades 2 y 3 se realizan en conjunto por ser dependientes. Cualquier modificación en los requisitos funcionales del sistema que determine cambios en los escenarios podría activar nuevamente dicho ciclo de actividades.

4 Un modelo de ejecución para OASIS

Siendo la utilidad fundamental de este trabajo su uso en la validación de los requisitos expresados en una especificación OASIS, es indispensable mantener la implementación lo más cercana posible a la semántica de OASIS. De esta forma existirá un camino de formalización y verificación de la implementación obtenida respecto de la semántica de una especificación OASIS. En este sentido, el modelo de ejecución usado es un animador abstracto de fórmulas de obligación, permiso y cambio de estado, asociadas a un objeto. La información utilizada por el modelo es la signatura de atributos y acciones del objeto junto al conjunto de fórmulas en lógica dinámica que describen su comportamiento. A continuación se describen brevemente los conceptos incluidos en el modelo de ejecución propuesto en [16].

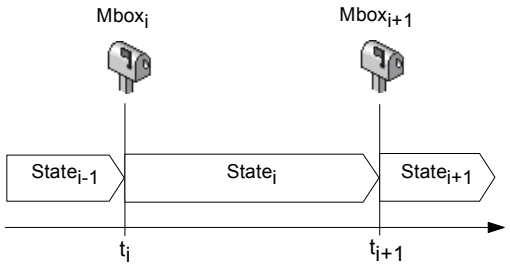


Figura 2: Ciclo de vida de un objeto

4.1 Aspectos básicos

La secuencia de pasos que le acontecen a un objeto está ordenada respecto del tiempo. Podemos suponer que existe un objeto "reloj" que envía acciones especiales — llamadas *ticks* — hacia cada objeto del sistema[3]. Los *ticks* recibidos por un objeto son correlativos con los números naturales, t_1, t_2, etc. Sean i y j números naturales tales que se cumple $i < j \Longleftrightarrow t_i < t_j$.

Definición 9 ***Buzón.*** *Es el conjunto de acciones que pueden formar parte del paso que un objeto ejecuta en un determinado tick. El buzón en el instante t_i se denota por $Mbox_i$. Consideraremos cada buzón como un buffer ilimitado, en el que las acciones se ordenan por orden de llegada.*

Definición 10 ***Estado i-ésimo del objeto.*** *Denotado por $State_i$, representa el estado del objeto en el intervalo $[t_i,t_{i+1})$. Es decir, desde t_i inclusive y hasta justo antes de t_{i+1}, el estado se considera constante.*

La Figura 2 ilustra la vida de un objeto según las definiciones dadas. El conjunto de acciones en cada paso es obtenido desde el buzón y la ejecución del paso en un instante produce un cambio de estado que no será observable hasta el siguiente *tick*. Así, en la estructura de tiempo utilizada aunque la ocurrencia de acciones es instantánea, los estados tienen una cierta duración.

4.2 Clasificación de acciones en el buzón

El procesamiento de las acciones del buzón implica clasificar las acciones contenidas en él. A continuación se presentan las categorías de acciones identificables en el buzón, caracterizadas como subconjuntos de $Mbox_i$ (es decir, en el instante t_i).

Definición 11 ***Acciones obligadas de requerir.*** *Acciones asociadas a obligaciones por requerir un servicio (en las cuales el cliente es el propio objeto) y que* ***deben*** *acontecer. El conjunto de acciones obligadas de requerir se denota por $OReq_i$.*

Las acciones de requerir están determinadas por fórmulas de obligación, es decir, de la forma: $\phi[\neg a]false$, en las cuales a es la tupla $< self, Servidor, Servicio >$, es decir, el cliente de la acción es el propio objeto, indicado usando la palabra $self$.

Definición 12 ***Acciones obligadas de proveer.*** *Acciones correspondientes a servicios que el objeto* ***debe*** *proveer. Estas acciones están asociadas a obligaciones, en las cuales el cliente no es el propio objeto. El conjunto de acciones obligadas de proveer en se denota por $OProv_i$.*

Las acciones obligadas de proveer están determinadas por fórmulas de obligación, es decir, de la forma: $\phi[\neg a]false$, en las cuales a es la tupla $< Cliente, self, Servicio >$, es decir, el servidor de la acción es el propio objeto.

Definición 13 ***Acciones obligadas de ejecutar.*** *Este conjunto se denotado por $OExec_i$ y se define como: $OExec_i = OReq_i \cup OProv_i$.*

Supondremos que el garantizar la no-existencia de conflicto[4] entre acciones obligadas es parte de la verificación estática de la especificación. Es decir, en tiempo de ejecución no puede presentarse conflicto entre obligaciones. Del mismo modo, supondremos que no puede darse una situación en la cual una acción obligada no está permitida.

[3] Para propósitos de animación, en este punto no tiene mayor importancia el que el reloj sea sólo uno para todo el sistema, uno para cada objeto o cualquier otra alternativa.

[4] En el modelo de ejecución se establece que dos acciones están en conflicto si afectan a conjuntos no disjuntos de atributos.

Definición 14 ***Acciones no-obligadas de ejecutar.*** *Acciones correspondientes a servicios solicitados por otros objetos (o por el propio objeto) y que el objeto* ***puede*** *proveer, dependiendo de los permisos establecidos sobre dichas acciones y verificados sobre $State_i$. El conjunto de acciones no-obligadas se denota por $\overline{O}Exec_i$.*

Definición 15 ***Acciones rechazadas.*** *Es un subconjunto de $\overline{O}Exec_i$ constituido por acciones no-obligadas de ejecutar cuya ocurrencia no está permitida. El conjunto de acciones rechazadas se denota por $Reject_i$.*

Las acciones rechazadas están determinadas por fórmulas de prohibición, es decir, de la forma: $\phi[a]false$.

Definición 16 ***Acciones candidatas.*** *Es un subconjunto de $\overline{O}Exec_i$ constituido por acciones no-obligadas de ejecutar cuya ocurrencia está permitida. El conjunto de acciones candidatas se denota por $Cand_i$.*

Además, se cumple que $\overline{O}Exec_i = Reject_i \cup Cand_i$.

Definición 17 ***Acciones ejecutadas.*** *Es el conjunto de acciones correspondiente al paso que se ejecuta en t_i. El paso se denota por $Exec_i$. El paso estará constituido por $OExec_i$ más un subconjunto de $Cand_i$, es decir, acciones no-obligadas de ejecutar que están permitidas.*

La selección del subconjunto de acciones de $Cand_i$ que serán incluidas en $Exec_i$ puede responder a distintos criterios y la única restricción es que no se produzcan conflictos entre las acciones del paso.

Definición 18 ***Acciones en conflicto.*** *Es un subconjunto de $Cand_i$ constituido por acciones no-obligadas de ejecutar cuya ocurrencia está permitida, pero que están en conflicto con alguna de las acciones obligadas de ejecutar o con alguna de las acciones no-obligadas seleccionadas. El conjunto de acciones en conflicto se denota por $Conf_i$.*

Se utiliza un criterio sencillo para la selección de acciones desde $Cand_i$: ante dos acciones que no pueden ser seleccionadas por estar en conflicto se seleccionará aquella llegada antes al buzón. Este criterio evita postergación indefinida de acciones en conflicto. Las acciones en conflicto $Conf_i$ serán "copiadas" al buzón correspondiente al siguiente instante ($Mbox_{i+1}$).

El comportamiento del objeto está caracterizado por un algoritmo que repetidamente procesa el buzón para determinar los subconjuntos definidos anteriormente y produce el cambio de estado del objeto.

Las equivalencias establecidas entre conceptos de OASIS y construcciones en programación lógica concurrente [13] junto al modelo de ejecución han permitido definir las pautas generales para traducir una especificación OASIS hacia el lenguaje lógico concurrente KL1. Posteriormente, se ha construido un traductor que recupera la especificación almacenada en el repositorio, genera un programa KL1 y lo compila, obteniéndose finalmente un prototipo ejecutable de la especificación. El módulo asociado al entorno gráfico de animación permite al analista interactuar de una forma cómoda con el prototipo. La versión preliminar de este módulo incluye un análisis básico de trazas pero aún no incorpora la comparación con trazas esperadas obtenidas de los escenarios definidos. El siguiente apartado muestra la funcionalidad hasta ahora implementa en dicho módulo a través de un ejemplo.

5 Entorno de animación

El siguiente ejemplo en OASIS modela un sistema bancario de "juguete".

```
conceptual schema banking_system
class account
identification
    number:(number);
constant attributes
    number:nat;
    name:string;
variable attributes
    balance:nat(0);
    times:nat(0);
    pin:nat(0);
    rank:nat(0);
derived attributes
    good_balance:bool;
derivations
    good_balance:={balance>=100};
events
    open new;
    close destroy;
```

```
    deposit(Amount:nat);
    withdraw(Pin:nat,Amount:nat);
    pay_commission;
    change_pin(Pin:nat,NewPin:nat);
    change_rank(Rank:nat);
valuations
    [deposit(Amount)]
        balance:=balance+Amount,
        times:=times+1;
    [withdraw(Pin,Amount)]
        balance:=balance-Amount,
        times:=times+1;
    [self:pay_commission[5]]
        balance:=balance-1;
    [::pay_commission]
        times:=0;
    [change_pin(Pin,NewPin)]
        pin:=NewPin;
    [change_rank(Rank)]
        rank:=Rank;
preconditions
    withdraw(Pin,Amount) if
        (pin=Pin and balance>=Amount) or
            (pin=Pin and balance<Amount
        and rank=2);
    change_pin(Pin,NewPin) if (pin=Pin);
    close if (balance=0);
triggers
    ::pay_commission when
        (times>=5 and good_balance=false
        and rank=0);
end class

class customer
identification
    name:(name);
constant attributes
    name:string;
events
    add new;
    remove destroy;
end class

interface customer(someone)
            with account(someone)
    services(deposit,withdraw,change_pin);
end interface

interface account(someone) with self
    services(pay_commission);
end interface

end conceptual schema
```

La Figura 3 ilustra el inicio de la sesión de animación. La animación empieza desde el ambiente integrado, donde previamente se carga (o se crea) un esquema conceptual, se verifica su consistencia y se genera el prototipo. La ventana de la Figura 3 está dividida en dos áreas: a la izquierda el área de objetos permite visualizar los objetos que están activos y a la derecha se muestran las listas de acciones procesadas hasta el último *tick* significativo (aquel en el cual algún objeto alcanzó un nuevo estado). Se ha implementado un mecanismo versátil de filtros para la visualización de acciones y facilitar el análisis de las trazas de objetos. Por un lado, en el área de objetos se puede seleccionar un objeto, un grupo de objetos o todos los objetos del esquema conceptual (haciendo click en el fondo del área izquierda). Adicionalmente, con los botones del área derecha, se puede seleccionar el subconjunto de acciones que se desea visualizar. Por defecto se muestran todas ($Mbox_i$), pero se pueden filtrar las obligadas de requerir ($OReq_i$), las no-obligadas de ejecutar ($\overline{O}Exec_i$), las acciones en conflicto ($Conf_i$), las acciones ejecutadas ($Exec_i$) o las acciones rechazadas ($Reject_i$)[6]. Los botones de la parte inferior del área izquierda tienen la siguiente funcionalidad: el botón *play* comienza la animación; el botón *stop* detiene la animación para comenzar una nueva con *play*; el botón pausa suspende la animación y habilita los botones *rewind* y *forward* que permiten retroceder o avanzar examinando las trazas en *ticks* ya ejecutados (el área de la derecha se refresca a medida que se cambia el *tick*. Al avanzar o retroceder sólo se muestran los *ticks* significativos.

Al comienzo, se refleja la creación de un metaobjeto por cada clase del esquema. Además, también por defecto, se crea un objeto `user` identificado por `root`. A través de este objeto el analista puede crear las instancias necesarias de las clases correspondientes.

[5] En OASIS, para evitar detallar cada elemento de la tupla $< Cliente, Servidor, Servicio >$ que representa a una acción, se adoptan las siguientes simplicaciones: a) Si el servidor es el propio objeto, se escribe *Cliente:Servicio* y si el cliente no es relevante sólo se escribe *Servicio*. b) Si el cliente es el propio objeto, se escribe *Servidor::Servicio*. *c)* En el caso particular de comunicación *self*, la acción de requerir se escribe *::Servicio* y la acción de proveer *self:Servicio* (o simplemente *Servicio*).

[6] En esta versión del traductor y entorno de animación no ha sido incorporado el tratamiento de acciones obligadas de proveer ($OProv_i$).

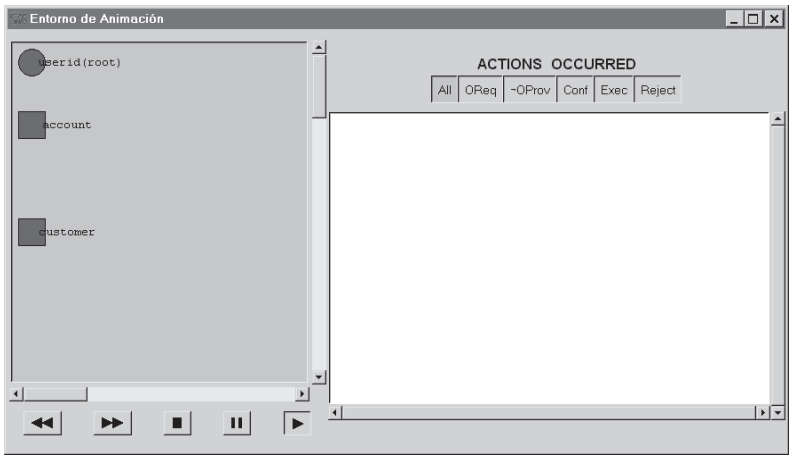


Figura 3: Inicio de la sesión de animación

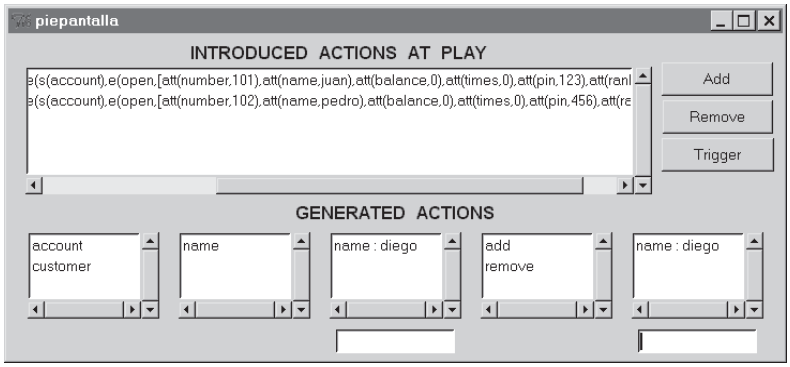


Figura 4: Construcción de acciones de creación de instancias

Al hacer un click sobre un objeto aparece la ventana de la Figura 4. Esta ventana muestra todas las clases y servicios que son accesibles para el objeto seleccionado. Además, se mostrará una ventana indicando el estado del objeto seleccionado. Por defecto, el objeto `user(root)` puede acceder a todos las clases del esquema y a todos sus servicios. Para otros objetos dicho acceso depende exclusivamente de las interfaces establecidas en el esquema conceptual. En la ventana de la Figura 4 se realiza la creación de dos instancias de cuenta (con número `101` y `102`) y una instancia de cliente identificado por el nombre "diego".

Después de lanzar las acciones de creación y hacer click en el fondo del área de objetos, la ventana resultante se muestra en la Figura 5. A continuación, se recreará un escenario en el cual un objeto cliente realiza cuatro depósitos y un reintegro en una cuenta. Los servicios solicitados son: `deposit(10)`, `deposit(20)`, `deposit(30)`, `deposit(40)` y `withdraw(50)`. Seleccionando el objeto `customer(diego)` se accederá a una ventana similar a la Figura 4 desde la cual el analista puede construir las acciones y requerir los servicios en representación del objeto `customer(diego)`. Las cinco acciones serán enviadas al objeto `account(101)`.

Si posteriormente se selecciona el objeto `account(101)` y se presiona el botón $Exec_i$, puede verse en la Figura 6 la lista de todas las acciones ejecutadas hasta el último *tick* significativo. La Figura 7 muestra el estado del objeto `account(101)` a partir de dicho *tick*.

Las limitaciones de presentación del presente trabajo impiden ilustrar más exhaustivamente una situación de exploración de trazas ejecutadas. Sólo es posible mencionar las principales funcionalidades del entorno tal como se ha intentado.

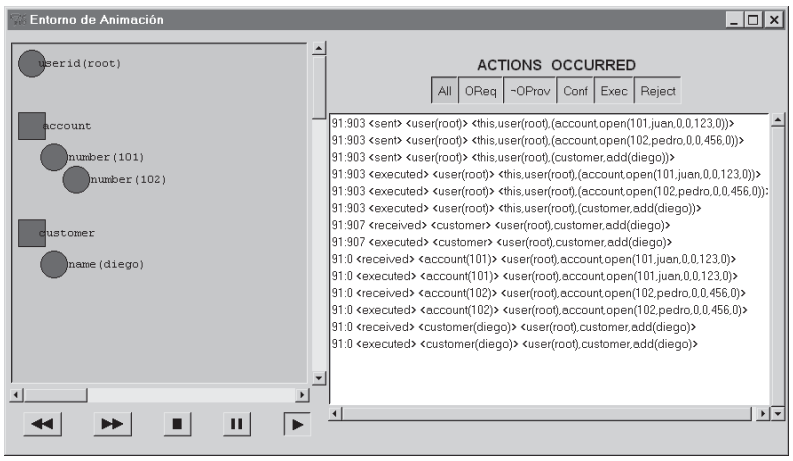


Figura 5: Sesión de animación después de crear tres instancias

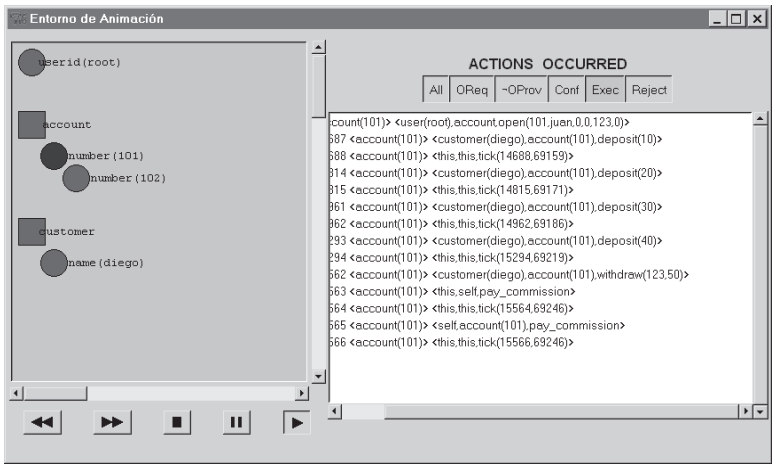


Figura 6: Acciones ejecutadas por el objeto `account(101)`

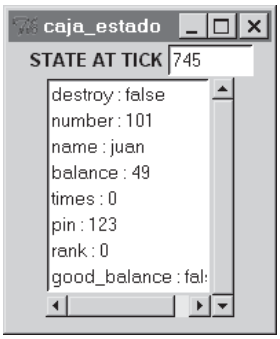


Figura 7: Estado actual del objeto `account(101)`

6 Conclusiones

En OASIS, el comportamiento de un objeto es formalizado con una variante de lógica dinámica y se expresa como un conjunto de fórmulas que representan obligaciones, permisos y cambios de estado. Así, el prototipo automáticamente obtenido es un animador para este tipos de fórmulas. El asegurar la fidelidad de la animación respecto del modelo conceptual OASIS presenta los mismos problemas que se tienen en la fase de pruebas para verificar la corrección de la implementación respecto del modelo conceptual. Sin embargo, en este trabajo dicho problema es atenuado por los siguientes factores: a) el modelo conceptual está basado en un enfoque formal, para el cual se ha establecido un modelo abstracto de ejecución inspirado en la semántica de OASIS y b) la animación que se ha propuesto está implementada en programación lógica concurrente, la cual ofrece la posibilidad de justificar formalmente las equivalencias entre constructores de la especificación OASIS del sistema y el programa obtenido para animación.

Se han expuesto las líneas generales de un ambiente integrado para la especificación de modelos conceptuales. Aunque este trabajo, se enmarca dentro del contexto de herramientas CASE, el énfasis se ha puesto en la validación temprana de requisitos mediante la animación de la especificación, usando un prototipo generado automáticamente. Esto ha hecho prescindir de funcionalidades que normalmente están presentes en un CASE pero que no tienen una relación estrecha con la especificación y validación del modelo conceptual. El ambiente propuesto está en construción y los módulos componentes se encuentran en distintos estados de avance. No se ha cubierto toda la expresividad de OASIS y las extensiones necesarias son aún considerables. Se ha bosquejado el proceso de modelado conceptual apoyado por animación. El enfoque utilizado, que combina un modelo formal con técnicas de escenarios y prototipación automática de especificaciones, provee mejoras significativas al proceso de modelado conceptual.

Referencias

[1] Åqvist L. Deontic logic. In D.M. Gabbay and F.Guenthner, editors, Handbook of Philosophical Logic II, pp.605-714. Reidel, 1984.

[2] Craigen D., Gerhart S., Ralston T. Formal methods reality check: Industrial usage. IEEE Transactions on Software Engineering, vol.21, n.2, pp. 90-98, February 1995.

[3] Chikayama, T., KLIC User's Manual. Institute for New Generation Computer Technology, Tokyo JAPAN, March 1995.

[4] Dubois,E., Du Bois P., Petit M. O-O Requirements analysis: An agent perspective. In Proc.of the 7th European Conference on Object Oriented Programming ECOOP 93, pp.458-481. July 1993.

[5] Feenstra R., Wieringa R.J. LCM 3.0: A language for describing conceptual models. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1993.

[6] Harel D. Dynamic Logic. In Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pags. 497-694. Reidel 1984.

[7] Heymans P. The Albert II Specification Animator. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm.

[8] Herzig R., Gogolla M. An animator for the object specification language TROLL light. Proc.Colloq.on Object-Orientation in Databases and Software Engineering, Montreal 1994.

[9] Hinchey M.G., Bowen J.P. To formalize or not to formalize?. En mesa redonda "An invitation to formal methods", IEEE Computer, pp.18-19, Abril 1996.

[10] Jungclaus R., Saake G., Hartmann T., Sernadas C.TROLL - A Language for Object-Oriented Specification of Information Systems. ACM Transactions on Information Systems, Volume 14, Number 2, pp.175-211, April 1995.

[11] Letelier P., Ramos I. Un metamodelo estático para especificaciones OASIS y su implementación en una base de datos relacional. Informe técnico DSIC-II/19/96, DSIC-Universidad Politécnica de Valencia, 1996.

[12] Letelier P., Ramos I., Sánchez P., Pastor O. OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto. Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV-98.4011, 1998.

[13] Letelier P., Sánchez P., Ramos I. Animation of system specifications using concurrent logic programming. Symposium on Logical Approaches to Agent Modeling and Design, ESSLLI'97, Aix-en-Provence, France, August 1997.

[14] Letelier P., Sánchez P., Ramos I. Especificaciones de proceso para objetos y su representación en lógica dinámica. Actas de las IV Jornadas de Ingeniería del Software JIS'98, Murcia, Noviembre 1998.

[15] Letelier P., Sánchez P., Ramos I. Prototyping a requirements specification through an automatically generated concurrent logic program. G. Gupta (Ed.): Practical Aspects of Declarative Languages, Springer-Verlag, LNCS 1551, pp. 31-45, 1998.

[16] Letelier P., Ramos I., Sánchez P., Pastor O. Un modelo de ejecución para especificaciones OASIS 3.0. Actas de las III Jornadas de Trabajo MENHIR, Murcia, Noviembre 1998.

[17] Meyer J.-J.Ch. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. In Notre Dame Journal of Formal Logic, vol.29, pp.109-136, 1988.

[18] Milner R. Communication and Concurrency. Prentice Hall Series in Computer Science, C.A.R. Hoare, Series Editor. 1989.

[19] Ousterhout J. Tcl and the Tk Toolkit. Addison-Wesley, 1994.

[20] Rational Software Corporation. UML notation guide. Versión 1.1, Septiembre 1997. http://www.rational.com/uml.

[21] Rolland C., Ben Achour C., Cauvet C., Ralyté J., Sutcliffe A., Maiden N.A.M., Jarke M., Haumer P., Pohl K., Dubois E., Heymans P. A Proposal for a Scenario Classification Framework, Technical Report CREWS 96-01, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm.

[22] Sánchez P., Letelier P., Ramos I. Constructs for Prototyping Information Systems using Object Petri Nets, Proc.of IEEE International Conference on System Man and Cybernetics, pp. 4260-4265, Orlando, USA, Octubre 1997.

[23] Sernadas C., Gouveia P., Sernadas A. Oblog: Object-oriented, logic-based conceptual modelling. Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, 1096 Lisboa, Portugal, 1992.

[24] Siddiqi J., Morrey I.C., Roast C.R., Ozcan M.B. Towards quality requirements via animated formal specifications. Annals of Software Engineering, n.3, 1997.

[25] 2RARE. 2 Real Applications for Requirements Engineering. Deliverable D12: "Final report on user's results" and Deliverable 13: "Final report on supplier's results", October 1996. http://www.info.fundp.ac.be/~phe/2rare.html.

[26] Wieringa R.J., Meyer J.-J.Ch. Actors, Actions and Initiative in Normative System Specification Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

[27] Wing J.M. A specifier's introduction to formal methods. IEEE Computer, pp.8-24, September 1990.

OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*

P. Sánchez, P. Letelier, I. Ramos, Ó. Pastor

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia. Valencia (España)
{ppalma, letelier, iramos, opastor}@dsic.upv.es
http://www.dsic.upv.es/users/oom

Resumen

OASIS v.3.0 es un enfoque formal para la especificación de modelos conceptuales siguiendo el enfoque orientado a objetos. Esta nueva versión supone diversas mejoras y aportaciones que podemos resumir en: (1) definición de un marco formal homogéneo para el modelo OASIS, (2) incorporación de la perspectiva cliente, (3) consideración de un concepto de proceso más ámplio que engloba obligaciones y prohibiciones, (4) enriquecimiento de los mecanismos de definición de clases complejas (agregación y herencia), (5) incorporación de un metanivel que permite abordar aspectos relativos a evolución y reutilización de especificaciones software, y (6) presentación de OASIS como lenguaje con una sintaxis bien definida.

1 Introducción

OASIS (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [14] es un enfoque formal para la especificación de modelos conceptuales siguiendo el paradigma orientado a objetos. Otras propuestas con una motivación similar a la de OASIS son TROLL [12], CMSL [24], LCM [8], Albert [5], Oblog [21] y Gnome [22].

Este artículo presenta las principales características de OASIS 3.0. El modelo OASIS se expone en [14] como un lenguaje de especificación con una sintaxis textual bien definida. Además se establecen los principios formales del lenguaje y se realiza un recorrido exhaustivo de la capacidad expresiva del mismo.

*Este trabajo está finaciado por el proyecto *MENHIR* de la Comisión Interministerial de Ciencia y Tecnología con referencia TIC97-0593-C05-01.

Con respecto de la versión anterior OASIS 2.2 [18], en OASIS 3.0 se han introducido las siguientes mejoras:

- Establecimiento de un marco formal uniforme que cubre todas las secciones de la especificación de una clase.
- Incorporación de la perspectiva cliente. Además de los servicios que puede proveer un objeto, toda solicitud de servicio hecha por un objeto es tratada también como una acción para sí mismo.
- Redefinición del concepto de proceso, distinguiendo entre procesos de obligación (operaciones) y procesos de prohibición (permisos). Las operaciones permiten modelar un comportamiento algorítmico asociado a un servicio no atómico provisto por el objeto. Los protocolos constituyen una extensión al concepto *process* de la anterior versión.
- Enriquecimiento de los mecanismos usados para definir clases complejas: agregación y herencia. Respecto a la primera se han caracterizado con mayor detalle las propiedades asociadas a los distintos tipos de agregación [6, 7, 11]. En herencia se ha introducido una visión más amplia, incluyendo tres formas de herencia [26]: particiones estáticas, particiones dinámicas y roles.
- Extensión del modelo OASIS con un metanivel que permite abordar de forma homogénea y elegante los aspectos de: evolución, reutilización y gestión de configuraciones de software [3].

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99). 1999.

2 Introducción a OASIS 3.0

En OASIS, un **objeto** es un proceso observable cuya vida está caracterizada por la ocurrencia de acciones, tanto si son solicitadas como si son recibidas por el objeto. Así, un objeto puede actuar como cliente o como servidor según esté solicitando u ofreciendo servicios[1]. Una **acción** es una tupla formada por el cliente, el servidor y el servicio solicitado. En la vida de un objeto específico, las acciones cuyo cliente es el mismo objeto son acciones solicitadas. Las acciones cuyo servidor es el mismo objeto son acciones servidas.

Los servicios que un objeto proporciona a nivel "atómico" se denominan **eventos**. Cada objeto tiene un evento de creación (que inicia su vida) y opcionalmente uno de destrucción. Los eventos pueden ser estructurados como **procesos** en un nivel "molecular". Además de la semántica propia del sublenguaje utilizado para especificar procesos, añadiremos una semántica adicional para distinguir entre procesos de obligación y procesos de prohibición. Un proceso de obligación es un servicio de mayor nivel ofrecido por el objeto. Un caso particular de proceso de obligación es cuando, además, se asume que el proceso actúa como "todo o nada" y se denomina **transacción**. Un proceso de prohibición impide la ejecución de determinadas secuencias de acciones en la vida del objeto definiendo las secuencias que están permitidas.

En OASIS, un sistema de información es entendido como una sociedad de objetos autónomos y concurrentes que interactúan entre ellos mediante acciones. Cada objeto puede solicitar sólo los servicios que le son accesibles de otros objetos. Las **interfaces** son un mecanismo que permite establecer relaciones de accesibilidad entre objetos, definiendo la visión que tiene cada objeto respecto de la sociedad de objetos.

Cada objeto encapsula su estado y las reglas que rigen su comportamiento. Como es habitual en todo entorno OO, los objetos pueden ser vistos desde dos perspectivas distintas: estática y dinámica. Desde el punto de vista estático, llamaremos **atributos** al conjunto de propiedades que describen estructuralmente al objeto. Los valores asociados a cada propiedad estructural del objeto caracterizan el **estado del objeto** en un instante dado. La evolución de los objetos (perspectiva dinámica) viene caracterizada por la noción de **cambio de estado**: la ocurrencia de una acción puede generar cambios (definidos por **evaluaciones** y **derivaciones**) en los valores de atributos. La actividad de un objeto está determinada por un conjunto de reglas: **precondiciones**, **restricciones de integridad**, **disparos**, **protocolos** y **operaciones**.

Por otro lado, la vida de un objeto puede representarse como una secuencia de **pasos**. Cada paso está formado por un conjunto de acciones que ocurren en un instante dado de la **vida del objeto.** Las acciones son abstracciones de cambios atómicos en el estado del objeto.

Cada objeto tiene un identificador único (**oid**) proporcionado implícitamente por el sistema. Sin embargo, el objeto es referenciado mediante mecanismos de identificación pertenecientes al espacio del problema. Una función de identificación establece correspondencias entre los mecanismos de identificación y el *oid* del objeto.

Llamamos **tipo** a la plantilla que describe la estructura y el comportamiento de un objeto. Una **clase** se compone de un nombre de clase, una o más funciones de identificación y un tipo. Una clase compleja es aquella definida utilizando otras clases. Las relaciones entre clases disponibles en OASIS son **agregación** y **herencia**.

3 Semántica de OASIS

La semántica de OASIS es dada en términos de una estructura de Kripke (W, τ, ρ). W es el conjunto de todos los mundos[2] posibles que un objeto puede alcanzar. Sea F el conjunto de Fórmulas bien formadas (Fbf) evaluadas sobre el estado (mundo asociado) en el cual se encuentra el objeto, $\underline{A}$ el conjunto "ground" de acciones de la signatura del objeto y $2^{\underline{A}}$, el conjunto de pasos instanciados posibles. Las funciones τ y ρ se definen como:

$$\tau : F \rightarrow 2^W$$
$$\rho : 2^{\underline{A}} \rightarrow (W \rightarrow W)$$

La función τ asigna a una fórmula en la lógica de estado (Lógica de Predicados de Primer Orden) el conjunto de mundos en los cuales se satisface. La función ρ asigna a cada paso una relación binaria entre mundos, la cual es la semántica declarativa del lenguaje. Siendo $\mu \in 2^{\underline{A}}$ un paso y $w, w' \in W$, el significado buscado es: $(w, w') \in \rho(\mu)$ si y sólo si

[1] Los conceptos cliente y servidor son usados en su sentido genérico, no se refieren específicamente a arquitecturas distribuidas cliente/servidor. Conceptos cercanos a lo que hemos preferido denominar "cliente" son: agente [5, 9] y actor [25, 1].

[2] De acuerdo con lo dicho, los estados son aserciones (fórmulas), los mundos son estructuras sobre las que dichas fórmulas son interpretadas.

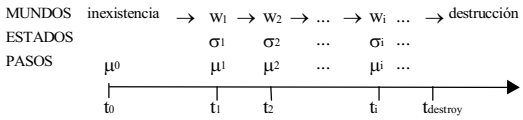


Figura 1: Relación entre mundos, estados y pasos del objeto.

la ocurrencia de μ conduce al objeto desde el mundo w al mundo w'.

Definición 1 ***Transición válida.*** *Cada par* $(w, w') \in \rho(\mu)$ *se denomina transición válida, con* $w, w' \in W$ *y* $\mu \in 2^{\underline{A}}$.

La relación entre estados, mundos, pasos y la vida del objeto se representa en la Figura 1 donde t_i es el tiempo en el cual se produce el i-ésimo *tic* de reloj para el objeto. Obviamente, se cumple que: $\models_{w_i} \sigma_i \ , i = 1, ..., n$.

De forma global, OASIS está basado en Lógica Deóntica [2]: la lógica de las obligaciones, prohibiciones y permisos. En [14] se presentan las fórmulas y especificaciones de proceso utilizadas en cada una de las secciones de una plantilla de clase OASIS y cómo la plantilla completa de la clase se corresponde con fórmulas en una variante de Lógica Dinámica [10] formalizado en [16]. Así, la interpretación de las diferentes fórmulas en Lógica Dinámica, usadas en la plantilla de clase OASIS, es la siguiente:

1. Fórmulas del tipo $\psi \rightarrow [a]\phi$, en particular $[a]\phi$, cuando ψ es *true* (cualquier estado del objeto).

   Siendo $w \in \tau(\psi)$ el mundo actual, si a ocurre entonces el mundo alcanzado es $w' \in \tau(\phi)$. Así, las evaluaciones establecen las propiedades que deben satisfacerse en el mundo alcanzado como efecto atómico de la ejecución de una acción. Dicho de otra forma: $\models_w \psi \wedge \models_{w'} \phi$.

2. Fórmulas de prohibición[3] $\neg\phi \rightarrow [a]false$

   Siendo $w \in \tau(\phi)$ el mundo actual, si a ocurre entonces no existe por definición una transición válida. Es decir, una fórmula de prohibición impide que una acción ocurra estando el objeto en determinados estados: aquellos donde $\models_w \neg\phi$.

[3] Una acción está permitida en un estado si no existe una prohibición que se satisfaga en dicho estado.

3. Fórmulas de obligación $\phi \rightarrow [\neg a]false$

   Siendo $w \in \tau(\phi)$ el mundo actual, si la acción a no ocurre entonces no existe por definición un mundo válido alcanzable. Es decir, una obligación fuerza a que la acción a ocurra cada vez que el estado del objeto es tal que $\models_w \phi$.

4 Especificación de clase

Una especificación OASIS es, esencialmente, un conjunto estructurado de definiciones de clase. Para una clase simple todas las propiedades quedan establecidas en su plantilla. Para una clase compleja, además de las propiedades establecidas por su plantilla, existirán propiedades adicionales determinadas por las relaciones de clase que la definen.

A continuación mostramos un resumen de los aspectos más significativos de OASIS como lenguaje, recorriendo cada una de las partes que forman la plantilla de una clase OASIS.

4.1 Atributos

Los atributos son propiedades estructurales de las instancias de una clase y poseen un tipo asociado. En OASIS se distinguen tres tipos de atributos: *constantes*, *variables* y *derivados*. Para cada atributo definido existirán funciones y operaciones asociadas, disponibles para ser usadas en la especificación de la plantilla y que son útiles para manipular atributos cuya cardinalidad es mayor que uno. Además se puede especificar la representación escogida para atributos multivaluados (**list**, **set** o **bag)**.

4.2 Restricciones de integridad

Las restricciones de integridad son fórmulas basadas en el estado del objeto y que deben ser satisfechas cada vez que se ejecuta una acción independiente (no incluída en una transacción) o cuando se ejecuta la acción que finaliza una transacción. Se pueden clasificar en *estáticas* o *dinámicas*, dependiendo de si se refieren sólo a un estado o relacionan diferentes estados, respectivamente. Para especificar restricciones de integridad dinámicas se utilizan los operadores temporales tradicionales.

Ejemplo 1 *Algunas restricciones de integridad dinámicas:*

- `sometimes {saldo>100000} since {interes>10};`

- `always (> 3 months)`

```
{proyecto_terminado=true}
since {proyecto_inscrito=true};
```

4.3 Eventos

Un evento es la abstracción de un cambio de estado atómico e instantáneo que le puede acontecer a un objeto.

- **new** y **destroy** indican que se trata del evento de creación o de destrucción de instancias, respectivamente. Ambos eventos son únicos para cada clase. El evento **new** tiene como parámetros los valores para los atributos constantes y variables del objeto.

- Distinguimos entre eventos privados (sin la etiqueta **calling to members** o **sharing with members**), eventos implicados (con la etiqueta **calling to members**) y eventos compartidos (con la etiqueta **sharing with members**). Los primeros son servicios ofrecidos o requeridos por un objeto y cuya ocurrencia depende sólo del comportamiento del objeto en sí mismo. Los eventos implicados y los compartidos corresponden a servicios provistos por un objeto agregado, coordinados con servicios en sus objetos componentes. En el primer caso la ocurrencia del evento en el objeto agregado implica necesariamente la ocurrencia del correspondiente servicio en sus objetos componentes. Cuando se trata de eventos compartidos, dicha implicación es en ambos sentidos.

Ejemplo 2 *La clase* ***préstamo****, definida como agregación entre un objeto de la clase* ***libro*** *y uno de la clase* ***socio****, tiene su evento de creación compartido con eventos en sus componentes. Considerando que* ***codigo*** *y* ***numero*** *son los mecanismos de identificación de* ***libro*** *y* ***socio****, respectivamente, en la especificación de la clase* ***préstamo*** *tendremos:*

```
class prestamo
    ...
events
prestar(Fecha:date,NumSocio:nat,NumLibro:nat)

new sharing with members
 libro(codigo,[NumLibro]).ser_prestado(Fecha),

 socio(numero,[NumSocio]).obtener_libro;
    ...
end class
```

4.4 Evaluaciones

Las evaluaciones son fórmulas en lógica dinámica que establecen cómo las acciones afectan el estado del objeto. Los atributos variables modifican sus valores según lo establecido en las evaluaciones. Estas fórmulas dinámicas para evaluaciones son del tipo $\psi \rightarrow [a]\phi$, y se interpretan como: "si en un determinado estado del objeto se satisface ψ y ocurre la acción a, en el estado inmediatamente posterior del objeto se satisface ϕ". Es decir, la condición de evaluación ψ se evalúa y se satisface en el estado en el que ocurre la acción y ϕ se satisface en el instante inmediatamente posterior a la ocurrencia de a.

Ejemplo 3 *Una evaluación que establezca una cierta relación entre* ***deposito(Cantidad:nat)*** *y* ***reintegro(Cantidad:nat)*** *respecto del atributo* ***saldo*** *de una cuenta bancaria.*

```
[deposito(Cantidad)]saldo:=saldo+Cantidad;
[reintegro(Cantidad)]saldo:=saldo-Cantidad;
```

4.5 Precondiciones

En algunas ocasiones no es suficiente la iniciativa de un cliente para que una acción ocurra en el servidor sino que además ciertas condiciones deben satisfacerse en dicho servidor. Si la precondición asociada a una acción a no se satisface en el servidor, en lugar de acontecerle la acción a le acontece la acción $\neg a$ (es decir, cualquier otra acción, incluso ninguna).

En el contexto de lógica dinámica, las precondiciones tienen la forma $\neg\phi\,[a]\,false$, donde ϕ es una Fbf interpretada como una condición para la ocurrencia de la acción indicada. El significado es "si ϕ no se cumple, la ocurrencia de la acción no lleva al objeto a un siguiente estado".

Ejemplo 4 *El evento* ***reintegro*** *en la clase* ***cuenta*** *tiene la siguiente precondición:*

```
reintegro(Cantidad) if {saldo >= Cantidad};
```

4.6 Disparos

Un disparo modela la siguiente situación: si un objeto está en un estado que satisface una condición de disparo entonces debe generarse la obligación asociada. Si el objeto en el cual se establece la obligación es precisamente el cliente de dicha acción, dicha acción ocurre como una solicitud de servicio desde dicho objeto. Si el objeto en el cual existe la

obligación sólo es el servidor de dicha acción, deberá esperar hasta que el cliente de la acción le solicite el servicio de la acción[4].

Para caracterizar la actividad declarada en el párrafo de disparos del lenguaje OASIS utilizamos la fórmula dinámica $\phi[\neg a]false$, cuyo significado es: "si ϕ se satisface y no se ejecuta la acción indicada, entonces el objeto no alcanza un estado válido. En otras palabras, si se cumple ϕ, la acción debe ocurrir para que el objeto alcance un estado siguiente". Consideraremos que se cumple:

- Los disparos, por representar una obligación, deben ser consistentes con los permisos definidos mediante precondiciones y protocolos.
- El disparo establecido por una obligación se mantiene mientras la condición de disparo se satisface, es decir, el objeto continuará solicitando un determinado servicio o esperando por ofrecer cierto servicio, según corresponda.

Ejemplo 5 *Cuando el porcentaje de utilización del disco de un servidor llega al 80%, el objeto de la clase* ***servidor*** *solicita a los usuarios que liberen espacio de disco.*

```
usuario(everyone)::
aviso(fecha,hora,``sin espacio'')
    when {porcentaje_utilizado > 80
        and avisado_80_%=false};
```

4.7 Operaciones y protocolos

Existen tres semánticas posibles que pueden ser asociadas en la especificación de un proceso en OASIS:

(a) La semántica del lenguaje utilizado para su especificación. En OASIS, el lenguaje utilizado para especificar procesos es un subconjunto de CCS [17], compartiendo así una semántica basada en la noción de sistema de transición etiquetado y todas las propiedades formales establecidas en CCS.

(b) La semántica de permisos, es decir, una especificación de proceso establece secuencias de acciones que *pueden* ocurrir.

(c) La semántica de obligaciones, es decir, una especificación de proceso establece secuencias de acciones que *deben* ocurrir.

[4] Esto corresponde al concepto de *action waiting* que se trata en [14] en el capítulo de interacción entre objetos.

Un proceso en OASIS tendrá la semántica (a) junto con la semántica (b) ó (c) dependiendo de si se trata de un protocolo o una operación, respectivamente. Así, la sección *operations* se utiliza para especificar secuencias obligatorias de acciones y la sección *protocols* para especificar secuencias permitidas de acciones. Un caso particular de proceso de obligación es el que actúa como una transacción, es decir, con la política de "todo o nada". En este caso se puede declarar con el calificativo de *transaction* para el proceso. Asumiremos siempre por defecto que el proceso no actúa como transacción.

Existen dos secciones en las cuales se describen especificaciones de proceso: *operations* y *protocols*. Para cada operación o protocolo debe existir un estado que representa el estado de inicio del proceso.

Ejemplo 6 *Un protocolo en la clase* ***máquina****. Se trata de una máquina de chocolatinas que acumula hasta tres monedas como crédito*[5]*.*

```
maq:
 maq   =  moneda.MAQ1;
 MAQ1  =  chocolatina.maq+moneda.MAQ2+
          ::return_moneda.maq[6];
 MAQ2  =  chocolatina.MAQ1+ moneda.MAQ3+
          ::return_moneda.MAQ1;
 MAQ3  =  chocolatina.MAQ2 +
          ::return_moneda.MAQ2;
```

4.8 Clases Complejas

Una clase compleja utiliza en su definición otras clases (simples o complejas). Las relaciones para la definición de clases complejas son agregación y herencia. Consideraremos que se cumple:

- Independientemente de las relaciones entre clases complejas establecidas, la especificación de una clase se hace separadamente de las relaciones entre clases en las cuales pueda participar.
- Diremos que una clase compleja tiene propiedades emergentes si tiene propiedades adicionales a las capturadas implícitamente por la relación

[5] En OASIS, para evitar detallar cada elemento de la tupla $< Cliente, Servidor, Servicio >$ que representa a una acción, se adoptan las siguientes simplicaciones: a) Si el servidor es el propio objeto, se escribe *Cliente*:*Servicio* y si el cliente no es relevante sólo se escribe *Servicio*. b) Si el cliente es el propio objeto, se escribe *Servidor*::*Servicio*. *c)* En el caso particular de comunicación *self*, la acción de requerir se escribe *::Servicio* y la acción de proveer *self:Servicio* (o simplemente *Servicio*).

[6] Permiso para realizar un *action sending*, en el cual tanto el cliente como el servidor son el mismo objeto.

entre clases que la define (caso contrario, la especificación explícita de su plantilla de clase es opcional).

4.8.1 Agregación

Una agregación modela la noción de relaciones estructurales entre objetos. Un caso particular es cuando la agregación representa la noción *parte_ de*. Una asociación (clase cuyas instancias son un grupos de objetos) se representa mediante una agregación multivaluada definida con un sólo componente. Dado que no puede existir comunicación por *action sharing* o *action calling* fuera de la estructura de agregación, el conjunto total de instancias que un objeto de una clase agregada puede referenciar de alguna de sus componentes está restringido a las instancias que forman parte del objeto agregado. Este aspecto es remarcado por la etiqueta **members** en la definición de eventos de la clase agregada.

Ejemplo 7 *Consideremos el objeto agregado* `coche` *cuyos componentes son* `motor`, `ruedas` *(debe tener 4 ruedas y opcionalmente una de repuesto) y el* `chasis` *que representa la carcasa. Esto nos lleva a tener la clase* `coche` *como objeto agregado con componentes* `chasis`, `ruedas` *y* `motor`*:*

```
coche aggregation of
static inclusive chasis
    towards(1,1) from(1,1);
static inclusive motor
    towards(1,1) from(0,1);
dynamic inclusive rueda
    towards(4,5) from(0,1)
list[‘‘adelante izq’’,‘‘adelante der’’,
‘‘atras izq’’,‘‘atras der’’,‘‘repuesto’’];
```

Un `chasis` no puede existir si no está asociado a un `coche` y un `coche` tiene exactamente un `chasis`. Por ser **static** `chasis` forma parte del `coche` de manera permanente y el que sea **inclusive** conlleva que cualquier cosa que se quiera hacer con el `chasis` es a través del `coche`. El `motor` puede existir independientemente de su relación con un `coche`. En cuanto a las `ruedas` se refiere, vemos que pueden existir fuera del contexto de un `coche` pero, sólo pueden formar parte de un `coche`. La pertenencia de una `rueda` a un `coche` puede variar con el tiempo (**dynamic**). La palabra reservada **towards** representa las cardinalidades vistas desde el objeto agregado hacia los componentes. La palabra **from** representa las cardinalidades vistas desde un objeto del componente hacia el agregado.

La asociación es vista como una agregación en la cual sólo existe un componente. Al realizarse la agrupación de instancias en la asociación, la especificación **where** permite seleccionar los objetos de la clase del componente que satisfacen la fórmula especificada. La palabra reservada **grouping by** indica que cada instancia del componente (grupo de objetos) se crea (o destruye) automáticamente agrupando los objetos de la clase componente según los valores de dichos atributos en los objetos de la clase del componente.

Visibilidad desde el agregado al componente Independientemente del tipo de agregación, existe visibilidad de atributos del componente desde el agregado. La coordinación entre actualizaciones de los atributos en el objeto componente respecto de actualizaciones en el objeto agregado se realiza implícitamente mediante eventos compartidos. En lo que a acciones se refiere, si la agregación es inclusiva todas las acciones del componente son también acciones en el agregado. Adicionalmente, usando **calling to members** se pueden definir eventos en el agregado que se corresponden con eventos en los componentes. Si la agregación es relacional cualquier coordinación entre ocurrencia de acciones en el agregado y en el componente debe establecerse mediante especificaciones **calling to members** o **sharing with members**, explícitamente declaradas en la clase agregada.

Para cada componente y atributo definido (con cardinalidad mayor que uno) existirán funciones y operaciones asociadas disponibles para ser utilizadas en las fórmulas de la plantilla. Algunas de estas funciones son **count**, **min**, **sum, position**, **first,** etc.

Ejemplo 8 *El siguiente ejemplo redefine la clase* `coche` *y establece relaciones entre los objetos agregados y sus componentes.*

```
class coche
...
derived attributes
    temperatura_del_coche:int;
    presion_promedio:real;
    gastos_1998:int;
derivations
    temperatura_del_coche:=
        motor.temperatura;
    presion_promedio:=
        avg(rueda.presion)
            where {rueda.instalada=true};
    gastos_1998:=
        sum(reparacion.importe)
          where {año=1998};
...
events
```

```
      arrancar_coche alias for
          motor.encender;
  borrar_reparaciones_un_año(ValorAño:nat)
      calling to members
      reparacion(ref,[ValorAño,_]).eliminar;
  girar_izq
      calling to members
          rueda[‘‘adelante izq’’].girar_izq,
          rueda[‘‘adelante der’’].girar_izq;
  girar_der
      calling to members
          rueda[‘‘adelante izq’’].girar_der,
          rueda[‘‘adelante der’’].girar_der;
  ...
  end class

  class reparacion
  identification
      ref:(año, mes)
  constant attributes
      año:nat; mes:nat;
      importe:real;
  events
      introducir new;
      eliminar destroy;
  ...
  end class



  coche aggregation of
  static inclusive chasis
          towards(1,1) from(1,1);
  static inclusive motor
          towards(1,1) from(0,1);
  dynamic relational reparacion
          towards(0,*) from(1,1);
  dynamic inclusive rueda
          towards(4,5) from(0,1)
  list[‘‘adelante izq’’, ‘‘adelante der’’,
      ‘‘atras izq’’, ‘‘atras
der’’,‘‘repuesto’’];
```

Ejemplo 9 *Dada la siguiente definición para un atributo:*

```
  números:nat list[1..10];.
```

Se podrían definir los siguientes atributos derivados usando la signatura implícita para dicho atributo:

```
  derivations
      sin_numeros:= {count(numeros)=0};
      tiene_numeros_grandes:=
      {(count(numeros)
          where value>100) > 0};
      suma_de_numeros:= sum(numeros);
      maximo_numero:= max(numeros);
      primer_numero:= first(numeros);
```

4.8.2 Herencia

Mediante la herencia podremos especializar (o generalizar) propiedades definidas en las clases. Utilizaremos las particiones estáticas en aquellas especializaciones que son fijas desde la creación del objeto. Por otro lado, definiremos particiones dinámicas para expresar que la partición a la que pertenezca el objeto no es fija y que depende de la ocurrencia de acciones específicas o de los valores de los atributos en un determinado instante. Los cambios de una instancia entre particiones dinámicas corresponden con lo que se denomina **proceso de migración**. Además de las particiones, utilizaremos las clases de rol cuando un objeto pueda desempeñar temporalmente el comportamiento especificado en la clase de rol. En el caso de roles, a diferencia de las particiones dinámicas, un objeto puede desempeñar varios roles de la misma clase de rol y de forma simultánea. Por esto, el objeto rol es un objeto distinto con su propio *oid* pero cuya existencia está supeditada a la existencia del objeto base.

Las acciones de creación de instancias son siempre dirigidas a las *especies*. Una especie es una clase como producto cartesiano entre las particiones de más bajo nivel en la misma jerarquía de clases. Consideramos que existe compatibilidad de comportamiento para las particiones estáticas y dinámicas. En cambio, para los roles consideramos como mínimo compatibilidad de signatura con extensiones tanto horizontales como verticales [23]. Además se considera que la especificación de una clase se hace separadamente de sus relaciones de herencia existentes. Cada especie involucra a varias clases por lo que conlleva la herencia múltiple de las propiedades especificadas individualmente. Para aquellas clases especies que incorporen propiedades emergentes (además de las implícitas por la herencia múltiple) se incluirá explícitamente la plantilla de clase correspondiente.

Ejemplo 10 *La clase (especie)* `camión*diesel` *puede tener como propiedad emergente un atributo que represente la fecha del último cambio de filtro de combustible. Para ello, especificaremos la plantilla de la clase* `camión*diesel` *como para el resto de clases.*

Considerando la compatibilidad de comportamiento en las particiones estáticas y dinámicas, deberán cumplirse ciertas restricciones para la especificación de precondiciones, protocolos, evaluaciones, disparos, operaciones y restricciones de integridad en las subclases definidas.

4.8.3 Particiones estáticas

Las instancias de las subclases definidas por particiones estáticas están asociadas a una partición a partir de su creación y se mantienen en ella durante toda su vida.

Ejemplo 11 *Dos particiones estáticas de una misma clase:*

```
camion, coche, otro_vehiculo
    static specialization of vehiculo;
gasolina, diesel, otro_tipo
    static specialization of vehiculo;
```

Las propiedades de la superclase y de las subclases indicadas en estas jerarquías se definirán separadamente mediante plantillas de clase (cuando sea necesario).

4.8.4 Particiones dinámicas

Un proceso migratorio es aquel por el que una instancia pasar de estar especializada en una subclase de una partición a otra subclase de la misma partición. Se dispone de dos formas alternativas para especificar el proceso de migración: por la ocurrencia de ciertas acciones en determinados estados del objeto, y por partición de los posibles estados del objeto en función de los valores que presenten sus atributos.

Particiones dinámicas por la ocurrencia de acciones específicas Cada partición dinámica expresa las distintas subclases de las que puede formar parte un objeto perteneciente a la clase que se particiona. El poder expresar dicho proceso de migración mediante la ocurrencia de acciones aporta al lenguaje una riqueza expresiva considerable. Las acciones implicadas en el proceso migratorio pertenecen a la subclase de la partición que se abandona. Por defecto, el **new** de las instancias es el servicio en la acción indicada al comienzo del proceso de migración.

Ejemplo 12 *Una especialización dinámica de la clase* `coche` *determinada por la ocurrencia de* `crear_coche`, `reparar` *y* `estropear` *puede ser:*

```
funcionando, estropeado
    dynamic specialization of coche
migration relation is
    coche = crear_coche.funcionando;
    funcionando = estropear.estropeado;
    estropeado = reparar.funcionando;
```

Según lo dicho, la creación de una instancia de `coche` implica comenzar perteneciendo a la partición `funcionando`. Cuando sea una instancia de `funcionando` le ocurrirán acciones de la signatura de `coche` más, posiblemente, otras de la subclase `funcionando`. Entre ellas está la acción `estropear` que pertenece a la signatura de `funcionando`. Su ocurrencia implica salir del proceso que representa a `funcionando` y entrar en el indicado en el proceso migratorio, esto es, la subclase `estropeado`.

Partición dinámica en función de valores de atributos Dado que en este tipo de partición el proceso migratorio se define en función el estado de los objetos, cuando el objeto alcanza un nuevo estado puede implicar su migración de una subclase a otra en la partición. La modificación del estado del objeto determina el proceso de migración entre las subclases de la partición dinámica. Además, la partición que se haga del conjunto de estados a partir de los valores de los atributos debe ser completa y disjunta.

Ejemplo 13 *Una partición dinámica de la clase* `cuenta` *en función del atributo* `saldo` *puede ser:*

```
no_rentable where {saldo<100000},
medio_rentable where {saldo>=100000
        and saldo<1000000},
muy_rentable where {saldo>=1000000}
        dynamic specialization of cuenta;
```

Una acción de creación debe especificar en qué partición estática se sitúa la nueva instancia. La subclase de la partición dinámica se determina implícitamente a partir del valor del atributo `saldo` incluido en el proceso de creación.

4.8.5 Roles

Mediante roles representamos conductas distintas de un objeto. Una clase puede tener asociada diferentes subclases de rol, cada una representando un patrón de conducta específico para un objeto en dicha subclase. El objeto sigue siendo instancia de una sola clase pero puede desempeñar distintos roles a lo largo de su existencia. Las instancias de una subclase de rol pueden ser creadas y destruidas. Un objeto puede estar asociado al mismo tiempo a dos o más instancias de roles, pudiendo desempeñar varias instancias de la misma subclase de rol simultáneamente.

Ejemplo 14 *Un ejemplo de roles definidos sobre la clase* `persona`.

```
estudiante towards(0,1) ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

Se ha querido representar con las cardinalidades el que una instancia de la clase `persona` puede desempeñar, simultáneamente, como máximo `10` *roles* de `empleado` o, alternativamente, como máximo un rol de `estudiante`. También es posible que una instancia de `persona` no desempeñe ningún rol.

El evento de creación es enviado a la clase `persona`. Posteriormente, si ocurre el evento `ser_matriculado` (que representa el **new** de `estudiante`), entonces la `persona` pasa a desempeñar el rol de `estudiante`. Si destruimos el objeto de `persona`, automáticamente se destruye el objeto `estudiante`, lo contrario no se cumple.

Ejemplo 15 *Consideremos la partición dinámica de* `persona` *junto a los roles definidos sobre la misma clase en el ejemplo anterior:*

```
niño where {edad<14},
adolescente where {14<=edad and edad<18},
adulto where {18<=edad}
    dynamic specialization of persona;

estudiante towards(0,1) ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

En este caso, la existencia del rol no supone aumentar el número de especies dado que es un nuevo objeto el que se crea cuando se comienza a desempeñar el rol. En cambio, a futuras particiones (estáticas o dinámicas) de la clase de rol sí podrían dar lugar a nuevas especies.

4.8.6 Creación y destrucción de instancias

En las particiones estáticas, dinámicas y roles, cada subclase hereda el conjunto de mecanismos de identificación definidos en sus superclases. Siempre existirá la posibilidad de definir mecanismos de identificación a los por defecto heredados. La creación de un nuevo objeto debe hacerse dentro de la especie a la que va a pertenecer. La acción de creación va dirigida a la especie como intersección de todas las clases en las que el nuevo objeto se especializa. Se consideran los siguientes casos con vistas a la creación de objetos:

1: La especie es intersección únicamente de subclases de particiones estáticas. En este caso, se deben especificar todas las subclases involucradas dado que quien crea el objeto posee toda la información necesaria para determinar la especialización concreta. Las acciones de creación deben dirigirse a las especies completas, es decir a `camión*diesel, coche*gasolina`, etc.

2: La especie es intersección de subclases de especializaciones dinámicas y opcionalmente estáticas. En este caso, se deben especificar todas las subclases estáticas involucradas si bien en las dinámicas distinguimos dos situaciones:

   – Aquellas subclases de especializaciones dinámicas cuyo proceso migratorio está definido sobre atributos. En este caso el modelo automáticamente deduce en qué subclase de dicha partición dinámica se especializa.

   – Aquellas subclases de especializaciones dinámicas cuyo proceso migratorio está definido sobre las acciones de migración. En este caso si sólo existe un punto de entrada en el proceso migratorio, la acción de creación será la indicada como comienzo del proceso y el modelo lo deduce automáticamente. En cambio, si el proceso de migración tiene varios puntos de entrada, se deberá explicitar la clase de comienzo, y de la misma forma que antes, la acción de creación será la incluida en la subclase dinámica indicada.

Dado que se pueden definir especializaciones estáticas y dinámicas de las clases de rol, la acción de creación del rol involucra también la creación del objeto en alguna de las especies de intersección aplicándose el mecanismo descrito anteriormente.

4.9 Interfaces

Una interfaz es una clase virtual, esto es, una vista de una clase definida por una proyección de la plantilla de una clase. Esta nueva plantilla establece las propiedades de los objetos de la clase original (servidores) que pueden ser usadas por determinados objetos clientes. Así, una interfaz también establece una relación de acceso o visibilidad entre clientes y servidores. Las interfaces permiten al diseñador proteger a una clase de usos no deseados, establecer visibilidad ajustada a las necesidades de cada objeto cliente, y establecer relaciones de acceso con el entorno del sistema.

Las interfaces definidas deben ser consistentes con las relaciones de acceso establecidas en la plantilla de cada clase al especificar acciones. Un objeto

cliente no puede solicitar un servicio a un determinado objeto servidor si no existe la correspondiente interfaz que establezca dicha relación de acceso al servicio.

Toda solicitud de servicio desde un cliente a un determinado servidor, especificada en una plantilla de clase, debe poderse deducir desde alguna definición de interfaz. El conjunto de servicios que un cliente puede requerir se obtiene inspeccionando todas las interfaces que se aplican al objeto como cliente. Del mismo modo, los clientes potenciales que pueden requerir un determinado servicio se obtienen inspeccionando todas las interfaces en las cuales se da acceso a dicho servicio.

La interfaz puede establecer relaciones de acceso entre objetos y/o conjuntos de objetos, dependiendo de que la referencia es a un objeto individual, grupo de objetos de una clase o todos los objetos de una clase.

Ejemplo 16 *Una interfaz para el usuario identificado por el valor en* `nombre=''Juan Pérez''`*, encargado de modificar el precio de los artículos del inventario.*

```
interface usuario(nombre,[''Juan Perez''])
            with articulo(someone)
    attributes(all)
    services(cambiar_precio)
end interface
```

5 Conclusiones

Este artículo presenta la versión 3.0 del lenguaje OASIS en el que se ha conseguido integrar tanto los aspectos formales que justifican el modelo como una sintaxis BNF precisa del mismo. Este trabajo se enmarca dentro del proyecto MENHIR en el cual destacamos las líneas de trabajo: (1) animación de especificaciones OASIS usando Redes de Petri Orientadas a Objetos (véanse [19, 20]) y Programación Lógica Concurrente (véanse [13, 15]) y, (2) la definición e implementación de un entorno de especificación y evolución de esquemas conceptuales (véase [4] donde se presenta la interfaz gráfica de usuario de una herramienta que utiliza la metaclase como núcleo de la misma y que permite la definición, validación y evolución de los esquemas conceptuales).

Referencias

[1] Agha G.A. *ACTORS: A model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[2] Åqvist L. Deontic logic. In D.M. Gabbay and F.Guenthner, editors, Handbook of Philosophical Logic II, pages 605-714. Reidel, 1984.

[3] Carsí J.A., Ramos I., Penadés M.C., Pelechano V. *Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic*. I Jornadas de Trabajo en Ingeniería del Software, Sevilla, 1996.

[4] Carsí J.A., Camilleri S., Canós J.H., Ramos I. *Propuesta de Interfaz Gráfica de Usuario homogénea para el Diseño y la Explotación de Sistemas de Información*. JIS'98, III Jornadas de Trabajo en Ingeniería del Software, Murcia, Noviembre, 1998.

[5] Dubois E., Du Bois P., Petit M. *O-O Requirements Analysis: an agent perspective*. In Proc. of the 7th.European Conference on Object Oriented Programming- ECOOP 93, pages 458-481, 1993.

[6] Ehrich H.-D., Goguen J.A., Sernadas A. *A Categorical Theory of Objects as Observed Processes*. LNCS 489, 1990.

[7] Ehrich H.-D. *Objects Specification*. Abteilung Datenbanken, Technische Universität, Braunschweig, Germany, 1995.

[8] Feenstra R.B., Wieringa R.J. *LCM 3.0: a languaje for describing conceptual models*. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[9] Genesereth M.R., Ketchpel S.P. *Software Agents*. Communications of the ACM, 37(7): pages 48-53, 1994.

[10] Harel D. *Dynamic Logic*. In Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pages 497-694, Reidel 1984.

[11] Hartmann T., Junglaus R., Saake G. *Aggregation in a Behaviour Oriented Object Model*. Abt. Datenbanken, TU Braunschweig, 1992.

[12] Jungclaus R., Saake G., Hartmann T., Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems*.

ACM Transactions on Information Systems, Volume 14, Number 2, pages 175-211, 1995.

[13] Letelier P., Sánchez P., Ramos I. *Animation of systems specifications using concurrent logic programming.* Symposium on Logical Approaches to Agent Modeling and Design, ESSLLI'97, Aix-en-Provence, France, 1997.

[14] Letelier P., Ramos I., Sánchez P., Pastor O. *OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto.* Servicio de Publicaciones de la Universidad Politécnica de Valencia (SPUPV-98.4011), ISBN 84-7721-663-0, Universidad Politécnica de Valencia, 1998. `www.dsic.upv.es/users/oom/books.html`.

[15] Letelier P., Sánchez P., Ramos I. *Prototyping a requirements specification through an automatically generated concurrent logic program.* First International Workshop on Practical Aspects of Declarative Languages (PADL'99), Texas, USA, Enero 1999.

[16] Meyer J.-J.Ch. *A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic.* In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.

[17] Milner R. *Communication and Concurrency.* Prentice Hall Series in Computer Science, C.A.R. Hoare, Series Editor, 1989.

[18] Pastor O., Ramos I. *OASIS versión 2 (2.2) : A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach.* SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.

[19] Sánchez P., Letelier P., Ramos I. *Constructs for prototyping information systems with Object Petri Nets.* Proceeding of the IEEE International Conference on SMC'97, pages 4260-4265, Orlando, 1997.

[20] Sánchez P., Letelier P., Ramos I. *Una Aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto.* III Jornadas de Ingeniería del Software (JIS'98), Noviembre, Murcia, España, 1998. (aceptado)

[21] Sernadas C., Gouveia P., Sernadas A. *Oblog: Object-Oriented, logic-based conceptual modelling.* Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, 1096 Lisboa, Portugal, 1992.

[22] Sernadas C., Ramos J. *Gnome: Sintaxe, semântica e cálculo.* Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, 1096 Lisboa, Portugal, 1994.

[23] Wegner P., Zdonik S. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like.* LNCS 322, ECOOP pages 55-77, 1988.

[24] Wieringa R.J. *A conceptual model specification language (CMSL Version 2).* Vrije U., The Netherlands, 1992.

[25] Wieringa R.J., Meyer J.-J.Ch. *Actors, Actions and Initiative in Normative System Specification.* Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

[26] Wieringa R., Jonge W., Spruit P. *Roles and dynamic subclasses: a modal logic approach.* Vrije U., The Netherlands, 1995.

Prototyping a requirements specification through an automatically generated concurrent logic program*

Patricio Letelier Pedro Sánchez Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology, 46020 Valencia (Spain)
{letelier, ppalma, iramos}@dsic.upv.es

Abstract. OASIS is a formal approach for the specification of object oriented conceptual models. In OASIS conceptual schemas of information systems are represented as societies of interacting concurrent objects. Animating such models in order to validate the specification of information systems is a topic of interest in requirements engineering. Thus a basic execution model for OASIS specifications has been developed. Concurrent Logic Programming is a suitable paradigm for distributed computation allowing a natural representation of concurrence. Using Concurrent Logic Programming, OASIS specifications are animated according to OASIS execution model. In this work, we show how OASIS concepts are directly mapped into concurrent logic programming concepts. To illustrate our ideas, an example of a bank account codified in KL1 is given and parts of the program that animates its corresponding OASIS specification are shown. This work has been developed in the context of a CASE tool supporting the OASIS approach. Our aim is to build a module for animation and validation of specifications. A preliminary version of this module is presented.

1 Introduction

Conceptual models, representing the functional requirements of information systems, are a key factor when linking the problem and solution domains. Building a conceptual model is a discovery process, not only for the analyst but also for the stakeholders. The most suitable strategy in this situation is to build the conceptual model in an iterative and incremental way, through analyst and stakeholder interaction. Conceptual modeling involve four activities: elicitation of requirements, modeling or specification, verification of quality and consistency, and eventually, validation.

Formal methods for conceptual modeling provide improvements in soundness and precision for specifications, simplifying their verification. However, when considering elicitation and requirements validation, prototyping techniques are more used. Hence, it is interesting to obtain a combination of both approaches.

* This research is supported by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) through the MENHIR proyect (grant no. TIC97-0593-C05-01).

Letelier, P., Sánchez, P., Ramos, I. Prototyping a requirements specification through an automatically generated concurrent logic program. Proc. of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99). 1999.

This work uses OASIS [9, 13] (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) as a formal approach for object-oriented conceptual specification of information systems. This is a step forward in a growing research field where validation of formal specifications through animation is being explored [18]. In this sense, some other proposals close in nature to OASIS are [7] and [6]. The differences, though, between these works and ours are basically determined by features of the underlying formalisms and the offered expressiveness . According to the presented results, the state of the art is similar and is characterized by preliminary versions of animation environments.

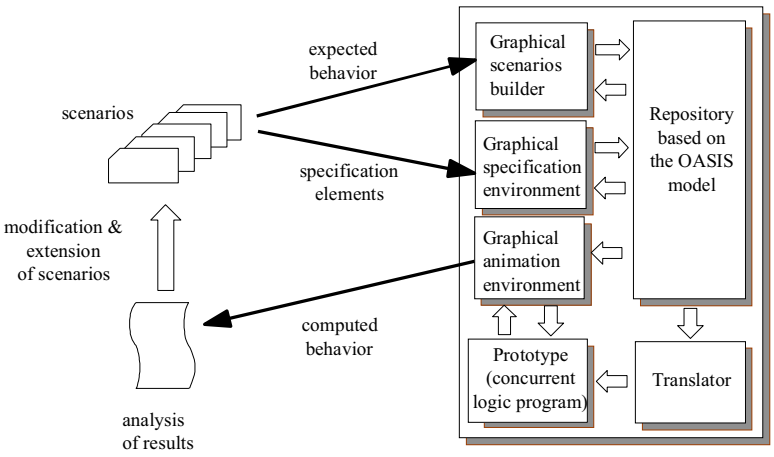


Fig. 1. A framework for incremental specification of requirements

Fig.1 shows our framework for elicitation, modeling, verification and validation of requirements. Elicitation is achieved using scenarios [15]. The expected behavior and elements of a given specification are extracted from scenarios. The graphical scenario builder helps defining scenarios in a suitable way. Functional requirements are modeled using a graphical specification module based on OASIS. Conceptual models can be verified according to OASIS formal properties. At each stage of the requirements specification process it would be possible to validate the behavior of the associated prototype against the expected behavior. This comparison could lead to updates or extensions of existing scenarios. This cycle continues until the requirements are compliant with the proposed set of scenarios.

Experiments have been carried out using Object Petri Nets [16] and Concurrent Logic Programming [8] as semantic domains for OASIS specifications. These efforts have resulted in a basic execution model [10]. This model is used to animate OASIS specifications implemented over concurrent programming environments.

In this context, we will present the mappings between OASIS concepts and clauses in concurrent logic programming. Those mappings have been implemented in a translator program. The translator takes an OASIS specification stored in the repository and generates automatically a concurrent logic program that constitutes a prototype for the corresponding conceptual model. Furthermore, through a preliminary version of the graphical animation environment the analyst can interact with the prototype in a suitable way. We have worked with the concurrent logic languages Parlog [3] and KL1[2] which have similar features. The implementation showed in this work is KL1 code.

2 OASIS

An OASIS specification is a presentation of a theory in the used formal system and is expressed as a structured set of class definitions. Classes can be simple or complex. A complex class is defined in terms of other classes (simple or complex). Complex classes are defined by establishing relationships among classes. These relationships can be aggregation or inheritance. A class has a name, one or more identification mechanisms for its instances (objects) and a type or template that is shared by every instance belonging to the class. We present next the basic concepts of OASIS.

Definition 1. ***Template or type****. A class template is represented by a tuple* $\langle Attributes, Events, Formulae, Processes \rangle$.

Attributes is the alphabet of attributes. For all $att \in Attributes$ exists the function:

$$att : sort\ of\ names \rightarrow sort\ of\ values$$

Events is the alphabet of events. For all $e \in Events$ is possible to get $\underline{e} = \theta e$ being θ a basic substitution of the parameters of the event. $Formulae$ is a set of formulae which are organized in sections and their underlying formalism depends on the section where they are used. *Processes* is the set of process specifications, classified in protocols and operations.

Definition 2. ***Service****. A service is either an event or an operation. The former is an instantaneous and atomic service. An operation is a non-atomic service and, in general, has duration.*

Definition 3. ***Action****. An action is a tuple* $\langle$*Client,Server,Service*$\rangle$*. It represents the action in the client object, associated to requiring the service, as well as the action in the server object, associated to providing the service.*

For each class we assume the implicit existence of A, a set of actions obtained from the services that objects in the class can request (clients) or provide (servers). For all $a \in A$ is possible to obtain $\underline{a} = \theta a$, being θ a basic substitution of client, server and service.

Definition 4. ***Object state***. *An object state is a set of evaluated attributes. It is expressed by well-formed formulae in First Order Logic.*

Definition 5. ***Step***. *A step is a set of actions occurring simultaneously in an object's life.*

Definition 6. ***Object life or trace***. *An object's life or trace is a finite prefix of object steps.*

2.1 OASIS expressed in Dynamic Logic

In [11] Deontic Logic [1] is described as a variant of Dynamic Logic [5]. The definition of deontic operators in Dynamic Logic is:

$\psi \rightarrow [a]false$ "the occurrence of a is forbidden in states where ψ is satisfied".
$\psi \rightarrow [\neg a]false$ "the occurrence of a is obligatory in states where ψ is satisfied".
$\psi \rightarrow [a]\phi$ "in states where ψ is satisfied, immediately after of the a occurrence, ϕ must be satisfied"

where ψ is a well-formed formulae that characterizes an object's state when the action a occurs and $\neg a$ represents the non-occurrence of the action a (i.e., only other actions different from a could occur). Furthermore, there is no state satisfying the atom $false$. This represents a state of system violation. Thus, one action is forbidden if its occurrence leads the system towards a violation state, and one action is obligatory if its non-occurrence leads the system towards a violation state. The OASIS *Formulae* and *Processes* are mapped to the formulae previously presented.

These formulae constitute a sublanguage of the language proposed and formalized in [20]. In [9] OASIS is presented as a specification language with a well defined syntax. Here is an example of part of a simple bank system using the OASIS syntax. This example will be used in the rest of the paper.

```
conceptual schema simple_banking_system
class account
identification
    number:(number);
constant attributes
    number:nat; name:string;
variable attributes
    balance:nat(0); times:nat(0); pin:nat(0); rank:nat(0);
derived attributes
    good_balance:bool;
derivations
    good_balance:={balance>=100};
events
    open new; close destroy;
```

```
    deposit(Amount:nat);
    withdraw(Pin:nat,Amount:nat);
    pay_commission;
    change_pin(Pin:nat,NewPin:nat);
    change_rank(Rank:nat);
valuations
    [deposit(Amount)] balance:=balance+Amount, times:=times+1;
    [withdraw(Pin,Amount)] balance:=balance-Amount, times:=times+1;
    [self:pay_commission] balance:=balance-1;
    [::pay_commission] times:=0;
    [change_pin(Pin,NewPin)] pin:=NewPin;
    [change_rank(Rank)] rank:=Rank;
preconditions
    withdraw(Pin,Amount) if (pin=Pin and balance>=Amount) or
                    (pin=Pin and balance<Amount and rank=2);
    change_pin(Pin,NewPin) if (pin=Pin);
    close if (balance=0);
triggers
    self::pay_commission when
                    (times>=5 and good_balance=false and rank=0);
end class

class customer
identification
    name:(name);
constant attributes
    name:string;
events
    add new; remove destroy;
end class

interface customer(someone) with account(someone)
    services(deposit,withdraw,change_pin);
end interface

interface account(someone) with self
    services(pay_commission);
end interface
end conceptual schema
```

In this example there are two classes: `customer` and `account`. The objects in both classes are active. An `account` object is forced to self-trigger an action with the event `pay_commission` whenever the trigger condition is satisfied. Although `customer` objects do not have explicit triggers, they have an interface with `account` objects enabling to require actions associated with the visible events. Thus `customer` objects are active objects as well. Furthermore, by default, there is always an object called `root`. This object will be responsible for activating the events that do not have an explicit client in the specification. In the example, the `root` object can require actions with the event `add` and `remove` to the `customer`.

2.2 An execution model for OASIS

Our execution model is an abstract animator for formulae of obligation, permission and change of state associated to an object. Next we briefly describe the concepts included in the execution model proposed in [10].

The sequence of steps in an object's life is sorted by time. We assume there is a "clock object" sending special actions — called *ticks* — to every object in the system. The received *ticks* by an object are correlative with natural numbers, t_1, t_2, etc. Hence, being i and j natural numbers then $i < j \iff t_i < t_j$.

Definition 7. ***Mailbox.*** *A mailbox is the set of actions that can be included in a step executed by one object at one tick. The mailbox at instant t_i is denoted by $Mbox_i$.*

Definition 8. ***Object's state at tick.*** *An object's state at tick is denoted by $State_i$ and represents the object's state in the interval $[t_i, t_{i+1})$. That is, between t_i (included) and t_{i+1} the state is considered constant.*

The processing of the actions inside the mailbox implies their classification. Next we present all possible actions that might be present in a mailbox. They are characterized as subsets of $Mbox_i$ (that is at instant t_i) .

- **Obligated actions**: these are actions associated to obligated service requests (which have as a client the object itself) and **have to** occur. The set of obligated actions is denoted by $OExec_i$. These actions are determined by obligation formulae, that is, their form is: $\phi[\neg a]false$.
- **Non-obligated actions:** these are actions corresponding to services requested by other objects (or itself) which **could be** provided or not depending on prohibitions established over those actions and verified in $State_i$. The set of non-obligated actions is denoted by $\overline{O}Exec_i$.
- **Rejected actions**: these are non-obligated actions whose occurrence is prohibited in $State_i$. The set of rejected actions is denoted by $Reject_i$. These actions are determined by prohibition formulae, that is, their form is: $\phi[a]false$.
- **Candidate actions:** these are non-obligated actions whose occurrence is permitted in $State_i$. The set of candidate actions is denoted by $Cand_i$.
- **Executed actions:** these are actions forming the step executed at t_i. The set of executed actions is denoted by $Exec_i$. A step is composed by $OExec_i$ joined with a subset of $Cand_i$.
- **Actions in conflict:** These are a subset of $Cand_i$ formed by actions in conflict[1] with some obligated or candidate action just selected. The set of actions in conflict is denoted by $Conf_i$.

A simple criterion is used in order to choose actions from $Cand_i$: when two actions are in conflict, the action which first arrived to the mailbox will be chosen. The actions in conflict $Conf_i$ are "copied" to the next mailbox ($Mbox_{i+1}$).

[1] Two actions are in conflict if they could change the value of non-disjoint set of attributes.

The object behavior is characterized by an algorithm (detailed in [10]) that manipulates each mailbox (at each *tick*) obtaining the subsets previously defined and producing the change of the object state.

3 Concurrent logic programming and OASIS

Concurrent logic languages arise as an attempt to improve the efficiency of logic languages by exploiting the stream AND parallelism. Besides, they are high level programming languages and very convenient for parallel and distributed systems. A concurrent logic program is a set of Horn Clauses with Guards.

$$H \leftarrow G_1, ..., G_n : B_1, ..., B_m \qquad n, m \geqslant 0$$

A goal has the following form:

$$\leftarrow M_1, ..., M_k \qquad k > 0$$

All M_i are evaluated in parallel (AND parallelism), using the program clauses for their reduction. For each M_i the clauses that can reduce it are examined in parallel (OR parallelism), selecting only one and avoiding backtracking. The criterion of selection is that the M_i unifies with the head of the clause, and the conjunction of guards $G_1, ..., G_n$ will be satisfied (evaluated also using AND parallelism, if they exist). If more than one clause could be chosen, then a sequential search can be established in textual order from top to bottom.

The integration of Concurrent Logic Programming and the OO modeling has generated a great deal of research. We are interested in modeling objects as perpetual processes according to the OASIS execution model. Modeling objects as perpetual processes is an approach initiate by Shapiro and Takeuchi [17], in which an object is implemented as a tail-recursive process that passes the update state of the object as arguments in the recursive call. The identity of the object is the name of an input stream argument of the process. Works in this direction are principally based on making OO extensions to concurrent logic languages. Within this approach some proposals are: Polka [4], L2||O2 [14] and A'UM [19]. Although some implementation aspects are common, our motivation is to generate automatically a concurrent logic program corresponding to an OASIS conceptual model.

3.1 Objects and classes in concurrent logic programming

Now we will sketch the essential features that allow considering an OASIS specification as a KL1 program. Details of intermediate clauses and clause bodies will be omitted to facilitate their reading. In Concurrent Logic Programming a

society of objects can be seen (at run time) as a goal to solve, where each object is a subgoal. Each object is evaluated using AND parallelism (inter-object concurrence).

$$\leftarrow object_1(In_1, Out_1, State_1), ..., object_k(In_k, Out_k, State_k) \qquad k > 0$$

In Concurrent Logic Programming, the partial instantiation of logical variables enables to use them as communication channels. In_i is the input channel for receiving actions, it is a merger of the messages received from itself and from other objects. Also, In_i is used as the object *oid*, so anybody knowing the value of *oid* could instantiate partially this variable, that is, send an action to that object. Out_i is the output channel for sending actions. $State_i$ is the list of terms $att(Attribute, value)$ for each attribute of the object.

In general, creation (and destruction) of objects at run time is required. Subgoals that have the capacity of generating (in their reduction) a new instance of an object should exist. These subgoals correspond in a natural form to the notion of class. Thus, classes are implemented as other objects of the society. A class has an attribute called *Population*, which is a list of pairs (In_i, Key) for each class instance. In_i is the object *Oid*, Key is a list of constant attributes that allows referring to the object. The predicate name of a class goal will be the name of the corresponding class. For an object goal the predicate name will be the name of its class with "`o_`" at the beginning.

Example 1. The class `account` in our bank system (at run time) is represented as a goal in Concurrent Logic Programming. Classes are goals at the beginning of the execution. Thus the class `account` appears as the following subgoal:

```
:- ...,account(In,Out,[att(population,[])]),...
```

3.2 Object behavior

Each object attempts to reduce itself using the clauses that represent its specification. Each clause recognizes one action and is able to reduce the object to a set of subgoals. Below an object goal and clauses with which this object goal could be reduced are shown. Considering $p, q, s, t \geqslant 0$ and $r > 0$:

$$: -..., o\_class_i(In^*, Out^*, State^*), ...$$

$$o\_class_i([action_{i1}|In], Out, State) : -$$
$$G_{i1}, ..., G_{ip}|$$
$$B_{i1}, ..., B_{it},$$
$$o\_class_i(In, NOut, NState).$$

...
$o\_class_i([action_{ir}|In], Out, State) :-$
$G_{i1}, ..., G_{iq}|$
$B_{i1}, ..., B_{is},$
$o\_class_i(In, NOut, NState).$

In^*, Out^* and $State^*$ represent the logical variables In, Out and $State$ at a given moment in the reduction of the goal o_class_i.

Object creation and destruction Classes are implemented as subgoals inside the initial goal. We obtain the instances by means of reduction of class goals whenever the action occurs with the event of creation. The creation of an object implies that the following subgoals are generated during the class goal reduction:

...
$NOut = \{Out, ObjectOut\}$,
$o\_class(Oid, ObjectOut, Attributes)$,
...

Out is the output channel for the class goal. This channel is separated into two new logical variables. $NOut$ will be used as the new output channel for the class goal. $ObjectOut$ will be the output channel for the object created. Oid is another new logical variable that will be used as object Oid and input channel.

Example 2. The clause to which the class `account` reduces when the action of creation occurs (action with the `open`[2] event). In the body of that clause there is the creation of a new object (goal) `account`.

```
account([action(Client,s(account),e(open,Attributes))|RestActions],
            Out,State) :-
    Out={NOut,ObjectOut},
    o_account(Oid,ObjectOut,Attributes),
    get_attributes(Attributes,[att(number,Number)]),
    NPopulation=[object(Oid,Number)|Population],
    update_state(State,[chg(population,NPopulation)],NState),
    ...
    account(RestActions,NOut,NState).
```

`get_attributes` is a predicate that extracts some attribute values from a list. `update_state` modifies a list of attributes producing a new one. When the event `open(101,john,0,0,1234,0,false)` is received the goal `account` becomes:

```
account(action(Client,c(account),
    e(open,[101,john,0,0,1234,0,false]))|Rest],Out,State)
```

This goal is reduced in the following two subgoals, that is, a new object `account` has been created.

[2] An event of creation has implicitly the constants and variable attributes of the object as arguments.

```
...
o_account(Oid,ObjectOut,[att(number,101),att(name,john),
    att(balance,0),att(times,0),att(pin,1234),
    att(rank,0),att(good_balance,false)]),
account(Rest,NOut,),[att(population,[object(Oid,101)]]).
```

The destruction of an object is obtained by the reduction carried out by selecting a clause whose body does not contain the same object as subgoal. Hence, the execution of the object ends.

Change of state We say that an object is implemented as a perpetual process because among the subgoals in which an object is reduced the same object appears. This produces the effect of continuity in the object life. Whenever the object goal is thrown as subgoal in the reduction, some of its attributes may be modified. Thus a change of state due to the occurrence of the associate action is represented. The effect "to execute action" is obtained in the reduction when the new input channel is used as the original one without considering the last executed action. The formulae that define the change of state when the event is executed are subgoals that assign new values to the attributes of the object.

Example 3. The execution of the action associated with the event `deposit(10)` sent to the recently created object is represented as the reduction of the goal:

```
o_account([action(Client,s(account,id(number,[att(number,101)])),
    e(deposit,[10]))|RestIn],Out,[att(number,101),att(name,john),
    att(balance,0),att(times,0),att(pin,1234),att(rank,0),
    att(good_balance,false)])
```

This goal is reduced using the following clause:

```
o_account([action(Client,Server,e(deposit,[N]))|Rest],Out,State) :-
    get_attributes(State,[att(balance,Balance),att(times,Times)]),
    NBalance:= Balance+N,
    NTimes:= Times+1,
    LExp1:= NBalance,
    RExp1:= 100,
    test_condition([[c(ge,LExp1,RExp1)]],NGood_balance),
    update_state(State,[chg(balance,NBalance),chg(times,NTimes),
    chg(good_balance,NGood_balance)],NState)
    o_account(Rest,Out,NState).
```

`test_condition` is a predicate that evaluates a list of conjunctions representing a well-formed formulae in the specification. If all the conjunctions are satisfied the second argument of `test_condition` is instantiated to *true*, otherwise it is $false$. Thus, in this case, the goal is reduced to the following goal, in this way, the object has changed its state.

```
o_account(Rest,Out,[att(number,101),att(name,john),att(balance,10),
    att(times,1),att(pin,1234),att(rank,0),att(good_balance,false)])
```

Prohibitions Action preconditions are implemented through intermediate predicates in the body of the object clauses.

Example 4. Here is the clause that verifies preconditions when the event `withdraw(1234,10)` is attended.

```
check_one_o_account_condition(action(Client,This,e(withdraw,[P,N])),
            State,Result,Msg) :-
    utility:get_attributes(State,[att(balance,Balance),
            att(rank,Rank),att(pin,Pin)]),
    LExp1:= Balance, RExp1:= N,
    LExp2:= Balance, RExp2:= N,
    LExp3:= Rank, RExp3:= 2,
    LExp4:= P, RExp4:= Pin,
    test_condition([[c(eq,LExp4,RExp4),c(ge,LExp1,RExp1)],
        [c(eq,LExp4,RExp4),c(lt,LExp2,RExp2),c(eq,LExp3,RExp3)]],
        Result),
    Msg=not([[c(eq,LExp4,RExp4),c(ge,LExp1,RExp1)],
        [c(eq,LExp4,RExp4),c(lt,LExp2,RExp2),c(eq,LExp3,RExp3)]]).
```

This time, the predicate `test_condition` is used to instantiate to *true* or *false* the variable `Result`. When the precondition is not satisfied another action is sent to the client including the message `Msg`.

3.3 Inter-object Communication

The communication mechanism offered by Concurrent Logic Programming is to share variable among subgoals, interpreted as a communication channel among objects. In order to allow many objects to communicate with a determined object, the architecture showed in Fig.2 has been implemented.

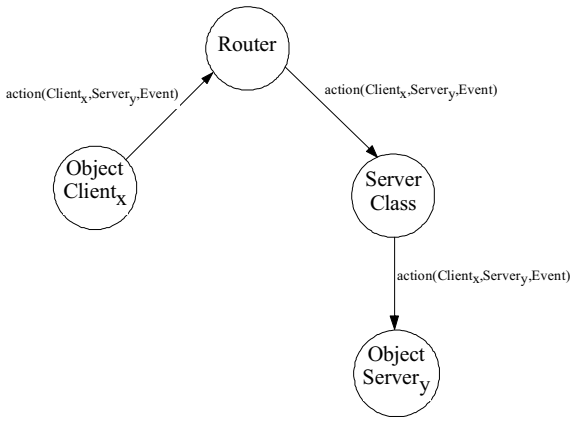


Fig. 2. Communication between objects

In general, if a client object $client_x$ has to send an action to a server object (or itself) $server_y$, $client_x$ instantiates partially its variable *Out*. The *Out* variable is merged with the rest of the output variables of objects and is used as input variable for an additional goal called *router*. The *router* goal has as many output variables as classes in the system. Whenever the goal *router* receives an action in its input, it will put the action in the corresponding output channel according to the server of the action.

The first goal in a KL1 program is the atom `main`. Thus, in the body of this predicate the global schema of communication is established.

Example 5. The main predicate defined for our bank system.

```
main:-
    take_active_actions(ExternActions),
    UserActions=[action(c(tester),s(user),e(add,[att(userid,root)]))
        |InUser],
    user(InUser,OutUser,[att(population,[])]),
    customer(InCustomer,OutCustomer,[att(population,[])]),
    account(InAccount,OutAccount,[att(population,[])]),
    generic:new(merge,{ExternActions,OutUser,OutCustomer,OutAccount},
        Actions),
    router(Actions,InUser,InCustomer,InAccount).
```

`take_active_actions` is a predicate that takes the external actions during the session of animation. There is always a class called `user` and an object of this class whose attribute `userid` has the value `root`.

Obligations In OASIS the object communication is activated by obligations. In this work we will focus on the fundamental communication mechanism, represented by triggers in the OASIS syntax. Triggers represent asynchronous communication. Hence, triggers are implemented executing $Out = [Action|NOut]$ without waiting for an answer. Triggers are generated due to a change of state. Thus, in the body of each clause that represents a change of state related with a condition in the trigger, a goal that evaluates the condition and could shoot the trigger will be added. This sentence will appear in the body of a selected clause in the reduction of a client object. *Out* is the identifier of the shared variable that constitutes the output channel of actions for the client object. This is a term containing the client reference. *NOut* is the new variable for the output channel of the client object.

Actions must first arrive to the router that sends the action to the corresponding input channel of class. If the action server is a class instance, the associated class searches in its *Population* attribute the input channel of the server object.

Example 6. A goal representing an object `account` reaching a new state:

```
o_account(In,Out,[att(number,101),att(name,john),att(balance,80),
    att(times,5),att(pin,1234),att(rank,0),att(good_balance,false)])
```

The possibility of trigger activation in the new state (that will be reached) was detected during the previous change of state. The clause doing this work is:

```
verify_o_account_triggers([t1|RestTriggers],State,Triggers) :-
    utility:get_attributes(State,[att(times,Times),
                att(good_balance,Good_balance),att(rank,Rank)]),
    LExp1:= Times,    RExp1:= 5,
    LExp2=Good_balance,    RExp2=false,
    LExp3:= Rank,    RExp3:= 0,
    utility:test_condition([[c(ge,LExp1,RExp1),
        c(eq,LExp2,RExp2),c(eq,LExp3,RExp3)]],Answer),
    utility:get_attributes(State,[att(number,Number)]),
    utility:put_trigger(Answer,action(this,self,e(pay_commission,[])),
        Triggers,NTriggers),
    verify_o_account_triggers(RestTriggers,State,NTriggers).
```

`put_trigger` is a predicate that puts the action in the trigger list if `Answer` is instantiated to *true*. In this case because `test_condition` returns *true* in `Answer`, we are sure that the object *Mbox* contains the action:

```
action(this,self,e(pay_commission,[]))
```

This is an action that must be triggered because the client term is `this`. Thus it is put in the output channel using:

```
Out=[action(This,self,e(pay_commission,[]))|NOut]
```

Now `This` is the client specification referring to its identification.

4 A graphical animation environment

In the OASIS context, the system behavior is determined by the behavior of its objects. An object behavior can be observed by analyzing the actions occurred and the states reached by the object. In this sense, the animation of an OASIS specification allows examining actions and states of the objects. Following the guides mentioned before, a translator from OASIS to KL1 has been implemented. This translator takes as input a system specification from an OASIS repository and produces a KL1 program that is compiled in order to obtain the prototype. The translator and the prototype are programs running in a Unix workstation. The interface has been implemented in Tcl/Tk [12] using the Tcl plug-in for Netscape and establishing a socket connection to the Unix workstation.

Fig.3 shows the interface for a preliminary version for an environment of animation. The idea is to make easier the use of the prototype. The object society is drawn in the upper left corner, on the right the traces of actions of an object (or object group) are listed. The list of traces can be filtered according to the kind of actions defined ($OExec_i$, $\overline{O}Exec_i$, $Conf_i$, $Exec_i$ and $Rejected_i$). In the state area the state of the object is presented (only when one object is selected). Buttons *play*, *pause*, *stop*, *forward* and *review* are provided in order to control the session of animation. When the animation is paused it is possible to explore the traces of actions and states at previous instants. Eventually, the two entry widgets allow building an external action sent by the analyst in representation of one object in the system.

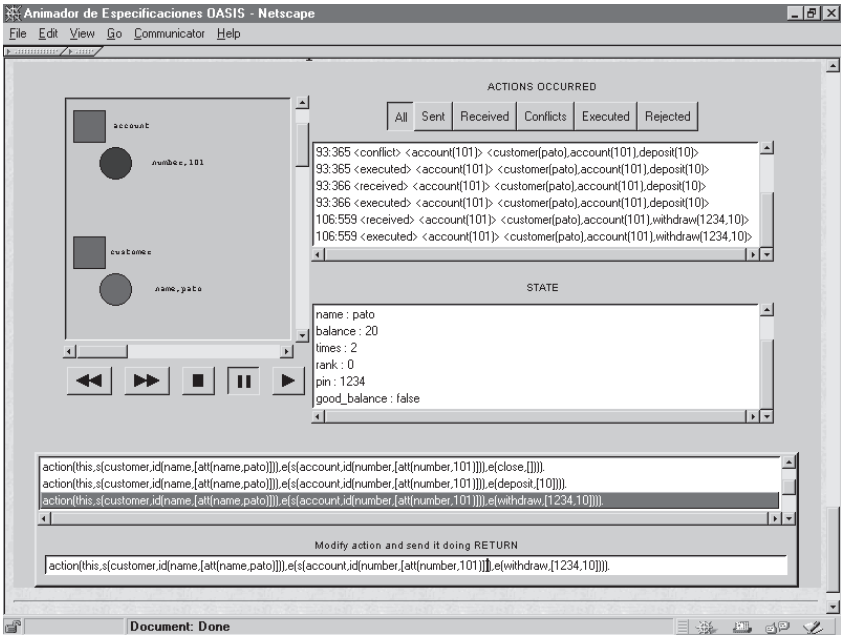


Fig. 3. An animation session

5 Conclusions

We have shown how the main features of OASIS can be naturally and directly represented in a concurrent logic program. Using the execution model of OASIS as a guide we can obtain a useful animation of the OASIS specification. Our animation is only applied to purposes of requirements validation and do not claim to be the final software product. The fidelity of the obtained concurrent logic program in relation to the OASIS system specification is a matter that is still being studied. In this case the verification and demonstration tasks would be supported by three important factors: firstly at the conceptual level the model is described in a formal language, secondly the abstract execution model is inspired by the semantics of that formal language and eventually there are some proposals dealing with formalization of concurrent programming languages. These factors do not determine the required justification but give a way of formalization on which we are working.

We have built a translator program to obtain a concurrent logic program automatically from an OASIS specification using the established correspondences. This work is being integrated into a CASE tool for system modeling supporting the OASIS model.

References

1. L. Åqvist. Deontic logic. In D.M. Gabbay and F.Guenthner, editors, Handbook of Philosophical Logic II, pages 605-714. Reidel, 1984.
2. T. Chikayama. KLIC User's Manual. Institute for New Generation Computer Technology, Tokyo JAPAN, 1995.
3. T. Conlon. Programming in PARLOG. Addisson-Wesley, 1989.
4. A. Davison. Polka: A Parlog object-oriented language, Ph.D. thesis, Department of Computer Science, Imperial College London, 1989.
5. D. Harel. Dynamic Logic. In Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pages 497-694. Reidel 1984.
6. P. Heymans. The Albert II Specification Animator. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm.
7. R. Herzig and M. Gogolla. An animator for the object specification language TROLL light. In Proc. Colloq. on Object-Orientation in Databases and Software Engineering, Montreal 1994.
8. P. Letelier, P. Sánchez and I. Ramos. Animation of system specifications using concurrent logic programming. Symposium on Logical Approaches to Agent Modeling and Design, ESSLLI'97, Aix-en-Provence, France, 1997.
9. P. Letelier, I. Ramos, P. Sánchez and O. Pastor. OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto. SPUPV-98.4011, Servicio de Publicaciones Universidad Politécnica de Valencia, 1998.
10. P. Letelier, I. Ramos and P. Sánchez. Un modelo de ejecución para especificaciones OASIS 3.0 Infórme Técnico DSIC-II/36/98, Universidad Politécnica de Valencia, 1998.
11. J.-J.Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.
12. J. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley, 1994.
13. O. Pastor and I. Ramos. OASIS version 2 (2.2) : A Class-Definition language to model information systems using an object-oriented approach, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.
14. E. Pimentel. L2||O2: Un lenguaje lógico concurrente orientado a objetos, Tesis Doctoral, Facultad de Informática, Universidad de Málaga, 1993.
15. C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyté, A. Sutcliffe, N.A.M. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois and P. Heymans. A Proposal for a Scenario Classification Framework, Technical Report CREWS 96-01, http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm.
16. P. Sánchez, P. Letelier and I. Ramos. Constructs for Prototyping Information Systems using Object Petri Nets, Proc. of IEEE International Conference on System Man and Cybernetics, pages 4260-4265, Orlando, USA, 1997.
17. E. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog, en New Generation Computing, vol.1, pages 25-48, 1983.
18. J. Siddiqi, I.C. Morrey, C.R. Roast and M.B. Ozcan. Towards quality requirements via animated formal specifications. Annals of Software Engineering, n.3, 1997.
19. K. Yoshida and T. Chikayama. A'UM: A string based concurrent object-oriented language, In Proc. of the Int.Conf. on FGCS, ICOT, pages 638-649, 1988.
20. R.J. Wieringa and J.-J.Ch. Meyer. Actors, Actions and Initiative in Normative System Specification. Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

From CASE to CARE (Computer-Aided Requirements Engineering)*

Oscar Pastor, José H. Canós, Isidro Ramos

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Camí de Vera, S/N, E-46071 València (Spain)
Tel: +34 96 3877350; Fax: + 34 96 3877357
e-mail: { opastor | jhcanos | iramos}@dsic.upv.es

Abstract

Software Engineering is still lacking methods which are capable of properly capturing the relevant System properties in the problem space and at the same time generating a correct representation (a final software product) in the solution space. This product must be functionally equivalent to the system specification and obtained in an automated way. The design of methods of this kind and the development of associated support tools is one of the current challenges in the field.

This leads us to real CARE Computer-Aided Requirements Engineering environments, where the replacement of the S (of CASE) by the R is very meaningful: we want to focus on the Requirements Engineering process to capture properly the relevant system properties. Our CARE advanced tools will convert these "patterns" of behavior into the most convenient representation in a given software development environment, by properly "compiling" the system specification following a set of mappings between conceptual patterns and software representations.

To do it, a formal basis and a model with the necessary expressiveness are needed in order to do this. The contribution of this paper is the object-oriented OASIS model as the basic model for such a CARE environment, together with a method for software production –the OO-Method- as an operational implementation of these ideas.

1. Introduction

Conventional CASE tools have traditionally dealt with the software production process by using different abstractions and notations for "*what"* the system is (problem space) and *"how"* it is to be represented in a particular software development environment (solution space). This fuzzy strategy has always originated doubts about the real utility of these CASE environments which are intended to ease the software production process. It has often been experienced that the use of CASE tools, rather than easing the solution of a problem, creates a new problem besides the original one: how to properly use the different notations provided by the tool in a set of diagrams, which often have overlapping semantics.

---

* This work has been supported by the CICYT under MENHIR/ESPILL TIC 97-0593-C05-01 Project and by DGEUI under IRGAS GV97-TI-05-34 Project.

Pastor,O., Canós,J.H., Ramos,I. From CASE to CARE (Computer-Aided Requirements Engineering). Proc. of the International Conference on Conceptual Modeling (ER'99). 1999.

Structured Analysis techniques use two different abstractions to deal with statics and dynamics. On one hand, an Entity-Relationship-like data model is provided to represent the static system architecture including data components, their definition and their relationships. On the other hand, a process model is necessary to properly specify the system behavior. Object Oriented (OO) technology has emerged as an important advance by introducing the notion of object as the only abstraction, (including both statics and dynamics) to represent reality. However, the corresponding OO CASE tools have not been able to clarify the software production process.

Nowadays, we find a set of what could be called *conventional* OO methodologies, coming from practical use in industrial software production environments, which do not have a formal basis and which often use classical structured concepts together with the introduction of OO features ([1,2,3,4,5)]. Recent proposals are trying to create a unified framework for dealing with all the existing methods (e.g. UML [6]). They have the implicit danger of providing users with an excessive set of methods and notations whose semantics overlap.

On the other hand, in their quest for a solution, other authors have proposed the use of OO *formal specification languages* (Oblog [7,8], Troll [9,10] , Albert [11], OASIS [12, 13]). These languages have a solid mathematical background and deducible formal properties such as soundness and completeness.

Neither of these approaches offers an acceptable individual solution. Formal methods are hard to use and do not reach industry. Conventional methods are so ambiguous that in practice they often generate the same "hostile" reaction for different reasons. However, if both approaches have some good properties when dealing with the problem of elaborating quality software, a simple question arises: why not blend their good properties leaving the useless ones out?

Software Engineering is still lacking a method which is capable of:

1. properly capturing the relevant system properties in the problem space
2. generating a solution in the form of an end software product in the solution space, which is functionally equivalent to the system specification and is obtained in an automated way.

This leads us to a Computer-Aided Requirements Engineering (CARE) environment, where the replacement of the S by the R is very meaningful: we want to focus on the Requirements Engineering process to capture properly the relevant system properties. Our advanced CARE tools will convert these patterns of behavior into the most convenient representation in a given software development environment, by properly "compiling" the system specification.

This approach creates new avenues for dealing with traditional Software Engineering problems that have not yet found a clear solution in the conventional CASE world, such as reuse, schema evolution or automated code generation. If the specification expressiveness is based on a notation which is close to the problem space, the reusability and evolution of such a specification will be easier. If we determine the software components that correspond to a finite set which includes the potentially valid behavioral patterns, code generators could be implemented by programming the mappings that convert the quoted behavioral patterns in their corresponding concrete software representations.

Our research work has been directed towards designing and implementing an OO software production environment whose objective is to combine the pragmatic

aspects attached to the so-called conventional methods with the good formal properties of the OO specification languages. To do it, a graphic, OO conceptual modeling environment that collects the relevant system properties for building a formal, textual OO specification in an automated way, is introduced.

In contrast to other works in this area ([14,15]), we focus on the definition of a concise execution model and on the mapping between the specification language and the execution model notions. This makes possible to build an operational implementation of a software production environment allowing for real automated prototyping, by generating a complete system prototype (including statics and dynamics) in the target software development environment. A CARE workbench which supports this working environment in a unified way is currently available for prototyping purposes. In this paper, we will show that with the formal OASIS model and the corresponding operational implementation of the OO-Method, we support previous works based on the idea that we can significantly reduce the complexity of advanced-application specification and implementation by using a model-equivalent language (a language with a one-to-one correspondence to an underlying, executable model [16]). This model-equivalence is obtained in our case by defining a precise set of conceptual modeling patterns, and the mapping that converts them into concrete software representations.

To accomplish these aims, the structure of this paper is the following. After the introduction, we describe the most relevant features of the OASIS object oriented model, introducing the basic ideas on OO conceptual modeling that are the basis of the work. Next, we introduce the main contributions of the OASIS model to give precise solutions to traditional Software Engineering problems such as schema evolution or methodological guidance from problem space notions to solution space representations. An operational implementation based on the OO-Method – the methodological approach attached to OASIS – is introduced in section 4. We will focus on how to represent a conceptual model in a particular software development environment according to an abstract execution model, which will fix the operational steps to follow when we want to give a concrete system implementation. A software prototype which is functionally equivalent to a system specification can be obtained in the context of the methodology. We describe the code generation strategy used. Conclusions and the list of references end the paper.

2. The OASIS Model

Prior to the presentation of the OASIS model, let us briefly introduce Dynamic Logic, which will be used in this paper to describe it.

2.1 Formal Framework: Dynamic Logic

Dynamic Logic (DL) is a modal logic proposed by David Harel [17] in the late 70's as a formalism for reasoning about those programs that check and modify a given environment. The main goal was to describe in a formal way the effect of programs execution as a first step for further formal reasoning about them. Although Harel's work on DL is very extensive, we present below the dynamic formulae that are relevant to our purposes.

A Dynamic formula is expressed as $\phi\ [p]\ \psi$ where ϕ and ψ are first order formulae, p represents a program (in a general sense) and $[\_]$ is the necessity operator.

The intuitive meaning of such a formula can be stated as follows: "*if ϕ holds , then after execution of the program p, ψ must hold*".

Dynamic formulae (as opposed to first order formulae) have a special singularity. The latter are interpreted in a single interpretative structure, whereas the former need several structures (in particular, 1st-order logic is a subset of DL). This important difference can be understood by taking into account that dynamic formulae are used to describe an evolutionary process, that is, a certain change of state. Thus, a formal dynamic theory is interpreted by using a Kripke structure $K=(\Omega, \omega_0, \rho)$ where:

- Ω is a set of possible worlds, each one of which is defined as a first order interpretative structure,
- ω_0 is the initial world,
- ρ is an accesibility relation among worlds (see Figure 1).

The effect of the execution of any program can be described by using this language. For our purposes, we use DL to describe the evolution of objects. In other words, we use it to describe the state changes in objects that take place as a reaction to certain external stimuli (namely events[1]) that will play the role of programs in classical DL. We present the OO model in which our work is based, first from an intuitive point of view and later through more precise concepts.

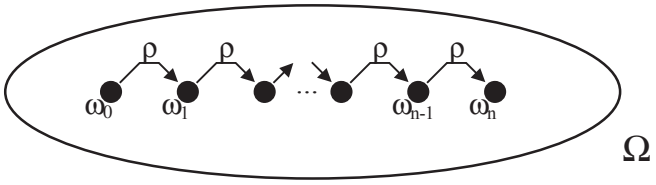


Figure 1 Simple Kripke structure

2.2 Basic Concepts

In the OASIS model, Information Systems (IS) are considered as societies of communicating objects that evolve through state changes following certain behavior rules. An *object* is observed in order to know some of its structural properties, which are represented in the model by the notion of *attribute*. Each object is referred to by means of its *identifier*, which distinguishes it from any other object in the IS. The *state* of an object is given at any moment by the set of values of its attributes (represented as a 1st-order presentation). Thus, a change in any of the attribute values will lead to a change in the object's state. The value of an attribute may change due to the occurrence of an *event* in the life of the object; the way the value of an attribute changes after an event occurrence is described using valuation rules. In addition, a set of integrity constraints restrict the possible states and state sequences or *traces* that an object can reach during its life. Object lives are maximal traces.

[1] similar notions in other models are messages, events, transactions, etc.

A formal definition of the OASIS model can be found in [13]. However, the following concepts are relevant to the aim of this paper.

Object Types. An object type is defined as a tuple T=(A,X,F,P), where:

- A is a set of *attributes*, which may be constant, variable and derived; for a given object, constant attributes take their values when the object is created and do not change during its life, whereas the values of variable attributes change with event occurrences; finally, derived attributes take their values from other attributes according to some derivation rules.
- $X = X_s \cup X_a$ is a set of *events*; X_s are the *services* provided by the instances of the type, and X_a are the *actions* that might be requested to other objects in the system, even to themselves. For every action there must be a service with the same name in some type of the system.
- $F = F_v \cup F_d \cup F_p \cup F_t$ is a set of *formulae* which capture different aspects of the OASIS behavioral model; F_v are the *valuation* formulae that state how attribute values are changed by events; F_d is a set of *derivation* rules associated to derived attributes; F_p are the event *preconditions*, that define the valid states of an object for an event occurrence to be relevant (i.e., to actually change the object's state); F_t are the *triggering rules*, that define part of the reactive behavior of the OASIS objects by stating events that must be activated when a condition is satisfied; finally, F_c are *integrity constraints* which are expressed as temporal formulae that must hold over any sequence of states of an object.
- P is a term built over the Basic Process Algebra (BPA) defined in [18]; every actual life of an object must be a ground process term of P.

Objects and Classes. An *object template* is a triple (T, I, P), with T being a type, I an identification function (a set of pairs (identifier, instance)) and P the public part of T that for a type states which attributes and/or events[2] are accessible to the objects in the system. So, an object o in a state is a triple (Σ, i, S), Σ being a template, $i = I(o)$ the identifier of o obtained by applying the identification function $I \in \Sigma$ to it, and S the state of the object. S is described using first-order formulas (precisely, a set of ground atoms, one per attribute in the type of the object).

A *class* adds to the object template the aspects of *factory* (i.e. creation and deletion of objects) and *extent* that also include some other object models [19]. Classes are represented as objects whose template includes the properties listed below:

- object factory properties are those that allow a class object to create new objects called instances of the class. Of these, we describe the following:
  - a collection of attributes whose current values define the template to be used to create instances; examples are *constant-attribute-list*, *derived-attribute-list*, *private-event-list*, and *constraint-list* ([20]).
  - two special events, *new* and *destroy*, used to create (using the template defined by the factory attributes) and destroy instances, respectively

[2] in the sequel, we will refer to both attributes and events as properties

– an attribute *next-oid* whose value is used by *new* to assign an identifier to the created object, and which is updated after its use

- collection properties: an attribute *extent* which at any moment holds the set of identifiers of the alive instances of the class, i.e. objects created by the class object that have not yet been destroyed.

Hence, the schema meta-information is included in the model as part of the state of the OASIS classes seen as objects. This is a very important property that will be used later in this paper to include schema evolution as a built-in property of the OASIS model (see section 3).

Class Hierarchies. The OASIS classes can be elementary or complex; complex classes represent either generalization/specialization or aggregation relationships. A description of these relationships is outside the scope of this paper, due to the fact that the abstract execution model we introduce below is orthogonal to this dimension. We refer the reader to [13] for a detailed description of the OASIS class relationships.

3. The OASIS Approach to the Classical Software Engineering Problems.

The OASIS model allows the specification of the structure and behavior of an Information System in great detail. The structural aspects are captured as in other OO models. However from the behavioral point of view the OASIS expressiveness is greater. The OASIS model is more complex in the sense that not only the signature of the events is given in each object type, but also the way they are allowed to happen. That is, the patterns of allowed behaviors of the objects are explicitly captured in the model following two approaches:

– *State dependent specification*: event preconditions and triggering relationships capture the deontic concepts of permission and obligation, respectively.
– *State independent specification*: the allowed patterns of behavior are given by using terms of the Basic Process Algebra.

This richer expressiveness allows us to capture more semantics of the problem domain; the price for this is to get a more complex model to understand and use, although the benefits we obtain compensate for this problem, specially if we combine the process specification capabilities with the metalevel aspects as they are introduced in the OASIS model (see below).

To be homogeneous and consistent, the OASIS model views all class objects as instances of another class, namely the OASIS *Metaclass*. The OASIS *Metaclass* is the only predefined object in the model and is an instance of itself (closing in this way the circularity induced by the model). All other objects (either class objects or plain instances) can be created dynamically through the event *new* associated to any class object. This has implications in the quality of the software artifacts obtained by using the OASIS Model. We illustrate them at two levels, namely the *object* or Information System (IS) level and the *metalevel*.

3.1 The Object Level

At an object level, the conceptual models we get are semantically richer and more complete because the behavior of an IS is included in the conceptual model. As OASIS has strong logical and categorical foundations, the operational semantics of DL makes OASIS specifications executable. Therefore, it is no longer necessary to use a host language (like C++, Smalltalk or others) that use a different epistemological background, a different modeling metaphor and different software tools (Compilers or Interpreters) to capture this behavior.

From a Software Engineering point of view, we obtain homogeneity in the sense that a unique language which technically implements the model expressiveness acts as the Object Definition Language (ODL), as the Object Query Language (OQL) and finally, as the Object Manipulation Language (OML) [20]. The ODL is a class definition language. The OQL is the technical implementation of the declarative language of the state logic used [21] and the OML is the language supporting the process expressiveness of the model. An OASIS object (which is a process) can be seen as the program (in the classical sense) that the quoted object represents. This is usual in some OO+Logic Programming Approaches (e.g. [22]). In this way, a program in OASIS is an active object implementing the obligation of fulfilling its process part.

The homogeneity in the conceptual models produced by using OASIS allows for the definition of a unique translation schema for any target environment: the Execution Model (EM) (see 4.1 below). Its introduction allows for a true implementation of the Automatic Programming Paradigm, fulfilling the gap between the problem space and the solution space descriptions.

The software tools implementing these ideas are clearly an advancement over the classical CASE approach. A companion methodology, called OO-METHOD [23], is added to OASIS and is introduced in section 4. As will be shown, advanced visual programming environments are used in OO-METHOD to produce the Conceptual Models that will automatically be mapped to implementations in OO programming languages.

3.2 The Metalevel

As in many other approaches, the classes composing a given Conceptual Model are instances of a metaclass (the OASIS Metaclass or OM for short). What is new in the OASIS Model is the dual view of these metaobjects[3]:

- As classes, they have a name, a template (T1) which includes factory services and the extent.
- As (meta)objects, they have an oid and a template T2 whose attribute set defines the properties of their instances, and a dynamic behavior given by a signature of events and a process; these events change T1 in accordance with the formulae in T2. Note that T2 defines a schema evolution mechanism.

In general, T1 and T2 are different for each class. The only exception is OM, for which T1=T2. This way we cut the infinite circularity of other metaclass-based

[3] In the sequel, we will use the terms class and metaobject interchangeably to refer to the instances of the metaclass OM

approaches. It is interesting to note that the process part of OM captures the method to be followed by a system designer to build an IS. Another interesting feature comes from the fact that the extent of OM is the repository of applications.

The coding of a model in itself is a form of reflectivity that provides us with a simple and elegant way to cope with the classical Software Engineering problems of software evolution, software reuse, software versioning, software process formalization, true methodological aids and guidance, etc. Although the unique model used in our case is OASIS, other models could be used simultaneously if several metaclass were defined and included in a software production environment (a modern CARE tool). These models can be independent or related by the relationships provided by the model (e.g. is-a, part-of, etc.).

3.3 The Problems and the Solutions

Several Software Engineering problems are presented and the solutions within the OASIS Model are outlined in the following points. A set of these solutions has already been implemented. Others have been clearly stated and specified within the model but not implemented and others have only been enunciated. The aim of this section is to illustrate the good properties of the OASIS model within the Software Engineering field and how future CARE environments will be produced in the near future.

Software Evolution. An application (a conceptual schema in OASIS) is defined as the state of a given class that can be changed at any moment by the user through the services offered by the metaobject. The changes resulting from the occurrence of the event in the metaobject state are described in a declarative way in the OM specification. On the one hand, some changes make template T2 evolve, and on the other hand, other changes must affect to extent in order to remain compliant with T2.

Two implementations of the evolution services have been performed. One of them [25] uses Transaction Frame Logic [24] as the underlying logic language and the other one has been implemented on top of a Clausal Dynamic Logic programming environment.

Software Versioning. Software evolution is motivated by the dynamic of business rules in the problem space. If we try to keep track of the history of the different Conceptual Models (each of which is valid within a period of time) we need a time stamping of the different states that the class defining the conceptual schema reaches during its life. For this purpose, a time model based on Kowalski's Event Calculus is used in OASIS so that each state is stamped by the periods of time in which the conceptual model it represents is valid. Then, one may retrieve past versions of schemas by means of temporal queries [20].

Obviously, the same time model can be used at the object level to implement classical temporal data models and the historic files. *Method Aids and Guidance.* Current industrial CASE tools and Companion Methodologies provide a set of visual languages to describe the Conceptual Models and a sometimes tedious set of heuristics that the developer has to follow to be compliant with the method rules. This is one of the reasons for the traditional CASE underachievement. In practice, the developer develops software which is only partially compliant with the companion method.

It is clear that the only way in which a software development method can be successfully applied is by implementing it in the computer by means of a set of aids

and guidance tools. Within OASIS, the process part of the metaclass OM clearly and unambiguously fixes the way in which software must be built. The implementation of these ideas within a CARE Environment is the goal of the MENHIR Project that unites five spanish university teams and four software companies for 3 years (1998-2000).

A partial implementation of these ideas is the OO-METHOD CASE tool whose basic concepts are presented in the following point. We will emphasize how the set of services provided by the metaobject OASIS fixes a particular implementation where all these services are provided in a conventional CARE environment and where the Conceptual Model is converted into a final software product as a value-added feature, by implementing an abstract execution model that guides the process of implementing OO-Method Conceptual Models.

4. The Methodology

The *OO-Method* is an Object-Oriented Software Production Methodology whose phases are shown in Figure 2. Basically, we can distinguish two components: the conceptual model and the execution model. When facing the conceptual modeling step of a given Information System, we have to determine the components of the object society without any implementation concern. The problem at this level is to obtain a precise system definition (the conceptual model). Once we have an appropriate system description, a well-defined execution model will fix the characteristics of the final software product in terms of user interface, access control, service activation, etc.; in short, all the implementation-dependent properties.

In this context, we start with an Analysis step where three different models are generated: the Object Model, the Dynamic Model and the Functional Model. They describe the Object Society from three complementary points of view within a well-defined OO framework. For these models, we have preserved the names used in many other well-known and widely-used OO methodologies, even if the similarities are purely syntactic as can be seen throughout this paper.

The OO-Method implements the set of services provided by the metaclass OM in a conventional way to build a conceptual model. For every service there is an option implemented in the most convenient way in the tool: for instance, for the *create attribute* service, a friendly and convenient implementation is provided in the context of a class specification.

From these analysis models, a corresponding formal and OO OASIS specification (the OO-Method design tool) can be obtained in an automated way. This is done through an automatic translation process. The resultant OASIS specification acts as a complete system repository, where all the relevant properties of the component classes are included. This is an important contribution of the OO-Method approach: the analysis models are intended to capture only the relevant system information, and what is considered relevant is fixed by the semantics of the OASIS specification language.

According to the execution model, a prototype which is functionally equivalent to the specification is built in an automated way. This may be done in both declarative (Prolog-based) and imperative environments (specially those visual OO programming environments that are widely used nowadays). The code generation strategy is independent of any concrete target development environment, even if

currently our selected environment for automated code generation are Visual C++, Delphi, Java, Visual Basic and PowerBuilder.

The basic characteristics of the three models (object, dynamic and functional) that constitute the conceptual model are explained in [26]. Now, we introduce the execution model features to explain how the conversion strategy from the former to the latter is done within OO-Method.

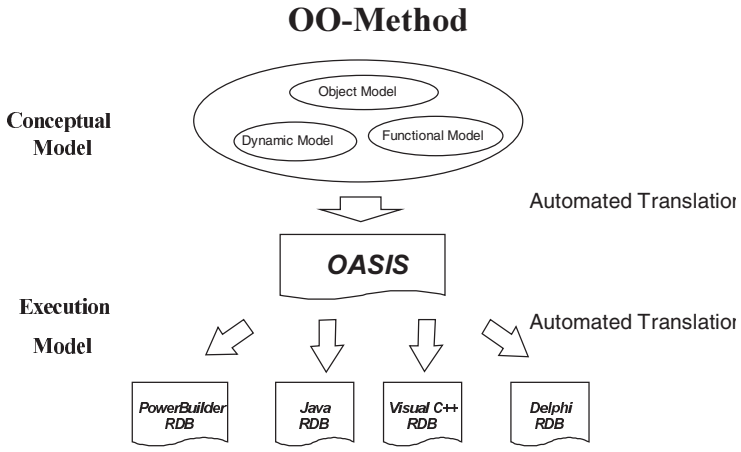


Figure 2. Phases of OO-Method.

4.1 Execution Model

Once all the relevant system information in the specification that we have called conceptual model is collected , the execution model has to accurately state the implementation-dependent features associated to the selected machine representation of the object society. More precisely, we have to explain the pattern to be used to implement object properties in any target software development environment.

Our idea at this point is to give an abstract view of an execution model that will set the programming pattern to be followed when dealing with the problem of implementing the conceptual model. This execution model has three main steps:

1. *access control:* first, the user logging in the system has to be identified as a member of the corresponding object society.
2. *object system view:* once the user is connected, he must have a clear representation of which classes he can access. In other words, his object society view must be stated, precisely defining the set of object attributes and services he will be allowed to see or activate, respectively.
3. *service activation:* finally, after being connected and having a clear object system view, the user will be able to activate any available service in the user's world view. Among these services, we will have event or transaction activation served by other objects, or system observations (object queries).

Any service execution is characterized by the following sequence of actions:

1. *object identification*: as a first step, the object acting as server has to be identified. This object existence is an implicit condition for executing any service, except if we are dealing with a *new*[4] event. At that moment, their values (those that characterize its current state) are retrieved.
2. *introduction of event arguments*: the rest of the arguments of the event being activated must be introduced.
3. *state transition correctness*: we have to verify in the State Transition Diagram associated to every class that a valid state transition exists for the selected service in the current object state.
4. *precondition satisfaction*: the precondition associated to the service that is going to be executed must hold. If not, an exception will arise, informing that the service cannot be activated because its precondition has been violated.
5. *valuation fulfillment*: once the precondition has been verified, the induced event modifications are effective in the current object state.
6. *integrity constraint checking in the new state*: to assure that the service activation leads the object to a valid state, we must check that the (static and dynamic) integrity constraints hold in this final resulting state.
7. *trigger relationships test*: after a valid change of state, and as a final action, the set of rules condition-action that represents the internal system activity have to be verified. If any of them holds, the corresponding service activation will be triggered. It is the analyst's responsibility to assure the termination and confluence of such triggers.

The previous steps guide the implementation of any program to guarantee the functional equivalence among the object system description collected in the conceptual model and its reification in a software programming environment according to the execution model.

Next, we are going to present the code generation strategy used in the implementation of the previous execution model in a graphic environment, which opens up the possibility of creating a CARE tool that, starting from a set of graphical OO models obtained during the conceptual modeling step (according to OO-Method) can generate a functional software prototype at any time.

4.2 Code Generation Strategy

Once an abstract execution model has been defined, we will have different concrete implementations of this execution model for different software development environments. It is important to remark that the representation of the conceptual model in the selected execution model is done according to the principles introduced above, thus generating a prototype in an automated way by adapting the code generation strategy that we present to the particularities of the corresponding target development environment.

---

[4] Formally, a new event is a service of a metaobject representing the class, which acts as object factory for creating individual class instances (see section 3)

Using a conventional graphical system, the code generation process creates four types of windows as can be seen in Figure 3:

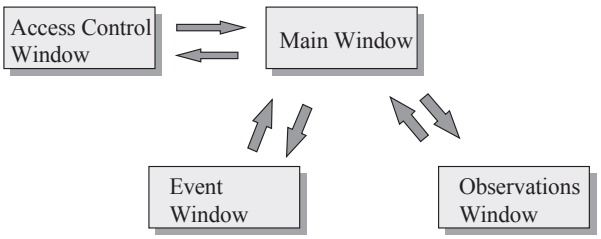


Figure 3. Overview of the generated code structure.

- *Access Control Window*: this is the log-in window (Figure 4), where the corresponding active user has to be identified. This is done by introducing its object identifier, class name and password. The identification is verified on the database to ensure that the object exists. Once the object is incorporated to the system, it will see the available system class services through menu items of the main menu.

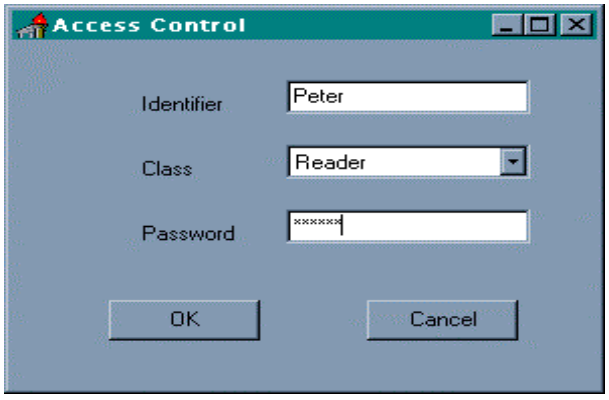


Figure 4. Access Control Window

- *Main Window*: it characterizes the system view that the connected object has (Figure 5). All the services of the classes are requested through it. For instance, in a simple library system, we will have as main options in the menu bar *loans, books, readers* and so on, representing the system classes we can access. Generally speaking, it has the following options:
  - the typical File item option of conventional graphical applications.

– for every class, a pull-down menu including an item for observations (queries), a section with its subclasses (if any) and a last section with the available class services. For instance, for the book class, we can activate the observations option, or the *new-book, loan, return...* events
– an interactions item, which allows for the activation of global interactions.

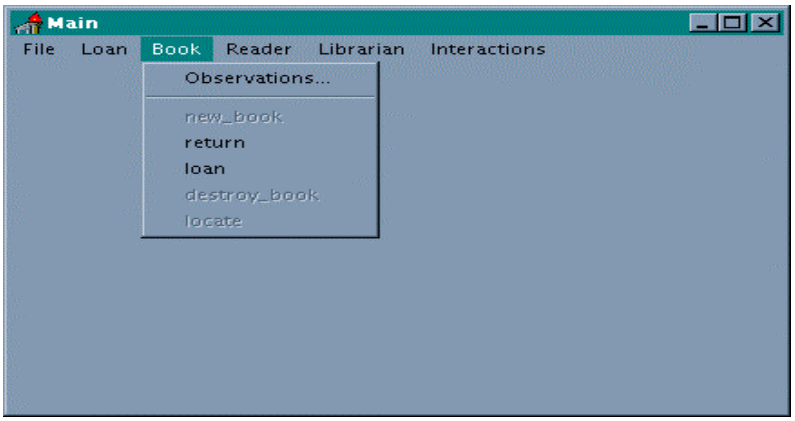


Figure 5 Main Window.

- *Event window*, where the corresponding arguments are introduced and the induced actions are executed through the OK control button. It has an script associated to it including the event pre/post conditions, specified in the OO-Method Conceptual Model and which are properly represented in the target software development environment.
- *Observations window*: this screen is intended to be a Query By Example pattern where the user can see the results of any query made about the current object state.

This is a precise strategy for executing OO-Method conceptual models, that will guide any concrete implementation where the particular aspects of the corresponding target development environment should be taken into account.

5. Conclusions

The provision of automated software engineering tools to support the whole software development process is very important in the promotion and adoption of the process and its associated methods. The development of these tools is a significant software engineering task. In this paper, such an environment is introduced using what we call CARE technology, based on a formal object-oriented model (OASIS) and a method (OO-Method) for capturing system requirements and converting them into executable software representations in a target software development environment.

Traditional approaches to the problem of producing software of quality are characterized by uncertainty, complexity and excessive effort to finally produce doubtful results: deciding whether a conceptual pattern with appropriate expressiveness exists, choosing between several possible representations apparently meant for the same purpose, obtaining a correct software representation which is functionally equivalent to the conceptual model, etc. are common problems solved in OO-Method by providing a precise set of conceptual modeling patterns (that have a formal counterpart in the OASIS model), their graphical representation and their corresponding software representation. Research work is being developed to

(a) introduce interface patterns in the specification of the OO-Method Conceptual Models,
(b) to improve the quality of the final software product obtained in an automated way and
(c) to implement more accurately the ideas of the Oasis Metaclass presented in this paper to have more advanced CARE environments dealing properly with schema evolution and method guidance.

References

[1] Wirfs-Brock R., Wilkerson B., Wiener L. *Designing Object Oriented Software*. Englewood Cliffs, Nj. Prentice-Hall. 1990

[2] Rumbaugh J. et al. W. Object Oriented Modeling and Design. Englewood Cliffs, Nj. Prentice-Hall. 1991

[3] Jacobson I. et al. G. OO Software Engineering , a Use Case Driven Approach. Reading, Massachusetts. Addison –Wesley

[4] Booch,G. OO Analysis and Design with Applications. Addison-Wesley, 1994

[5] Coleman D., Arnold P., Bodoff S., Dollin S. Gilchrist H., Hayes F., Jeremes P. *Object-Oriented Development; The Fusion Method*. Prentice-Hall 1994.

[6] Booch G., Rumbaugh J., Jacobson I. *UML. v1*. Rational Software Co., 1997.

[7] Sernadas A., Sernadas C., Ehrich H.D. *OO Specification of Databases: An Algebraic Approach*. In P.M. Stocker, W. Kent eds., proc of VLDB87, pags: 107-116, Morgan Kauffmann, 1987

[8] ESDI S.A., Lisboa. OBLOG CASE V1.0- User's Guide

[9] Jungclaus R., Saake G., Sernadas C. *Formal specification of Object Systems*. Eds. S. Abramsky and T. Mibaum Proceeding of the TapSoft's 91, Brighton. Lncs. 494, Springer Verlag 1991, pgs: 60-82.

[10] Hartmann T., Saake G., Jungclaus R., Hartel P., Kusch J. *Revised Version of the Modeling Language Troll (Troll version 2.0)*. Technische Universitat Braunschweig, Informatik-Berichte, April 1994

[11] Dubois E., Petit M., Wu S. *ALBERT: A Formal Agent-Oriented Requirements Language for distributed Composite Systems*. In Proc. CAiSE'94 Workshop on Formal Methods for Information Systems Dynamics, pags: 25-39, University of Twente, Technical Report, 1994

[12] Pastor O., Hayes F., Bear S. *OASIS: An object-oriented specification language*. In P. Loucopoulos, editor, Proceedings of the CAiSE'92 conference, pp. 348-363, Berlin, Springer, LNCS 593 (1992).

[13] Pastor O., Ramos I. *Oasis 2.1.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, October 95 (3rd edition).

[14] Wieringa, R.J., Jungclaus, R., Hartel, P., Hartmann, T., Saake, G., OMTROLL Object Modeling in TROLL. Proceedings of the International Workshop on Information Systems - Correctness and Reusability (IS-CORE'93). Hannover, September 1993. Udo W. Lipeck, G.Koschorrek (eds.).

[15] Jackson R.B., Embley D.W., Woodfield S.N. *Automated Support for the Development of Formal Object-Oriented Requirements Specification*. In Proceedings of CAiSE-94 Conference. Utrecht, The Netherlands. Lecture Notes in Computer Science; v. 811; p.135-148. 1994.

[16] Liddle S.W., Embley D.W., Woodfield S.N. *Unifying Modeling and Programming Through an Active, Object-Oriented, Model-Equivalent Programming Language* . In Proceedings of 14th International Conference on Object-Oriented and Entity-Relationship Modeling (OOER'95). 13-15 Dec 1995. Gold Coast, Australia. Lecture Notes in Computer Science v. 1021; p. 55-64. 1995.

[17] Harel, D., *First-Order Dynamic Logic*, Springer-Verlag, 1979.

[18] Wieringa R.J. {\em A Formalization of Objects using Equational Dynamic Logic}, 2nd Conference on Deductive and OO Databases, pags 431-452, Springer 1991, Lecture Notes in Computer Science 566.

[19] Catell, R. G. G. (ed.), *The Object Database Standard: ODMG-93, Release 1.1*, Morgan Kaufmann Publishers, 1994.

[20] Canós,J.H.:Ramos, I.;Lozano,M.D. *KAOS: an Object-oriented software tool for the objects definition, updating, querying and programming in an object-oriented environment*. Proc of the IEEE 1998 Canadian Conference on Electrical and Computer Engineering, IEEE Press, 1998

[21] Ramos, I. *Logic and OO Databases: a Declarative Approach* Porceedings of the Dexa 90, Springer-Verlag, 1990

[22] McCabe, F.G., *Logic and Objects*, Prentice-Hall, 1992

[23] Pastor O. et al. *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods*. In Antoni Olivé and Joan Antoni Pastor editors, Proceedings of CAiSE97 conference, pp. 145-158, Berlin, Springer-Verlag, LNCS 1250. June 1997

[24] Kifer M., *Deductive and Object Data Languages: A Quest for Integration*, Proceedings of the DOOD-96 Conference, Springer-Verlag, 1996

[25] Carsí J.A.; *A DOOD System for Treating the Schema Evolution Problem*, EDBT'98 Demo Session, VI Intl.Conference on Extending Database Technology, Valencia, March 1998

[26] Pastor,O.;Insfran,E. ;Pelechano,V. ;García,J. *From Object Oriented Conceptual Modeling to Automated Programming in Java.* Conceptual Modeling – ER'98. Lecture Notes in Computer Science (1507), págs: 183-197, Springer-Verlag, 1998, ISBN: 3-540-65189-6 ; ISSN : 0302-9743, Singapore

Integration of the user model and human factors in an Object Oriented Software Production Environment *

María Dolores Lozano
Computer Science Department
University of Castilla-La Mancha
Albacete (Spain)
e-mail: mlozano@infeab.uclm.es

Isidro Ramos
Department of Information Systems and Computation
Polytechnic University
Valencia (Spain)
e-mail : iramos@dsic.upv.es

Keywords: Interaction Systems, User Interfaces, Object Oriented Paradigm

Abstract

Nowadays, most of the object-oriented techniques used in the development of systems does not take into account Human-Computer interaction issues. This is a very important aspect that must be tackled in order to correctly fulfil the user needs and requirements. The main purpose of this paper is to present an object oriented approach for the integration of a user model in an object oriented software production environment.

1. Introduction.

The main goal of this paper is to present a methodology for specifying user interfaces within an automatic software production environment**.** This environment is supported by an object oriented model, called OASIS (Open and Active Specification of Information Systems) [Pas95 y Pat98] and a methodology associated to the model called OO-Method [Pas97], both developed by one of the research groups of the department of information systems and computation from the Technical University of Valencia (Spain) covering the typical phases of Analysis, Design and Implementation within the principles of the Object Oriented Paradigm. Our aim in this work is to enrich this Formal and Object-Oriented Specification Language, OASIS, by including aspects of user interface specification. The user interface is an important entity placed between the user and the application that the user needs to interact with the system. Thus, the user interface must be designed to be capable of communicating with both the end user and the application.

So, in this paper we propose a user model which models the user-system interaction. Besides, we also show a specific example in which this methodology is applied in the phases of Analysis and Design, showing the different steps in which it is developed.

The model we propose includes the generation of graphical and object-oriented diagrams to represent user interface issues.

2. User Interface Specification.

* This work is partly supported by the MENHIR project from the Spanish government commission of Science and Technology with reference TIC-97-0593-C05-01.

Lozano, M., Ramos, I. Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. Workshop on Integrating Human Factors into Use Cases and OO Methods at ECOOP'99. 1999.

In order to include the specification of end-user needs in this development process we have followed a Task Analysis technique taking into account the ideas proposed by [Jaa95] and the principles of our OASIS model to develop a user model which fills the existing gap in the model as far as the integration of human factors in the model are concerned.

The location of the user interface specification process within the model is the following:

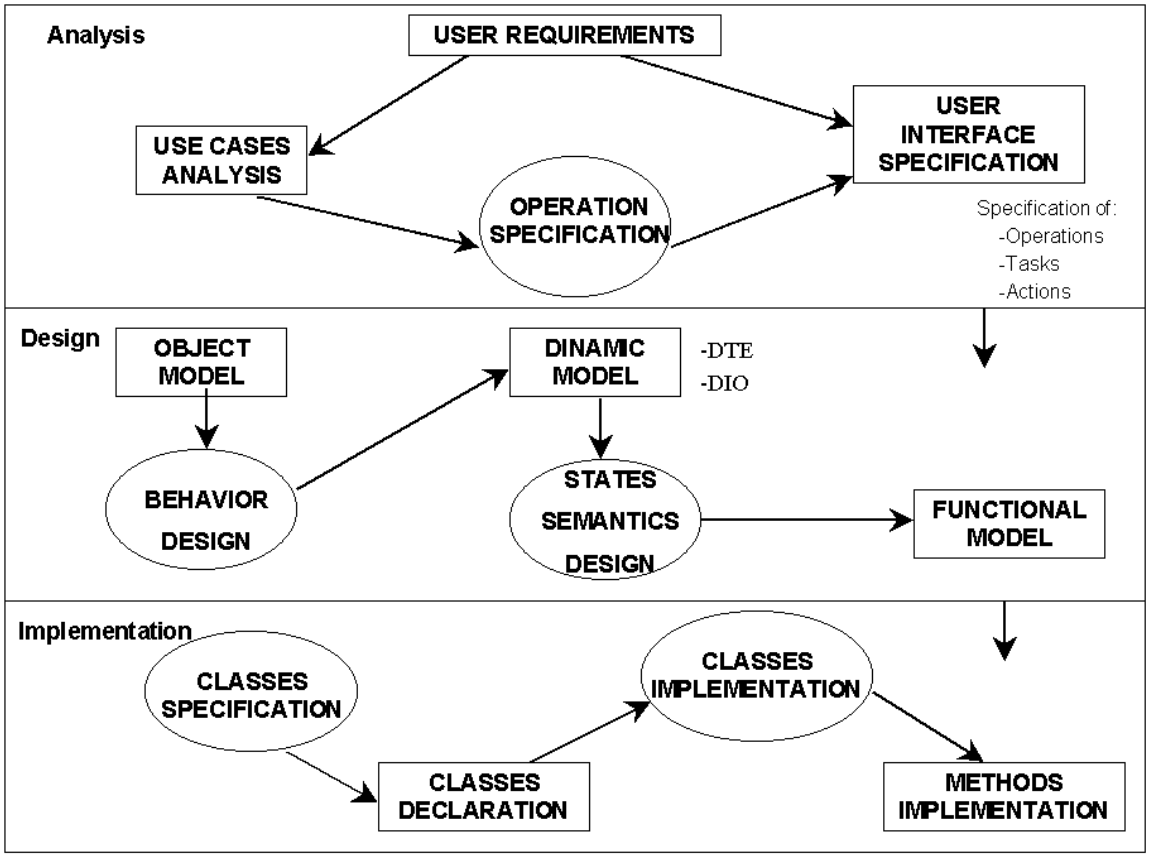


Fig. 1: Location of the object oriented specification process.

Next, we describe the model and in order to illustrate it, we will apply it to a simple case: a *Library*.

The process of developing the user interface for an Information System is tackled following the typical phases of Analysis, Design and Implementation.

Analysis Phase:

The user model is represented by means of what we have called the *user diagrams*. These ***user diagrams*** consists of three different types of diagrams:

* *The Dialog Structure Diagram*, which represents the windows and dialogs that the user needs to complete all the tasks he requires from the system, and the user selections to pass from one window to another.

In the *library* case this diagram would be the following:

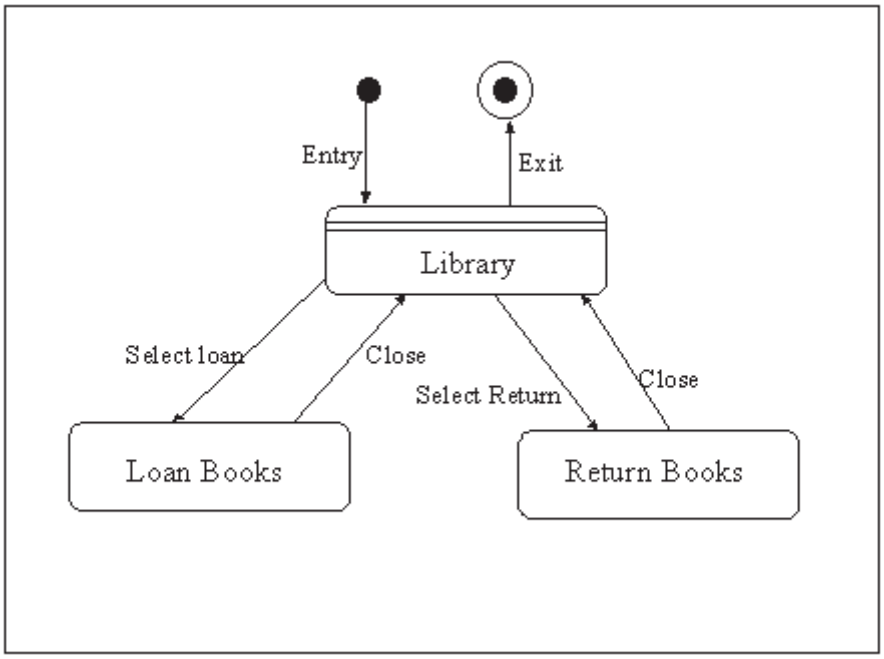


Fig. 2: Dialog Structure Diagram.

* *The Component Specification Diagrams,* which models the contents of every one of the windows and dialogs represented in the previous diagram. Every window is composed of components and tools. For instance, the component specification diagram for the dialog *Loan_Book* would be the following:

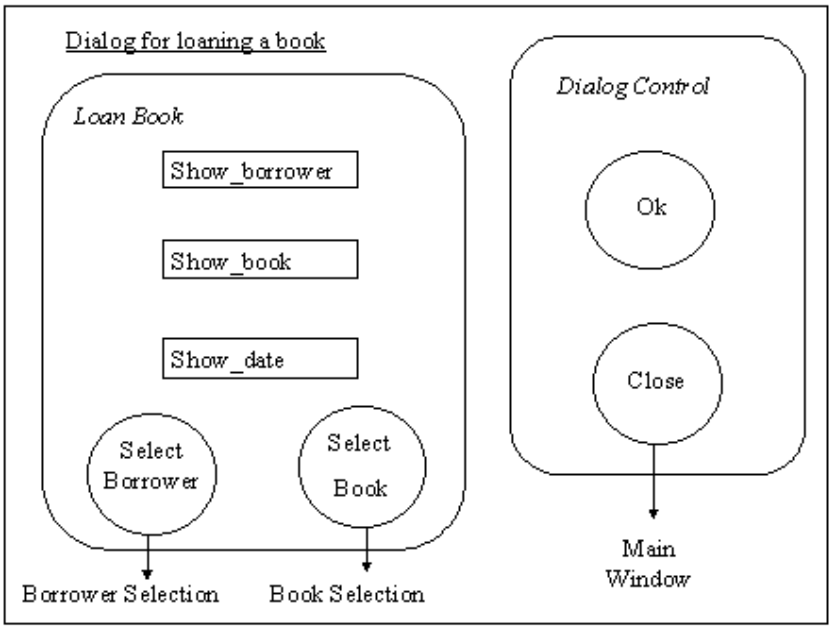


Fig. 3: Component Specification Diagram

***** *Components Behavior Definition Table* in which every one of the components and tools represented in the previous diagram is accurately described so that the user can understand the meaning of the components and tools through which he interacts and retrieves information from the system.

Design Phase:

The Graphical Models obtained in the first phase of Analysis are translated into an OASIS specification, which is the design tool within this methodology. This phase in still in under study. Up to know all the process of translating the Conceptual Scheme models of the information System into an OASIS specification is correctly carried out in automatic way. So, the idea is to do the same with the user interface models.

Implementation Phase:

This is the last step of the process. Implementation is tackled in a different way depending on the programming environment used.

In this phase we can use a Graphic User Interface Builder to get the visualisation of the specified interface.

3. Conclusion.

We can conclude that the methodology presented in this paper offers a set of operational methods which allows the design of user interfaces in an object oriented environment. It is not completely developed at this time, but, above all, it points out the importance of designing interaction between the end-user and the application before starting filling windows and windows with arbitrary sets of widgets (window gadgets) and other graphical elements. First to this, it is crucial to think about how should be the structure of the user interface of the system, considering first the general lines and then the details and final appearance, but only after having analysed the tasks, operations and components needed to fulfil the user's needs.

Besides, this object oriented approach facilitates the possibility of reusing user interface components. This possibility permits the construction of user interfaces form library classes with the advantage of implementing each individual feature only once.

4. Bibliography:

[Jaa95] Jaaksi, A. *Object-Oriented Specification of User Interfaces.* Software Practice and Experience. November 95

[Let98] Letelier, P., Ramos, I., Sanchez, P. Pastor,O. *OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling.* SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[Pas95] Pastor,O, Ramos,I. *OASIS 2.1.1.: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach.* Octubre 95 (3 ed.)

[Pas96] Pastor,O, Pelechano,V., Bonet, B., Ramos,I. *An OO Methodological Approach for Making Automated Prototyping Feasible.* Proc. Of DEXA-96, Springer-Verlag, September 1996.

[Pas97] Pastor, O., Insfran, E., Pelechano, V., Romero, J., Meseguer, J. *OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods.* Procs. of 9th. International Conference, CAiSE'97. Barcelona, Catalonia, Spain, June 16-20, 1997.

Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales*

Patricio Letelier **Pedro Sánchez** **Isidro Ramos**
Departamento Sistemas Informáticos y Computación
Universidad Politécnica de Valencia (España)
email {letelier | ppalma | iramos}@dsic.upv.es
http://www.dsic.upv.es/users/oom

Resumen

El modelo conceptual establece los requisitos funcionales del software. La prototipación es una de las técnicas usadas frecuentemente en la validación del modelo conceptual. Sin embargo, los resultados del proceso de validación, con o sin prototipación, distan de ser totalmente satisfactorios. La animación automática de modelos conceptuales emerge como un enfoque de prototipación que puede ayudar en la validación del modelo conceptual. Para realizar animación automática el principal obstáculo lo constituye la posibilidad de conseguir la ejecutabilidad de una especificación. En este sentido los métodos formales proveen ventajas significativas. Además del rigor y precisión provisto por un método formal para la especificación y verificación del modelo conceptual, respecto de su validación, puede definirse en mejores condiciones un proceso de animación automática que respete la semántica del modelo conceptual. $\mathcal{OASIS}$ es un enfoque formal para el modelado conceptual orientado a objeto. En este trabajo se presenta cómo se ha abordado la animación de especificaciones $\mathcal{OASIS}$ y en particular el modelo de ejecución utilizado. Dicho modelo es fiel a la semántica de $\mathcal{OASIS}$, por lo que constituye una base sólida para trabajos en validación de especificaciones $\mathcal{OASIS}$ mediante animación.

Palabras clave: modelo conceptual, especificación ejecutable, modelo de ejecución

1 Introducción

El modelo conceptual, representando los requisitos funcionales de un sistema de información, es la pieza clave para establecer el vínculo entre el espacio del problema y el espacio de la solución. La construcción del modelo conceptual es un proceso de descubrimiento, no sólo para el analista, sino también para el usuario. Durante el modelado conceptual se llevan a cabo esencialmente cuatro tareas: captura de requisitos, modelado o especificación, verificación de criterios de calidad y consistencia, y finalmente, validación. El grado de exhaustividad en la validación del modelo conceptual es un factor determinante para conseguir que el producto final se ajuste a lo esperado.

Para validar los requisitos del sistema lo más extendido es la utilización de técnicas de prototipación integradas a enfoques semiformales. Generalmente los prototipos son bosquejos de la interfaz del producto, concentrándose principalmente en aspectos de entrada y salida del sistema. Un tipo particular de prototipación está orientada a la simulación o animación detallada de la

*Este trabajo ha sido financiado por el proyecto MENHIR de la Comisión Interministerial de Ciencia y Tecnología, con referencia TIC97-0593-C05-01

Letelier, P., Sánchez, P., Ramos, I. Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales". Actas del VI Congreso Internacional de Investigación en Ciencias Computacionales (CIICC'99). 1999.

funcionalidad sistema. Esta forma de prototipación resulta atractiva para cubrir de forma exploratoria la validación del modelo conceptual, analizando su comportamiento ante situaciones de funcionamiento más complejas. El principal obstáculo para la animación del modelo conceptual es conseguir su ejecutabilidad, el esfuerzo de programación puede ser comparable al esfuerzo del desarrollo del producto final. En este sentido la utilización de un método formal para la especificación del modelo conceptual ofrece como ventaja el proporcionar una semántica precisa con la cual establecer un procedimiento automático para la animación de la especificación. Por lo anterior, resulta interesante combinar métodos formales y técnicas de prototipación. En los últimos años se ha registrado gran interés en torno a la validación de especificaciones formales mediante animación [11].

$\mathcal{OASIS}$ 3.0 (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [5] es un enfoque formal para la construcción de modelos conceptuales siguiendo el paradigma orientado a objeto. Otras propuestas formales con una motivación similar a la de $\mathcal{OASIS}$ son: TROLL [4], ALBERT [1] y OBLOG [9]. En este trabajo se expone resumidamente cómo hemos abordado la animación de especificaciones $\mathcal{OASIS}$ orientada a la exploración interactiva del comportamiento del modelo conceptual. Algunas propuestas relacionadas con validación mediante animación en otros modelos cercanos a $\mathcal{OASIS}$ son la presentada en [3] que utiliza TROLL y la desarrollada en [2], basada en ALBERT. Para OBLOG, se está construyendo una herramienta de animación pero aún no está disponible. Respecto de la animación establecida, las diferencias entre otras propuestas y este trabajo radican principalmente en las características formales del enfoque utilizado y la expresividad ofrecida. Además, en los trabajos relacionados señalados no se presentan detalles del modelo de ejecución utilizado. Finalmente, considerando los resultados publicados, el estado del arte es similar y se limita a versiones preliminares de entornos para animación.

A continuación se indica la organización del resto de este trabajo. En la sección 2 se exponen brevemente los aspectos básicos de $\mathcal{OASIS}$. La sección 3 presenta un modelo abstracto para la ejecución de especificaciones $\mathcal{OASIS}$. En la sección 4 se muestra un ejemplo de aplicación del modelo de ejecución. Finalmente, en la sección 5 se presentan las conclusiones y trabajo futuro.

2 $\mathcal{OASIS}$

Una especificación $\mathcal{OASIS}$ es una presentación de una teoría en el sistema formal usado, expresada como un conjunto estructurado de definiciones de clase. Las clases pueden ser simples o complejas. Una clase compleja es aquella que en su definición utiliza la definición de otras clases (simples o complejas). Las clases complejas se definen estableciendo relaciones entre clases. Éstas son agregación y especialización. Una clase se compone de un nombre de clase, uno o más mecanismos de identificación para instancias de la clase (objetos) y un tipo o plantilla que comparten todas las instancias. A continuación se presentan los conceptos básicos de $\mathcal{OASIS}$.

Definición 1 ***Plantilla o tipo***. *Una plantilla de clase está representada por la siguiente tupla:* $\langle Atributos, Eventos, Fórmulas \rangle$. *Atributos es el alfabeto de atributos. Eventos es el alfabeto de eventos. Fórmulas es un conjunto de fórmulas la variante de Lógica Dinámica [8] utilizada como formalismo subyacente.*

Para una clase simple todas las propiedades quedan establecidas en su plantilla. Para una clase compleja, además de las propiedades establecidas por su plantilla, existirán propiedades adicionales determinadas por los operadores de clase que la definen.

Definición 2 ***Servicio***. *Un servicio es un evento o una operación. En el primer caso se trata de un servicio atómico e instantáneo. Una operación es un servicio no atómico y que, en general, tiene duración.*

Definición 3 ***Acción***. *Una acción es una tupla ⟨Cliente,Servidor,Servicio⟩ que representa tanto la acción asociada a requerir el servicio en el objeto cliente como la acción asociada a proveer el servicio en el objeto servidor.*

Para cada clase, asumiremos la existencia implícita de un conjunto A de acciones obtenido a partir de los servicios que los objetos de la clase pueden requerir (cuando son clientes) o proveer (cuando son servidores). Al igual que para los eventos, $\forall a \in A$ podemos obtener $\underline{a} = \theta a$ siendo θ una substitución básica de los posibles cliente, servidor y servicio.

Definición 4 ***Atributo***. *Un atributo es un par ⟨nombre, sort de evaluación⟩, siendo nombre el identificador del atributo y sort de evaluación el nombre del conjunto soporte (carrier) con los valores que puede tomar dicho atributo.*

Definición 5 ***Estado del objeto***. *Llamaremos estado del objeto al conjunto de sus atributos evaluados. En $\mathcal{OASIS}$ el estado se expresa mediante Fórmulas bien formadas (Fbfs) de lógica de primer orden.*

El estado permite caracterizar a un objeto en un instante determinado. Llamaremos Σ al conjunto de estados alcanzables por un objeto y $\sigma \in \Sigma$ un estado en particular. El estado de un objeto cambia por la ocurrencia de acciones. Los cambios de estado son modelados por la siguiente relación:

$$efecto : \underline{A} \rightarrow (\Sigma \rightarrow \Sigma)$$

Donde $\underline{A}$ es el conjunto de acciones instanciadas ($\underline{a} \in \underline{A}$) que le pueden acontecer al objeto y $efecto$ es la función que asocia a cada acción instanciada una función de cambio de estado, siendo $(\Sigma \rightarrow \Sigma)$ el conjunto de todas las funciones de cambio de estado.

Definición 6 ***Conjunto consistente de acciones***. *Un conjunto de acciones $M \subseteq \underline{A}$ es consistente si a partir de un estado σ dado, el estado σ' alcanzado por la ocurrencia de las acciones en M es siempre el mismo, independiente del orden en el cual se ejecuten*[1] *las acciones.*

Definición 7 ***Paso***. *Llamaremos paso a un conjunto consistente de acciones que ocurren en un mismo instante de la vida del objeto.*

Definición 8 ***Vida o traza de un objeto***. *Llamaremos vida o traza de un objeto a un prefijo finito de pasos que le acontecen al objeto. De esta forma, la traza es una caracterización alternativa del estado del objeto en un instante determinado.*

2.1 $\mathcal{OASIS}$ expresado en Lógica Dinámica

En $\mathcal{OASIS}$ el comportamiento de los objetos de una clase está determinado por un conjunto de fórmulas en la variante de Lógica Dinámica utilizada. Dichas fórmulas establecen prohibiciones, obligaciones y cambios de estado para los objetos, ellas se muestran a continuación.

$\psi \rightarrow [a]false$	"La ocurrencia de a está prohibida en los estados que satisfacen ψ".
$\psi \rightarrow [\neg a]false$	"La ocurrencia de a es obligatoria en los estados que satisfacen ψ"
$\psi \rightarrow [a]\phi$	"En los estados que satisfacen ψ, inmediatamente después de la ocurrencia de la acción a, ϕ debe satisfacerse".

[1] Se utilizará "ejecución de acción" como sinónimo de "ocurrencia de acción" para hacer más intuitiva la presentación. Sin embargo, desde el punto de vista declarativo y formal, "ocurrencia" es el término más adecuado.

Donde a representa la ocurrencia de una acción. Por otra parte, ψ y ϕ son Fbfs en lógica de primer orden que caracterizan el estado del objeto en un instante determinado. Además $\neg a$ representa la no-ocurrencia de la acción a (es decir, que no ocurre nada o que ocurre otra acción distinta de a). Finalmente, $false$ es un átomo tal que no hay un estado que pueda hacerlo verdadero. Esta situación representa la no-existencia de un estado válido alcanzable.

El siguiente ejemplo modela una cuenta bancaria utilizando la sintaxis del lenguaje $\mathcal{OASIS}$.

```
class cuenta
identification
    numero:(numero);
constant attributes
    numero:nat;
    titular:string;
variable attributes
    saldo:nat(0);
    veces:nat(0);
    pin:nat(0);
    categoria:nat(0);
derived attributes
    buen_saldo:bool;
derivations
    good_balance:={saldo>=100};
events
    abrir new;
    cerrar destroy;
    ingreso(Cantidad:nat);
    reintegro(Pin:nat,Cantidad:nat);
    pagar_comision;
    cambiar_pin(Pin:nat,NuevoPin:nat);
    asignar_categoria(Categoria:nat);
valuations
    [ingreso(Cantidad)] saldo:=saldo+Cantidad, veces:=veces+1;
    [reintegro(Pin,Cantidad)] saldo:=saldo-Cantidad, veces:=veces+1;
    [pagar_comision[2]] saldo:=saldo-1;
    [::pagar_comision] veces:=0;
    [cambiar_pin(Pin,NuevoPin)] pin:=NuevoPin;
    [asignar_categoria(Categoria)] categoria:=Categoria;
preconditions
    reintegro(Pin,Cantidad) if {(pin=Pin and saldo>=Cantidad) or
            (pin=Pin and saldo<Cantidad and categoria=2)};
    cambiar_pin(Pin,NuevoPin) if {pin=Pin};
    cerrar if {saldo=0};
triggers
    ::pagar_comision when {veces>=3 and buen_saldo=false and categoria=0};
end class
```

[2] En OASIS, para evitar detallar cada elemento de la tupla $< Cliente, Servidor, Servicio >$ que representa a una acción, se adoptan las siguientes simplicaciones: a) Si el servidor es el propio objeto, se escribe *Cliente*:*Servicio* y si el cliente no es relevante sólo se escribe *Servicio*. b) Si el cliente es el propio objeto, se escribe *Servidor*::*Servicio*. *c)* En el caso particular de comunicación *self*, la acción de requerir se escribe *::Servicio* y la acción de proveer *self:Servicio* (o simplemente *Servicio*).

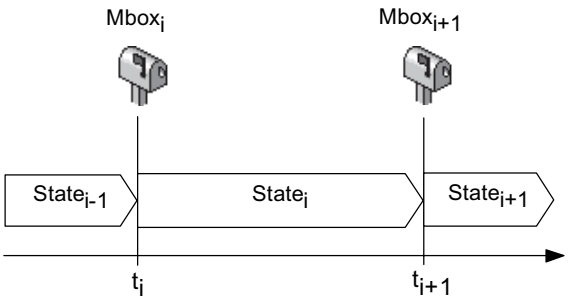


Figura 1: Ciclo de vida de un objeto

3 Un modelo abstracto para animación en $\mathcal{OASIS}$

El modelo abstracto de ejecución establece el vínculo entre la semántica de una especificación $\mathcal{OASIS}$ y una implementación concreta en un entorno de programación. Así, en nuestro caso, el modelo de ejecución no es más que un modelo para animar fórmulas de obligación, permiso y evaluación, asociadas a un objeto. En esta perspectiva, la información utilizada por el modelo es la signatura de atributos y acciones del objeto junto al conjunto de fórmulas en Lógica Dinámica que describen su comportamiento. Cuando se trata con objetos de clases simples esta información es precisamente la plantilla de la clase del objeto. Sin embargo, si la clase del objeto es compleja (por especialización o agregación) no toda la información está directamente en la plantilla de la clase del objeto. A pesar de esto, podemos suponer externo al modelo de ejecución el trabajo que determina la estructura y el comportamiento del objeto en el caso de objetos pertenecientes a clase complejas y así reducir el planteamiento del modelo de ejecución al caso de un objeto instancia de una clase simple.

3.1 Aspectos básicos

La secuencia de pasos que le acontecen a un objeto está ordenada respecto del tiempo. Podemos suponer que existe un objeto "reloj" que envía acciones especiales –llamadas *ticks*- hacia cada objeto del sistema. Los *ticks* recibidos por un objeto son correlativos con los números naturales, t_1, t_2, etc. Siendo i y j números naturales entonces se cumplirá que $i < j \Longleftrightarrow t_i < t_j$.

Definición 9 ***Buzón****. Llamaremos buzón al conjunto de acciones que pueden formar parte del paso que un objeto ejecuta en un determinado tick. El buzón en el instante t_i se denota por $Mbox_i$. Consideraremos el buzón como un buffer ilimitado, en el cual las acciones se ordenan por instante de llegada.*

Definición 10 ***Estado i-ésimo del objeto****. Denotado por $State_i$, representa el estado del objeto en el intervalo $(t_i, t_{i+1}]$. Es decir, inmediatamente después de t_i y hasta t_{i+1} inclusive, el estado se mantiene constante.*

La Figura 1 ilustra la vida de un objeto según las definiciones dadas. El conjunto de acciones en cada paso es obtenido desde el buzón de acciones del objeto y la ejecución del paso en un instante produce un cambio de estado, el cual no será observable hasta el siguiente *tick*. Así, en la estructura de tiempo utilizada aunque la ocurrencia de acciones es instantánea, los estados tienen una cierta duración.

3.2 Concurrencia Intra-Objetual

La concurrencia intra-objetual es imprescindible para una animación fiel a la semántica de $\mathcal{OASIS}$. Básicamente dos situaciones justifican la incorporación de concurrencia intra-objetual en el modelo de ejecución:

- Cuando en un instante más de una acción está obligada a ejecutarse, deben todas ellas realizarse en el mismo instante de lo contrario se contradice la semántica de las fórmulas de obligación correspondientes.
- En el caso de objetos agregados, la ejecución concurrente de los objetos componentes en forma natural puede producir acciones concurrentes vistas desde el objeto agregado.

3.3 Conflicto entre acciones

Si se ejecuta más de una acción en un paso deberían garantizarse dos propiedades: la consistencia y la terminación, o al menos indicar que dichas propiedades no se pueden determinar con certeza en la especificación. La propia definición del concepto de paso establece dicha consistencia, sin embargo, para el establecimiento del modelo de ejecución a continuación se explicará cómo se aseguran dichas propiedades.

Definición 11 ***Relación afecta****. Llamaremos Att al conjunto de atributos del objeto. Se define la relación afecta como:*

$$Afecta_{State_i} \subseteq Mbox_i \times Att$$

Siendo $a_1 \in Mbox_i$ y $att_1 \in Att$, diremos que $a_1 Afecta_{State_i} att_1$ si y sólo si la ejecución de la acción a_1 modifica[3] el valor del atributo att_1 en t_i.

Definición 12 ***Conflicto entre acciones****. Sea $Acc \subseteq Mbox_i$ y Att el conjunto de atributos del objeto. Diremos que Acc tiene conflicto entre acciones si NO se satisface la siguiente condición:*

$$\forall a_1, a_2 \in Acc \text{ y } \forall att \in Att : [a_1 Afecta_{State_i} att \wedge a_1 Afecta_{State_i} att] \Longrightarrow a_1 = a_2$$

Definición 13 ***Conjunto consistente de acciones (versión modelo de ejecución)****. Sea $Acc \subseteq Mbox_i$, diremos que Acc es consistente si no presenta conflicto entre acciones.*

Así, para garantizar la consistencia de un conjunto de acciones en el buzón bastará con asegurar que dichas acciones no están en conflicto. Esto puede detectarse automáticamente, basta con inspeccionar los conjuntos de atributos que pueden sufrir una asignación de valor para cada acción, revisando las evaluaciones asociadas a dicha acción. Los conjuntos de atributos obtenidos deben ser disjuntos para que el conjunto de acciones correspondiente sea consistente. Nótese que cuando existe conflicto entre acciones puede aún existir consistencia (según su definición original). Por ejemplo, consideremos las siguientes evaluaciones:

$$att = V[a_1]att = V + 1$$
$$att = V[a_2]att = V + 2$$

[3]La modificación puede ser indirecta en el caso de atributos derivados. Así, si att_2 es un atributo derivado definido en función de att_1, entonces: $a_1 Afecta_{State_i} att_1 \Longrightarrow a_1 Afecta_{State_i} att_2$.

Las acciones a_1 y a_2 están en conflicto pero al aplicar a_1 y luego a_2 el efecto es el mismo que al hacerlo en el otro orden, con lo cual a_1 y a_2 constituyen un conjunto consistente (según la definición de consistencia original). Sin embargo, la exclusión de estos casos no contradice la semántica de $\mathcal{OASIS}$ y facilita la determinación de los conjuntos de acciones que pueden ejecutarse en un paso.

Una vez seleccionado el conjunto consistente de acciones que forma el paso, se aplican todas las acciones a la vez, produciendo un sólo cambio de estado correspondiente al efecto de todas las acciones. Debido a esto último, la propiedad de terminación en el procesamiento de las acciones contenidas en un paso está siempre asegurada.

3.4 Clasificación de acciones en el buzón

El procesamiento de las acciones del buzón implica clasificar las acciones contenidas en él. A continuación se presentan las categorías de acciones identificables en el buzón, caracterizadas como subconjuntos de $Mbox_i$ (es decir, en el instante t_i).

Definición 14 ***Acciones obligadas de requerir****. Acciones asociadas a obligaciones por requerir un servicio (en las cuales el cliente es el propio objeto) y que* ***deben*** *acontecer. El conjunto de acciones obligadas de requerir se denota por* $OReq_i$.

Las acciones de requerir están determinadas por fórmulas de obligación, es decir, de la forma: $\phi[\neg a]false$, en las cuales a es la tupla $< self, Servidor, Servicio >$, es decir, el cliente de la acción es el propio objeto, indicado usando la palabra $self$.

Definición 15 ***Acciones obligadas de proveer****. Acciones correspondientes a servicios que el objeto* ***debe*** *proveer. Estas acciones están asociadas a obligaciones, en las cuales el cliente no es el propio objeto. El conjunto de acciones obligadas de proveer en se denota por* $OProv_i$.

Las acciones obligadas de proveer están determinadas por fórmulas de obligación, es decir, de la forma: $\phi[\neg a]false$, en las cuales a es la tupla $< Cliente, self, Servicio >$, es decir, el servidor de la acción es el propio objeto.

Definición 16 ***Acciones obligadas de ejecutar****. Este conjunto se denotado por* $OExec_i$ *y se define como:* $OExec_i = OReq_i \cup OProv_i$.

Supondremos que el garantizar la no-existencia de conflicto entre acciones obligadas es parte de la verificación estática de la especificación. Es decir, en tiempo de ejecución no puede presentarse conflicto entre obligaciones. Del mismo modo, supondremos que no puede darse una situación en la cual una acción obligada no está permitida.

Definición 17 ***Acciones no-obligadas de ejecutar****. Acciones correspondientes a servicios solicitados por otros objetos (o por el propio objeto) y que el objeto* ***puede*** *proveer, dependiendo de los permisos establecidos sobre dichas acciones y verificados sobre* $State_i$. *El conjunto de acciones no-obligadas se denota por* $\overline{O}Exec_i$.

Definición 18 ***Acciones rechazadas****. Es un subconjunto de* $\overline{O}Exec_i$ *constituido por acciones no-obligadas de ejecutar cuya ocurrencia no está permitida. El conjunto de acciones rechazadas se denota por* $Reject_i$.

Las acciones rechazadas están determinadas por fórmulas de prohibición, es decir, de la forma: $\phi[a]false$.

Definición 19 ***Acciones candidatas****. Es un subconjunto de* $\overline{O}Exec_i$ *constituido por acciones no-obligadas de ejecutar cuya ocurrencia está permitida. El conjunto de acciones candidatas se denota por* $Cand_i$*.*

Además, se cumple que $\overline{O}Exec_i = Reject_i \cup Cand_i$.

Definición 20 ***Acciones ejecutadas****. Es el conjunto de acciones correspondiente al paso que se ejecuta en* t_i*. El paso se denota por* $Exec_i$ *y estará constituido por* $OExec_i$ *más un subconjunto de* $Cand_i$*, es decir, acciones no-obligadas de ejecutar que están permitidas.*

La selección del subconjunto de acciones de $Cand_i$ que serán incluidas en $Exec_i$ puede responder a distintos criterios y la única restricción es que no se produzcan conflictos entre las acciones del paso.

Definición 21 ***Acciones en conflicto****. Es un subconjunto de* $Cand_i$ *constituido por acciones no-obligadas de ejecutar cuya ocurrencia está permitida, pero que están en conflicto con alguna de las acciones obligadas de ejecutar o con alguna de las acciones no-obligadas seleccionadas. El conjunto de acciones en conflicto se denota por* $Conf_i$*.*

Se utiliza un criterio sencillo para la selección de acciones desde $Cand_i$: ante dos acciones que no pueden ser seleccionadas por estar en conflicto se seleccionará aquella llegada antes al buzón. Este criterio evita postergación indefinida de acciones en conflicto. Las acciones en conflicto $Conf_i$ serán "copiadas" al buzón correspondiente al siguiente instante ($Mbox_{i+1}$).

3.5 Comunicación self

La comunicación del objeto consigo mismo se produce cuando una acción obligada de requerir es dirigida hacia el mismo objeto (el servidor es $self$). Esta comunicación es tratada de forma homogénea al resto de comunicaciones, es decir, la acción obligada de requerir será posteriormente recibida en el buzón como una acción no-obligada (una solicitud de servicio). Según esto, en dicha situación se distinguen dos acciones: la acción de requerir el servicio a sí mismo (obligación) y la acción no-obligada de proveer dicho servicio. En general la segunda acción podría sufrir algún retraso y ser procesada en un intervalo de tiempo posterior. La primera por tratarse de una obligación no puede ser rechazada, la segunda sí.

3.6 Actividad en el objeto

El comportamiento de un objeto estará regido por un bucle de actividad interna. Esta actividad realiza la transición del objeto desde un estado inicial $State_i$ a un estado final $State_{i+1}$. Utilizaremos una acción implícita para representar los *ticks* e incluirlos homogéneamente dentro del buzón. Un *tick* corresponderá a la acción $< reloj, Object, tick(t, i) >$. Donde $Object$ es una referencia de objeto servidor que coincide con el propio objeto, es decir, se trata como una acción de servicio solicitada por "reloj" y t es el tiempo real correspondiente al i-ésimo *tick*. La separación real de tiempo entre *tick* y *tick* podría no ser constante.

La Figura 2 ilustra los principales aspectos considerados en el modelo de ejecución. Un objeto transita desde el estado $State_i$ hacia el estado $State_{i+1}$ por la ocurrencia del paso representado por $Exec_i$ correspondiente a un subconjunto de acciones de $Mbox_i$. La comunicación desde otros objetos (o desde el propio objeto) se representa por las flechas etiquetadas como $\overline{O}Exec_i$ y $\overline{O}Exec_{i+1}$. La comunicación hacia otros objetos se representa en la parte inferior de la Figura 2

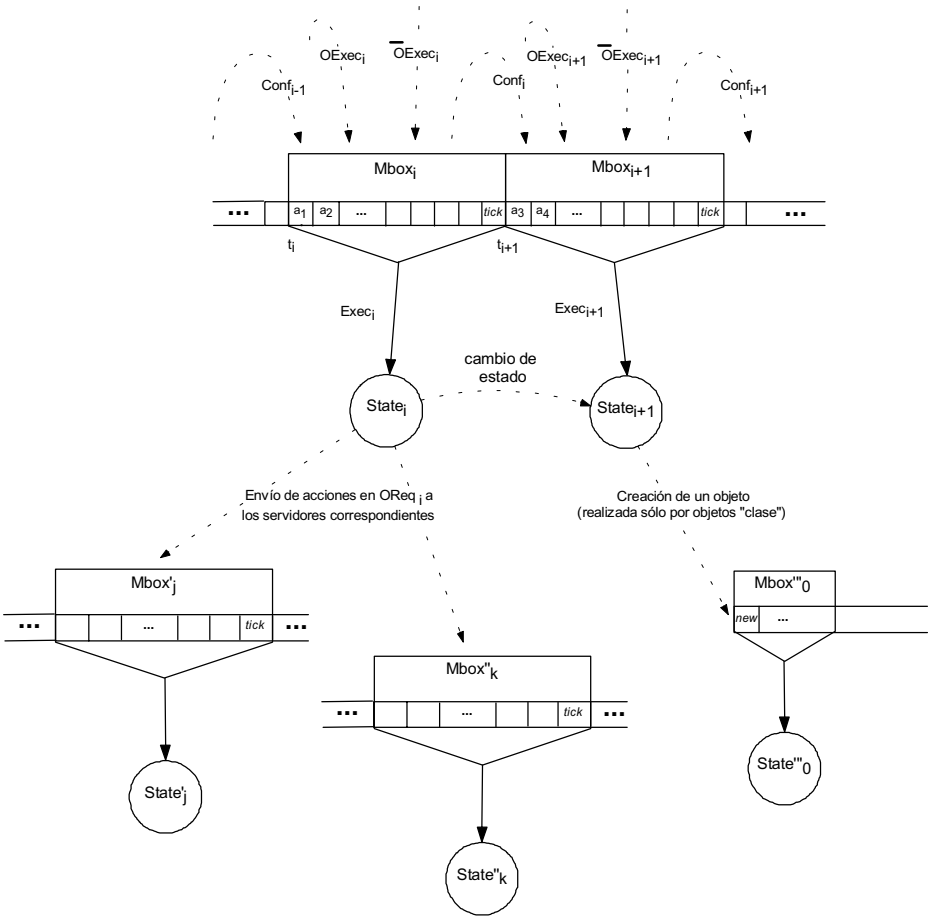


Figura 2: Vida de un objeto en OASIS

por las flechas saliendo hacia buzones de otros objetos. La flecha etiquetada con $Conf_i$ representa las acciones en conflicto en t_i, las cuales son copiadas al siguiente buzón. La flecha etiquetada con $OExec_{i+1}$ representa las obligaciones de requerir que el objeto adquirirá en el instante t_{i+1} producto de la ocurrencia de las acciones $Exec_i$.

Las clases en ejecución serán implementadas homogéneamente como objetos. Estos *metaobjetos* proveen el servicio de creación de objetos de la clase correspondiente. Por lo tanto, como ilustra la Figura 2 en su parte inferior derecha, si un *metaobjeto* ejecuta una acción de creación de instancias (acción *new*) dentro de su proceso de cambio de estado se incorporará la creación de un nuevo objeto, cuya primera acción ejecutada será la acción *new*.

Un objeto examinará su *Mbox* sólo cuando ha llegado al menos una acción *tick*. Si el objeto termina el cambio de estado y aún no se recibe la acción *tick*, esperará hasta que ésta llegue. Por el contrario, si el objeto termina el cambio de estado y ha llegado más de una acción *tick*, sólo se considerará la último, las otras son ignoradas. Nótese que la frecuencia de *tick* determina el tamaño del *Mbox* que se procesará, por lo tanto si se aumenta la frecuencia de *ticks (*sin variar la frecuencia de acciones que llegan al *Mbox*) se puede llegar al extremo en el cual el buzón contiene sólo una acción además de la acción *tick* o incluso sólo la acción *tick*. Por otra parte, reduciendo la frecuencia de *ticks* el buzón tiene la posibilidad de contener mayor cantidad de acciones lo cual podría producir más acciones en conflicto. Cualquiera sea el caso, la frecuencia de *ticks* y el ordenamiento de acciones del buzón representan el indeterminismo propio de los sistemas concurrentes. Sin embargo, la fidelidad respecto de la semántica de $\mathcal{OASIS}$ siempre se mantiene.

La semántica asociada a las acciones obligadas de proveer exige un tratamiento especial. Aunque ya esté presente un *tick* en el $Mbox$ el objeto no puede alcanzar el nuevo estado si no están presentes en él las acciones obligadas de proveer. Esto ocasionará una posible espera, en la cual podrán llegar nuevos *ticks* que serán ignorados. Sólo cuando todas las acciones obligadas de proveer están contenidas en el buzón y ha llegado un *tick* (en un instante posterior) el objeto procesará el buzón y alcanzará el estado siguiente.

Finalmente, el siguiente algoritmo expresa la actividad interna de cada objeto durante su existencia:

$i = 0$
$OProv_i = \{\}$
$State_0 \leftarrow \{\}$
REPETIR
    ESPERAR HASTA QUE $(tick \in Mbox_i \wedge OProv_i \subseteq Mbox_i)$
    $OReq_i \leftarrow obtener\_acciones\_de\_requerir(Mbox_i)$
    $enviar\_acciones\_de\_requerir(OReq_i)$
    $\overline{O}Exec_i \leftarrow obtener\_acciones\_no\_obligadas(Mbox_i)$
    $Reject_i \leftarrow calcular\_acciones\_rechazadas(\overline{O}Exec_i, State_i)$
    $Cand_i \leftarrow \overline{O}Exec_i - Reject_i$
    $Conf_i \leftarrow calcular\_acciones\_en\_conflicto(Cand_i, State_i)$
    $Exec_i \leftarrow OReq_i \cup OProv_i \cup (Cand_i - Conf_i)$
    $State_{i+1} \leftarrow determinar\_nuevo\_estado(Exec_i, State_i)$
    $OProv_{i+1} \leftarrow calcular\_próx\_accs\_obligadas\_de\_proveer(State_{i+1})$
    $OReq_{i+1} \leftarrow calcular\_próx\_accs\_obligadas\_de\_requerir(State_{i+1})$
    $Mbox_{i+1} \leftarrow Mbox_{i+1} \cup OReq_{i+1} \cup Conf_i$
    $i = i + 1$
FIN_REPETIR

Observaciones

- Las funciones "*obtener_...*" seleccionan subconjuntos de $Mbox$ de acuerdo con las definiciones dadas.
- El procedimiento *enviar_acciones_de_requerir* se encarga de hacer llegar las acciones de requerir a los buzones de los respectivos servidores.
- Las funciones "*calcular_...*" utilizan además las fórmulas que definen el comportamiento del objeto (fórmulas de obligaciones, permisos y cambios de estado).
- La función *determinar_nuevo_estado* obtiene el estado siguiente utilizando además las fórmulas de cambio de estado.
- Mediante las definiciones y comentarios respecto de cada uno de los conjuntos involucrados todas las funciones pueden ser implementadas sin ambigüedad en el entorno de programación concurrente elegido.
- El bucle de actividad del objeto puede terminar si el objeto ejecuta una acción asociada al evento *destroy*. Esto podría ocurrir sólo si dicho evento está definido en la clase del objeto.

4 El modelo de ejecución en funcionamiento

Usando el ejemplo de la cuenta bancaria, realizaremos una animación basada en el modelo de ejecución propuesto. La Figura 3 muestra parte del ciclo de vida de un objeto de la clase `cuenta` (entre t_1 y t_5). Para esta instancia de la clase cuenta consideraremos que su estado actual (hasta t_1) es: $State_1 = \{< num, 1010 >, < nombre, juan >, < saldo, 10 >, < veces, 1 >, < categoria, 1 >, < buen\_saldo, false >\}$

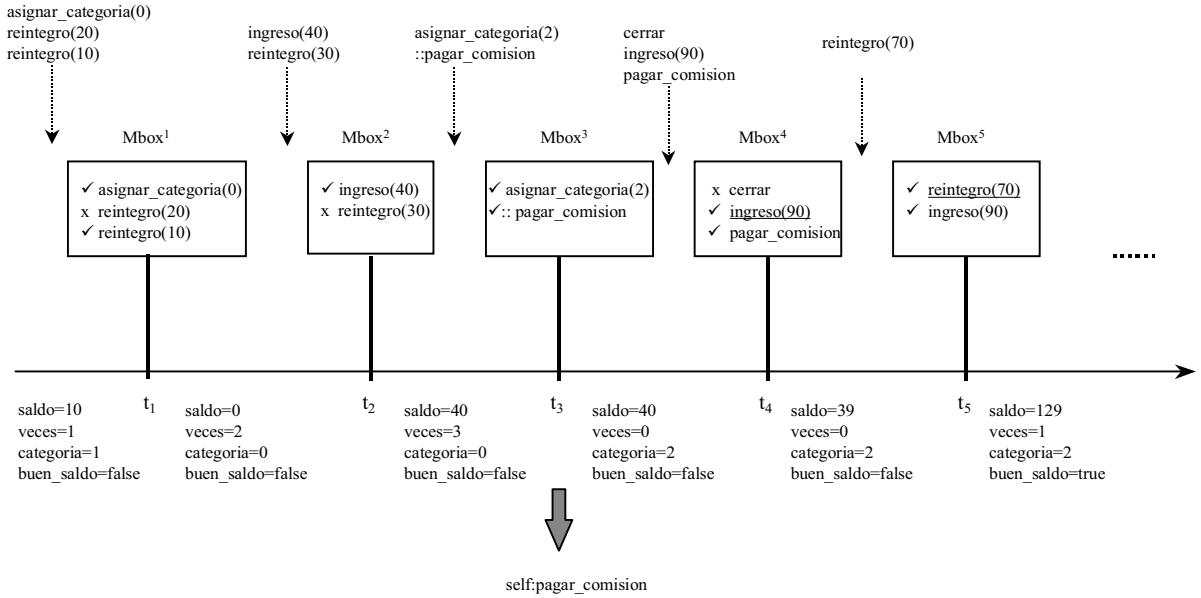


Figura 3: Ciclo de vida de una instancia de cuenta entre t_1 y t_5

El eje horizontal representa el tiempo de izquierda a derecha. Bajo el eje del tiempo se indican los valores de atributos variables del objeto. Entre $(t_i - t_{i+1}]$ el estado del objeto no cambia. Sobre el eje del tiempo y bajo los rectángulos se indica los cambios que sufrirán los atributos variables y que se reflejarán en el siguiente t_i. Los rectángulos representan el buzón ($Mbox_i$) y su contenido en cada. El símbolo "✓"delante de una acción en el $Mbox_i$ indica que está permitida en dicho instante, el símbolo "`x`" delante de una acción indica lo contrario. Las acciones en conflicto y no seleccionadas han sido subrayadas. Las acciones no-obligadas recibidas entre $(t_i - t_{i+1}]$ están representadas en las flechas descendientes entre $Mbox_i$ y $Mbox_{i+1}$. La flecha ancha inferior representa acciones obligadas de requerir.

Después de t_2, y sabiendo que en t_3 será satisfecha la obligación que produce una acción por requerir el servicio `pagar_comision`, es generada automáticamente la correspondiente acción de requerir en $Mbox_3$. En t_3 se tiene en el buzón una acción de requerir asociada a la obligación de solicitar a sí mismo el servicio `pagar_comision`. Posteriormente, se recibe como la correspondiente acción de solicitud de servicio y es ejecutada en t_4. En t_4 y t_5 el buzón contiene acciones que están en conflicto pues el atributo `saldo` puede ser modificado por más de una acción. En t_4 la acción cuyo servicio es `ingreso(90)` está en conflicto con la acción cuyo servicio es `pagar_comision`. Así, la acción cuyo servicio es `ingreso(90)` es postergada hasta el siguiente *tick*, siendo copiada a $Mbox_5$. En t_5 la acción cuyo servicio es `ingreso(90)` está en conflicto con la acción cuyo servicio es `reintegro(20)`. En este caso, esta última acción se ha copiado en el siguiente buzón.

5 Conclusiones y Trabajo Futuro

La animación de especificaciones formales resulta de gran interés para ayudar en la validación temprana de requisitos de un sistema. El principal obstáculo es obtener un prototipo ejecutable de la especificación. En este trabajo se ha presentado un modelo abstracto para la ejecución de especificaciones $\mathcal{OASIS}$. Utilizando este modelo de ejecución se han construido traductores de $\mathcal{OASIS}$ a distintos entornos de programación [7, 10]. Además, se ha construido una versión preliminar de un entorno para especificación incremental y validación de requisitos incluyendo animación automática de modelos $\mathcal{OASIS}$ [6]. En un trabajo posterior, el modelo de ejecución debería ser extendido para incluir aspectos más complejos del comportamiento tales como: restricciones de integridad, operaciones etiquetadas como transacción y otros mecanismos de comunicación más complejos incluidos en $\mathcal{OASIS}$.

Referencias

[1] Dubois,E., Du Bois P., Petit M. *O-O Requirements Analysis: An Agent Perspective*. In Proc. of the 7th European Conf. on OO Programming ECOOP'93, pp.458-481. July 1993.

[2] Heymans P. *The Albert II Specification Animator*. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm.

[3] Herzig R., Gogolla M. *An Animator for the Object Specification Language TROLL Light*. Proc. Colloq. on Object-Orientation in Databases and Software Engineering, 1994.

[4] Jungclaus R., Saake G., Hartmann T., Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems*. ACM Transactions on Information Systems, Volume 14, Number 2, pp.175-211, April 1995.

[5] Letelier P., Ramos I., Sánchez P., Pastor O. *OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*. Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV-98.4011, 1998.

[6] Letelier P., Sánchez P., Ramos I.. *Un Ambiente para Especificacion Incremental y Validacion de Modelos Conceptuales*. Actas del 2°Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software IDEAS'99, pp.216-228, Costa Rica, 1999.

[7] Letelier P., Sánchez P., Ramos I.. *Prototyping a Requirements Specification through an Automatically Generated Concurrent Logic Program*. In Proc. of First Int.Workshop on Practical Aspects of Declarative Languages, LCNS 1551, pp.31-45, Springer-Verlag, 1998.

[8] Meyer J.-J.Ch. *A Different Approach to Deontic Logic: Deontic Logic viewed as a Variant of Dynamic Logic*. In Notre Dame Journal of Formal Logic, vol.29, pp.109-136, 1988.

[9] OBLOG Software S.A. *The OBLOG Software Development Approach (White Paper)*. 1999, `http://www.oblog.pt/Download/Documentation.exe`

[10] Sánchez P., Letelier P., Ramos I. *Una Aproximación a la Representación de OASIS Gráfico en Redes de Petri Orientadas a Objeto*. Actas de las III Jornadas de Ingeniería de Software, JIS'98, pp.267-280, Universidad de Murcia, Murcia, 1998.

[11] Siddiqi J., Morrey I.C., Roast C.R., Ozcan M.B. *Towards Quality Requirements via Animated Formal Specifications*. Annals of Software Engineering, Vol.3, pp.131-155, 1997.

Modelado Conceptual con un Lenguaje Formal y Orientado a Objeto*

Pedro Sánchez **Patricio Letelier** **Isidro Ramos** **Oscar Pastor**

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia - España
{ppalma, letelier, iramos, opastor}@dsic.upv.es
http://www.dsic.upv.es/users/oom

Palabras Clave: Ingeniería de Requisitos, Modelado Conceptual, Modelado Orientado a Objeto

Resumen

El modelado conceptual tiene por objetivo el establecer los requisitos funcionales del sistema. El modelo conceptual establece el vínculo entre el espacio del problema y el espacio de la solución, constituyendo la base para las tareas posteriores en el desarrollo del sistema de información. Los enfoques más populares para abordar el modelado conceptual están basados en notaciones informales, donde la verificación y validación suelen ser poco exhaustivas y difíciles de realizar. Los métodos formales ofrecen una mejora en estos aspectos pero su introducción en entornos industriales de desarrollo de software no se ha popularizado por carencia de métodos y herramientas adecuadas. $\mathcal{OASIS}$ es un enfoque formal para el modelado conceptual orientado a objeto. En $\mathcal{OASIS}$, desde sus primeras versiones se ha puesto especial hincapié en su utilización como modelo subyacente en métodos y herramientas de aplicabilidad industrial. Así, siguiendo el modelo $\mathcal{OASIS}$, se han desarrollado una serie de herramientas integradas en un entorno CASE y un método ad hoc denominado OO-METHOD. Este artículo presenta algunas de las principales características incluidas en la última versión de $\mathcal{OASIS}$. Se describen brevemente los conceptos generales del modelo $\mathcal{OASIS}$ y se recorren algunos aspectos del lenguaje de especificación asociado.

1 Introducción

Generalmente, los requisitos del usuario son representados usando modelos semiformales. Notacionalmente los métodos orientados a objeto más populares han sido fundidos en UML (*Unified Modeling Language*) [12]. En los métodos semiformales la verificación y validación suele ser poco exhaustiva, obteniendo finalmente una implementación en un lenguaje imperativo sobre el cual es muy difícil estudiar su comportamiento y establecer su corrección. Muchos fracasos en proyectos de software han sido atribuidos a problemas en ingeniería de requisitos, tales como: requisitos mal documentados, requisitos imposibles de satisfacer o requisitos que no corresponden a las necesidades del usuario. En este sentido, y particularmente en el modelado conceptual, es interesante el uso de métodos formales.

*Este trabajo está financiado por el proyecto *MENHIR* (Métodos, Entornos y Herramientas para Ingeniería de Requisitos), de la Comisión Interministerial de Ciencia y Tecnología, con referencia TIC97-0593-C05-01.

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. Modelado conceptual con un lenguaje formal y orientado a objeto. Actas de la XXV Conferencia Latinoamericana de Informática (CLEI'99). 1999.

OASIS (*Open and Active Specification of Information Systems*) [5] es un enfoque formal para la especificación de modelos conceptuales siguiendo el paradigma orientado a objetos. Las propuestas formales más cercanas a *OASIS* son: TROLL [4], LCM [3], ALBERT [2] y OBLOG [14]. En términos generales, a diferencia de otros enfoques, en *OASIS* desde sus primeras versiones se ha puesto especial interés en tres aspectos esenciales que facilitan su integración en entornos industriales de desarrollo de software:

- Establecimiento de un método de desarrollo ad hoc para *OASIS*, llamado OO-METHOD [10].
- Generación automática de código a partir de especificaciones *OASIS*, orientada a la validación [6, 7] y construcción incremental del producto final [11].
- Construcción de herramientas que soporten el proceso de modelado conceptual. Se ha construido un entorno CASE basado en la versión precedente de *OASIS* y adaptado a OO-METHOD.

En este aspecto en OBLOG también el esfuerzo se han orientado hacia herramientas y un soporte metodológico que permita su uso industrial. Las otras propuestas se han mantenido principalmente en el ámbito académico.

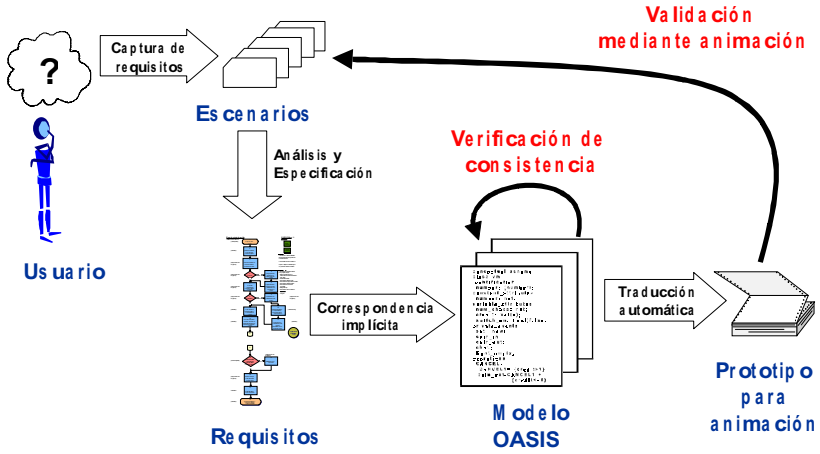


Figura 1: Modelado Conceptual basado en OASIS

La Figura 1 ilustra un proceso de modelado conceptual basado en *OASIS*. La captura de requisitos se realiza mediante técnicas de especificación de escenarios que permitan su comparación total o parcial respecto al comportamiento exhibido por un prototipo del sistema. A partir de los escenarios establecidos se obtienen los elementos de especificación que forman el modelo conceptual. La especificación se realiza mediante técnicas textuales y gráficas que de forma implícita determinan el modelo *OASIS* del sistema. El modelo conceptual es verificado según las reglas de construcción de *OASIS*. Mediante generación automática se obtiene un prototipo con funcionalidad equivalente a la especificación *OASIS*. A través de animación del prototipo la validación se orienta a la satisfacción de los requisitos expresados en los escenarios.

El objetivo de este trabajo es presentar las principales características de *OASIS* 3.0, la última versión publicada de *OASIS*. El resto del trabajo está organizado como se detalla a continuación. En la sección 2 se introducen los conceptos básicos de *OASIS*. Posteriormente,

la sección 3 resume los aspectos formales de $\mathcal{OASIS}$. En la sección 4 se presentan los apartados de especificación en una plantilla de clase. El capítulo 5 describe las relaciones entre clases. Finalmente, en el capítulo 6 se exponen las conclusiones.

2 Conceptos Básicos de $\mathcal{OASIS}$

En $\mathcal{OASIS}$, un **objeto** es un proceso observable cuya vida está caracterizada por la ocurrencia de acciones, que corresponden a servicios requeridos o provistos por el objeto, es decir, un objeto puede actuar como cliente o como servidor. Los servicios que un objeto proporciona a nivel "atómico" se denominan **eventos**. Cada objeto tiene un evento de creación (que inicia su vida) y opcionalmente uno de destrucción. Una **acción** es una tupla formada por el cliente, el servidor y el servicio solicitado.

La **vida de un objeto** puede representarse como una secuencia de **pasos**. Cada paso está formado por un conjunto de acciones que ocurren en un instante dado de la vida del objeto.

Los eventos pueden ser estructurados como **procesos** en un nivel "molecular". Además de la semántica propia del sublenguaje utilizado para especificar procesos (un sublenguaje de CCS [9]), se añade una semántica adicional para distinguir entre procesos de obligación (**operación**) y procesos de prohibición (**protocolo**). Una operación es un servicio de mayor nivel ofrecido por el objeto. Un caso particular de operación es cuando, además, se asume que el proceso actúa como "todo o nada" y se denomina **transacción**. Una operación determina secuencias obligadas de acciones en la vida del objeto, en cambio, un protocolo establece secuencias permitidas.

Cada objeto encapsula su estado y las reglas que rigen su comportamiento. Como es habitual en un entorno orientado a objeto, los objetos pueden ser vistos desde dos perspectivas distintas: estática y dinámica. Desde el punto de vista estático, llamaremos **atributos** al conjunto de propiedades que describen estructuralmente al objeto. Los valores asociados a cada propiedad estructural del objeto determinan el **estado del objeto** en un instante dado. La evolución de los objetos (perspectiva dinámica) viene caracterizada por la noción de **cambio de estado**: la ocurrencia de una acción puede generar cambios en los valores de atributos (definidos por **evaluaciones** y **derivaciones**). La actividad de un objeto está determinada por un conjunto de fórmulas: **precondiciones**, **restricciones de integridad**, **disparos**, **protocolos** y **operaciones**.

Cada objeto tiene un identificador único (**oid**) proporcionado implícitamente por el sistema. Sin embargo, la referencia a un objeto se hace mediante **mecanismos de identificación** pertenecientes al espacio del problema. Una función de identificación establece correspondencias entre los mecanismos de identificación y el *oid* del objeto.

Llamamos **tipo** a la plantilla que describe la estructura y el comportamiento de un objeto. Una **clase** se compone de un nombre de clase, uno o más mecanismos de identificación y un tipo. Una clase compleja es aquella definida utilizando otras clases. Las relaciones entre clases disponibles en $\mathcal{OASIS}$ son **agregación** y **especialización**.

3 Semántica de $\mathcal{OASIS}$

La semántica de $\mathcal{OASIS}$ es dada en términos de una estructura de Kripke (W, τ, ρ). W es el conjunto de todos los mundos[1] posibles que un objeto puede alcanzar. Sea F el conjunto de Fórmulas bien formadas (Fbf) evaluadas sobre el estado (mundo asociado) en el cual se encuentra

[1] De acuerdo con lo dicho, los estados son aserciones (fórmulas), los mundos son estructuras sobre las que dichas fórmulas son interpretadas.

el objeto, $\underline{A}$ el conjunto "*ground*" de acciones de la signatura del objeto y $2^{\underline{A}}$, el conjunto de pasos instanciados posibles. Las funciones τ y ρ se definen como:

$$\tau : F \rightarrow 2^W$$
$$\rho : 2^{\underline{A}} \rightarrow (W \rightarrow W)$$

La función τ asigna a una fórmula en la lógica de estado (Lógica de Predicados de Primer Orden) el conjunto de mundos en los cuales se satisface. La función ρ asigna a cada paso una relación binaria entre mundos. Siendo $\mu \in 2^{\underline{A}}$ un paso y $w, w' \in W$, el significado buscado es: $(w, w') \in \rho(\mu)$ si y sólo si la ocurrencia de μ conduce al objeto desde el mundo w al mundo w'.

En [5] se presentan las fórmulas y especificaciones de proceso utilizadas en cada una de las secciones de una plantilla de clase $\mathcal{OASIS}$ y cómo la plantilla completa de la clase se corresponde con fórmulas en una variante de Lógica Dinámica formalizada en [8].

4 Especificación de clase

Una especificación $\mathcal{OASIS}$ es, esencialmente, un conjunto estructurado de definiciones de clase. Para una clase simple todas las propiedades quedan establecidas en su plantilla. Para una clase compleja, además de las propiedades establecidas por su plantilla, existirán propiedades adicionales determinadas por las relaciones de clase que la definen.

A continuación mostramos un resumen de los aspectos más significativos de $\mathcal{OASIS}$ como lenguaje, recorriendo algunas de las partes que forman la plantilla de una clase $\mathcal{OASIS}$.

4.1 Atributos

Los atributos son propiedades estructurales de las instancias de una clase. Un atributo es un par <nombre, *sort* de evaluación>, siendo nombre el identificador del atributo y *sort* de evaluación el nombre del conjunto soporte (*carrier*) o valores que puede tomar dicho atributo. En $\mathcal{OASIS}$ se distinguen tres variedades de atributos: *constantes*, *variables* y *derivados*. Para cada atributo existen funciones y operaciones disponibles para ser usadas en la especificación de la plantilla, de acuerdo al tipo abstracto de datos asociado al atributo. Además, para atributos multivaluados se puede especificar la representación escogida utilizando los tipos abstractos de datos predefinidos **list**, **set** o **bag**.

4.2 Eventos

Un evento es la abstracción de un cambio de estado atómico e instantáneo que le puede acontecer a un objeto. Los eventos **new** y **destroy** indican que se trata del evento de creación o de destrucción de instancias, respectivamente. Ambos eventos son únicos para cada clase. El evento **new** tiene como parámetros los valores para los atributos constantes y variables del objeto.

Distinguimos entre eventos privados (sin la etiqueta **calling to members** o **sharing with members**), eventos implicados (con la etiqueta **calling to members**) y eventos compartidos (con la etiqueta **sharing with members**). Los primeros son servicios ofrecidos o requeridos por un objeto y cuya ocurrencia depende sólo del comportamiento del objeto en sí mismo. Los eventos implicados y los compartidos corresponden a servicios provistos por un objeto agregado, coordinados con servicios en sus objetos componentes. En el primer caso la ocurrencia del evento en el objeto agregado implica necesariamente la ocurrencia del correspondiente servicio en sus objetos componentes. Cuando se trata de eventos compartidos, dicha implicación es en ambos sentidos.

Ejemplo 1 *La clase* ***préstamo****, definida como agregación entre un objeto de la clase* ***libro*** *y uno de la clase* ***socio****, tiene eventos implicados respecto del evento de creación. Considerando que* ***codigo*** *y* ***numero*** *son los mecanismos de identificación de* ***libro*** *y* ***socio****, respectivamente, en la especificación de la clase* ***préstamo*** *tendremos:*

```
class prestamo
    ...
events
    prestar(Fecha:date) new  calling to members
        libro.ser_prestado(Fecha),
        socio.obtener_libro;
    ...
end class
```

4.3 Evaluaciones

Las evaluaciones son fórmulas en Lógica Dinámica que establecen cómo las acciones afectan el estado del objeto. Estas fórmulas para evaluaciones son del tipo $\psi \rightarrow [a]\phi$, y se interpretan como: "si en un determinado estado del objeto se satisface ψ y ocurre la acción a, en el estado inmediatamente posterior del objeto se satisface ϕ". Es decir, la condición de evaluación ψ se evalúa y se satisface en el estado en el que ocurre la acción y ϕ se satisface en el instante inmediatamente posterior a la ocurrencia de a.

Ejemplo 2 *Una evaluación que establezca una cierta relación entre* ***deposito(Cantidad:nat)*** *y* ***reintegro(Cantidad:nat)*** *respecto del atributo* ***saldo*** *de una cuenta bancaria.*

```
[deposito(Cantidad)]saldo:=saldo+Cantidad;
[reintegro(Cantidad)]saldo:=saldo-Cantidad;
```

4.4 Precondiciones

En algunas ocasiones no es suficiente la iniciativa de un cliente para que una acción ocurra en el servidor sino que además ciertas condiciones deben satisfacerse en dicho servidor. Si la precondición asociada a una acción a no se satisface en el servidor, en lugar de acontecerle la acción a se dice que le acontece la acción $\neg a$.

En el contexto de Lógica Dinámica, las precondiciones tienen la forma $\neg\phi \rightarrow [a]\, false$, donde ϕ es una Fbf interpretada como una condición para la ocurrencia de la acción indicada. El significado es "si ϕ no se cumple, después de la ocurrencia de la acción no existe un estado siguiente válido".

Ejemplo 3 *El evento* ***reintegro*** *en la clase* ***cuenta*** *tiene la siguiente precondición:*

```
reintegro(Cantidad) if {saldo >= Cantidad};
```

4.5 Disparos

Un disparo modela la siguiente situación: si un objeto está en un estado que satisface cierta condición entonces la ocurrencia de una determinada acción está obligada. Si el objeto en el cual se establece la obligación es precisamente el cliente de dicha acción, dicha acción ocurre como una solicitud de servicio desde dicho objeto. Si el objeto en el cual existe la obligación sólo es el servidor de dicha acción, deberá esperar hasta que el cliente de la acción le solicite el servicio de la acción.

Un disparo es modelado por la fórmula $\phi \rightarrow [\neg a] false$, cuyo significado es: "si ϕ se satisface y no se ejecuta la acción indicada, entonces el objeto no alcanza un estado válido. En otras palabras, si se cumple ϕ, la acción debe ocurrir para que el objeto alcance un estado siguiente".

Ejemplo 4 *Cuando el porcentaje de utilización del disco llega al 80%, el objeto de la clase* `servidor` *solicita a los usuarios que liberen espacio de disco.*

```
usuario(everyone)::aviso(fecha,hora,``sin espacio'')
    when {porcentaje_utilizado > 80 and avisado_80_%=false};
```

4.6 Operaciones y protocolos

Existen tres semánticas posibles que pueden ser asociadas en la especificación de un proceso en $\mathcal{OASIS}$:

(a) La semántica del lenguaje utilizado para su especificación. En $\mathcal{OASIS}$, el lenguaje utilizado para especificar procesos es un subconjunto de CCS, compartiendo así una semántica basada en la noción de sistema de transición etiquetado y todas las propiedades formales establecidas en CCS.

(b) La semántica de permisos, es decir, una especificación de proceso establece secuencias de acciones que *pueden* ocurrir.

(c) La semántica de obligaciones, es decir, una especificación de proceso establece secuencias de acciones que *deben* ocurrir.

Un proceso en $\mathcal{OASIS}$ tendrá la semántica (a) junto con la semántica (b) ó (c) dependiendo de si se trata de un protocolo o una operación, respectivamente. Así, la sección *operations* se utiliza para especificar secuencias obligatorias de acciones y la sección *protocols* para especificar secuencias permitidas de acciones. Un caso particular de proceso de obligación es el que actúa como una transacción, es decir, con la política de "todo o nada". En este caso se puede declarar con el calificativo de *transaction* para el proceso. Asumiremos siempre por defecto que el proceso no actúa como transacción.

Con la sintaxis y semántica establecida para las especificaciones de proceso existe un correspondencia directa de ellas con conjuntos de fórmulas en la Lógica Dinámica utilizada. Es decir, cualquier especificación de procesos puede ser interpretada como fórmulas de cambio de estado ($\psi \rightarrow [a]\phi$) junto a fórmulas de prohibición ($\neg\phi \rightarrow [a]\, false$) cuando se trata de un protocolo o junto a fórmulas de obligación ($\phi \rightarrow [\neg a] false$) cuando es una operación.

Ejemplo 5 *Un protocolo en la clase* `máquina`*. Se trata de una máquina de chocolatinas que acumula hasta tres monedas como crédito.*

```
maq:
 maq   =  moneda.MAQ1;
 MAQ1  =  chocolatina.maq + moneda.MAQ2 + ::return_moneda.maq;
 MAQ2  =  chocolatina.MAQ1 + moneda.MAQ3 + ::return_moneda.MAQ1;
 MAQ3  =  chocolatina.MAQ2 + ::return_moneda.MAQ2;
```

5 Clases Complejas

Una clase compleja utiliza en su definición otras clases (simples o complejas). Las relaciones para la definición de clases complejas son agregación y especialización. Independientemente de las relaciones entre clases establecidas, la especificación de una clase se hace separadamente de las relaciones entre clases en las cuales pueda participar. Diremos que una clase compleja tiene propiedades emergentes si tiene propiedades adicionales a las capturadas implícitamente por la relación entre clases que la define (en caso contrario, la especificación explícita de su plantilla de clase es opcional).

5.1 Agregación

Una agregación modela la noción de relaciones estructurales entre objetos. Dado que no puede existir comunicación por *action sharing* o *action calling* fuera de la estructura de agregación, las instancias a quienes un objeto de una clase agregada puede hacer referencia como componentes está restringido a las instancias componentes que efectivamente forman parte del objeto agregado. Este aspecto es remarcado por la etiqueta **members** en la definición de eventos de la clase agregada.

Ejemplo 6 *Consideremos el objeto agregado* `coche` *cuyos componentes son* `motor`, `ruedas` *(debe tener* `4` *ruedas y opcionalmente una de repuesto) y un* `chasis`*. Esto nos lleva a tener la clase* `coche` *como objeto agregado con componentes* `chasis`, `ruedas` *y* `motor`*:*

```
coche aggregation of
static inclusive chasis towards(1,1) from(1,1);
static inclusive motor towards(1,1) from(0,1);
dynamic inclusive rueda towards(4,5) from(0,1)
    list[‘‘adelante izq’’,‘‘adelante der’’,
        ‘‘atras izq’’,‘‘atras der’’,‘‘repuesto’’];
```

Un `chasis` no puede existir si no está asociado a un `coche` y un `coche` tiene exactamente un `chasis`. Por ser **static** `chasis` forma parte del `coche` de manera permanente y el que sea **inclusive** conlleva que cualquier cosa que se quiera hacer con el `chasis` es a través del `coche`. El `motor` puede existir independientemente de su relación con un `coche`. En cuanto a las `ruedas` se refiere, vemos que pueden existir fuera del contexto de un `coche` pero, sólo pueden formar parte de un `coche`. La pertenencia de una `rueda` a un `coche` puede variar con el tiempo (**dynamic**). La palabra reservada **towards** representa la multiplicidad vista desde el objeto agregado hacia los componentes. La palabra **from** representa la multiplicidad vista desde un objeto del componente hacia el agregado.

La especificación **where** permite seleccionar los objetos de la clase del componente que satisfacen la fórmula especificada. La palabra reservada **grouping by** indica que cada instancia del componente (grupo de objetos) se crea (o destruye) automáticamente agrupando los objetos de la clase componente según los valores de dichos atributos en los objetos de la clase del componente.

Para cada componente y atributo definido (con multiplicidad mayor que uno) existirán funciones y operaciones asociadas disponibles para ser utilizadas en las fórmulas de la plantilla. Algunas de estas funciones son **count**, **min**, **sum, position**, **first,** etc.

Ejemplo 7 *El siguiente ejemplo redefine la clase* `coche` *y establece relaciones entre los objetos agregados y sus componentes.*

```
class coche
...
derived attributes
    temperatura_del_coche:int;
    presion_promedio:real;
    gastos_1998:int;
derivations
    temperatura_del_coche:= motor.temperatura;
    presion_promedio:= avg(rueda.presion) where {rueda.instalada=true};
    gastos_1998:= sum(reparacion.importe) where {año=1998};
...
events
    arrancar_coche alias for motor.encender;
    borrar_reparaciones_un_año(ValorAño:nat) calling to members
            reparacion(ref,[ValorAño,_]).eliminar;
    girar_izq calling to members
            rueda[''adelante izq''].girar_izq,
            rueda[''adelante der''].girar_izq;
    girar_der calling to members
            rueda[''adelante izq''].girar_der,
            rueda[''adelante der''].girar_der;
...
end class
```

```
class reparacion
identification
    ref:(año, mes)
constant attributes
    año:nat; mes:nat;
    importe:real;
events
    introducir new;
    eliminar destroy;
...
end class
```

```
coche aggregation of
static inclusive chasis towards(1,1) from(1,1);
static inclusive motor towards(1,1) from(0,1);
dynamic relational reparacion towards(0,*) from(1,1);
dynamic inclusive rueda towards(4,5) from(0,1)
    list[''adelante izq'', ''adelante der'',
        ''atras izq'', ''atras der'',''repuesto''];
```

Ejemplo 8 *Dada la siguiente definición para un atributo:* ***números:nat list[1..10]****, se podrían definir los siguientes atributos derivados usando la signatura implícita para dicho atributo:*

```
derivations
    sin_numeros:= {count(numeros)=0};
    tiene_numeros_grandes:={(count(numeros) where value>100) > 0};
    suma_de_numeros:= sum(numeros);
    maximo_numero:= max(numeros);
    primer_numero:= first(numeros);
```

5.2 Especialización

Mediante especialización las propiedades definidas en las clases pueden ser refinadas. La clase original se denomina superclase, las clases obtenidas por especialización se llaman subclases. En $\mathcal{OASIS}$ existen tres mecanismos de especialización: particiones estáticas, particiones dinámicas y grupos de rol. La especialización incorporada en $\mathcal{OASIS}$ está inspirada en los trabajos de Wieringa [16]. La especialización establece jerarquías de clases. Una clase puede ser superclase de más de una jerarquía a la vez. En $\mathcal{OASIS}$ cada jerarquía de especialización es disjunta, es decir, las instancias de la superclase están asociadas a lo más a una subclase. Además, en el caso de particiones la jerarquía de especialización es total, es decir, cada instancia de la superclase es a la vez instancia de una de las subclases.

5.2.1 Particiones estáticas

Las instancias de las subclases definidas por particiones estáticas están asociadas a una partición a partir de su creación y se mantienen en ella durante toda su existencia.

Ejemplo 9 *Dos particiones estáticas de una misma clase:*

```
camion, coche, otro_vehiculo static specialization of vehiculo;
gasolina, diesel, otro_tipo static specialization of vehiculo;
```

Las propiedades de la superclase y de las subclases indicadas en estas jerarquías se definirán separadamente mediante plantillas de clase (cuando sea necesario).

5.2.2 Particiones dinámicas

Un proceso migratorio es aquel por el cual una instancia pasa de una subclase a otra. Se dispone de dos formas alternativas para especificar el proceso de migración: por la ocurrencia de ciertas acciones en determinados estados del objeto, y por partición de los posibles estados del objeto en función de los valores que presenten sus atributos.

Particiones dinámicas por la ocurrencia de acciones específicas En este caso el proceso migratorio se define mediante una especificación de proceso que utiliza la misma sintaxis de las operaciones o protocolos. Las acciones incluidas en la especificación de proceso pertenecen a alguna de las acciones de la subclase desde donde se migra. las constantes agentes que definen el proceso se corresponden con los nombres de las subclases de la partición. Por defecto, el **new** de las instancias es el servicio en la acción indicada al comienzo del proceso de migración.

Ejemplo 10 *Una especialización dinámica de la clase* `coche` *determinada por la ocurrencia de las acciones* `crear_coche`, `reparar` *y* `estropear` *puede ser:*

```
funcionando, estropeado dynamic specialization of coche
    migration relation is
        coche = crear_coche.funcionando;
        funcionando = estropear.estropeado;
        estropeado = reparar.funcionando;
```

Según lo dicho, la creación de una instancia de `coche` implica comenzar perteneciendo a la partición `funcionando`. Cuando sea una instancia de `funcionando` le ocurrirán acciones de la signatura de `coche` más, posiblemente, otras de la subclase `funcionando`. Entre ellas está la acción `estropear` que pertenece a la signatura de `funcionando`. Su ocurrencia implica salir del proceso que representa a `funcionando` y entrar en el indicado en el proceso migratorio, esto es, la subclase `estropeado`.

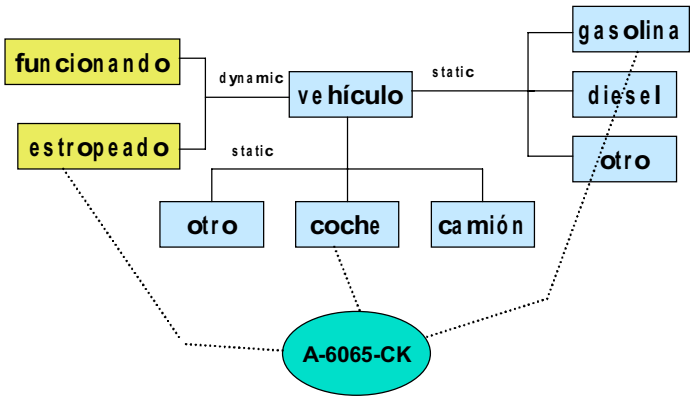


Figura 2: Particiones estáticas y dinámica sobre la clase vehículo.

Partición dinámica en función de valores de atributos En este caso el proceso migratorio queda determinado por el estado de cada objeto. Cada vez que el objeto alcanza un nuevo estado esto puede implicar su migración de una subclase a otra en la partición.

Ejemplo 11 *Una partición dinámica de la clase* `cuenta` *en función del atributo* `saldo` *puede ser:*

```
no_rentable where {saldo<100000},
medio_rentable where {saldo>=100000 and saldo<1000000},
muy_rentable where {saldo>=1000000}
        dynamic specialization of cuenta;
```

En el ejemplo, la subclase de la partición dinámica a la cual pertenece un objeto de la clase cuenta se determina implícitamente a partir del valor del atributo `saldo`.

Tanto en particiones estáticas como dinámicas cada objeto es instancia a la vez de la superclase y de una (y sólo una) de las subclases, es decir, se trata del mismo objeto (mismo *oid*). Esto tiene las siguientes consecuencias semánticas:

- Debe existir compatibilidad de comportamiento [15] viendo al objeto como instancia de la subclase respecto a su visión desde la superclase.
- Cuando el objeto es destruido en la superclase es destruido también en la subclase y viceversa.

5.2.3 Especies

Una especie es una clase obtenida como producto cartesiano entre las subclases de más bajo nivel en la misma jerarquía. Las acciones de creación de instancias en particiones son siempre dirigidas a las especies. Cada especie incluye las propiedades de varias clases por lo que conlleva la herencia múltiple de las propiedades especificadas individualmente. Para aquellas clases especies que incorporen propiedades emergentes (además de las implícitas por la herencia múltiple) se incluirá explícitamente la plantilla de clase correspondiente.

Ejemplo 12 *La clase (especie)* `camión*diesel` *puede tener como propiedad emergente un atributo que represente la fecha del último cambio de filtro de combustible. Para ello, especificaremos la plantilla de la clase* `camión*diesel` *como para el resto de clases.*

La Figura 2 ilustra las particiones que se han definido en los ejemplos anteriores. Además se muestra cómo un objeto (cuyo valor de mecanismo de identificación es la matrícula A-6065-CK) es instancia de una subclase de cada una de las particiones, es decir, en este caso es instancia de la especie `estropeado*coche*gasolina`.

5.2.4 Grupos de Rol

Mediante grupos de roles representamos conductas distintas de un objeto. Una superclase de un grupo de rol puede tiene asociada subclases de rol, cada una representando un patrón de conducta específico para objetos en dicha subclase. El objeto instancia de la superclase se denomina *player*. El objeto *player* es un objeto diferente al objeto de la subclase de rol (tienen distinto *oid*). Esto tiene tres consecuencias semánticas destacables:

- Un objeto *player* puede estar asociado al mismo tiempo a dos o más instancias de una subclase de rol.
- Pueda no existir compatibilidad de comportamiento entre ambos objetos, es decir, existe mayor libertar en cuando al refinamiento especificado en la plantilla de la subclase.
- El objeto *player* puede seguir existiendo cuando las instancias de una subclase de rol asociadas son destruidas.

Ejemplo 13 *Un ejemplo de un grupo de rol definidos sobre la clase* `persona`*.*

```
estudiante towards(0,1) ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

Con la restricción de multiplicidad se ha querido representar que una instancia de la clase `persona` puede desempeñar, simultáneamente, como máximo `10` *roles* de `empleado` o, alternativamente, como máximo un rol de `estudiante`. También es posible que una instancia de `persona` no desempeñe ningún rol.

El evento de creación es enviado a la clase `persona`. Posteriormente, si ocurre el evento `ser_matriculado` (que representa el **new** de `estudiante`), entonces la `persona` pasa a desempeñar un rol de `estudiante`. Si destruimos el objeto de `persona`, automáticamente se destruye el objeto `estudiante`, lo contrario no se cumple.

Ejemplo 14 *Consideremos la partición dinámica de* `persona` *junto a las subclases de rol definidas sobre la misma clase en el ejemplo anterior:*

```
niño where {edad<14},
adolescente where {14<=edad and edad<18},
adulto where {18<=edad}
    dynamic specialization of persona;

estudiante towards(0,1) ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

La especificación de un grupo de rol no supone aumentar el número de especies dado que cada subclase de rol constituye la raíz de una nueva jerarquía (pues se trata de objetos distintos). En cambio, las particiones (estáticas o dinámicas) de una subclase de rol dan lugar a nuevas especies.

6 Conclusiones

Los métodos semiformales usados para la especificación de requisitos funcionales continúan sin satisfacer las expectativas de quienes los utilizan. Aunque se agradece el esfuerzo realizado al integrar las principales propuestas de modelado en UML como notación estándar, no existen aportes significativos respecto del proceso de desarrollo ni de la calidad del modelo conceptual obtenido. $\mathcal{OASIS}$ es un enfoque formal donde cada constructor un significado preciso, los conceptos forman un conjunto reducido y consistente, y se ofrece la capacidad expresiva necesaria para el modelado conceptual. Mediante la utilización de herramientas basadas en $\mathcal{OASIS}$ se obtienen las siguiente mejoras: el disponer de una semántica bien definida evita ambigüedades en la especificación, es posible realizar verificaciones más exhaustivas de acuerdo a las propiedades del modelo y se pueden establecer con mayor precisión patrones de implementación para la generación automática de código. Las líneas de investigación actuales en torno a $\mathcal{OASIS}$ incluyen las siguientes áreas: aspectos metodológicos que definen el proceso de desarrollo de software [10], animación automática de especificaciones para la validación temprana de requisitos [6, 7, 13], evolución de esquemas conceptuales integrada en el mismo marco de $\mathcal{OASIS}$ [1] y generación automática de código en entornos de software industriales [11]. Todos estos esfuerzos están dentro del marco de extensión de la herramienta CASE basada en $\mathcal{OASIS}$.

Referencias

[1] Carsí J.A., Camilleri S., Canós J.H., Ramos I. *Propuesta de Interfaz Gráfica de Usuario homogénea para el Diseño y la Explotación de Sistemas de Información*. Actas de las III Jornadas de Trabajo en Ingeniería del Software, JIS'98, páginas 141-153, Murcia, 1998.

[2] Dubois E., Du Bois P., Petit M. *O-O Requirements Analysis: an agent perspective*. In Proc. of the 7th.European Conference on Object Oriented Programming, ECOOP'93, pages 458-481, 1993.

[3] Feenstra R.B., Wieringa R.J. *LCM 3.0: a languaje for describing conceptual models*. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[4] Jungclaus R., Saake G., Hartmann T., Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems*. ACM Transactions on Information Systems, vol.14, num.2, pages 175-211, 1995.

[5] Letelier P., Ramos I., Sánchez P., Pastor O. $\mathcal{OASIS}$ *3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*. Servicio de Publicaciones de la Universidad Politécnica de Valencia (SPUPV-98.4011), ISBN 84-7721-663-0, Universidad Politécnica de Valencia, 1998. `http://www.dsic.upv.es/users/oom/books.html`.

[6] Letelier P., Sánchez P., Ramos I. *Prototyping a Requirements Specification through an Automatically Generated Concurrent Logic Program*. Gupta (Ed.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, LNCS 1551, pages 31-45, Springer-Verlag, 1998.

[7] Letelier P., Sánchez P., Ramos I.. *Un ambiente para Especificación Incremental y Validación de Modelos Conceptuales*. Actas de las Segundas Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software, IDEAS'99, páginas 216-228, Costa Rica, 1999.

[8] Meyer J.-J.Ch. *A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic.* In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.

[9] Milner R. *Communication and Concurrency.* Prentice Hall Series in Computer Science, C.A.R. Hoare, Series Editor, 1989.

[10] Pastor O., Insfrán E., Pelechano V., Romero J., Merseguer J. *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods.* Conference on Advanced Information Systems Engineering, CAiSE '97, pages 145-158, Barcelona 1997.

[11] Pastor O., Pelechano V., Insfrán E., Gómez J. *From Object-Oriented Conceptual Modeling to Automated Programming in Java.* Proceedings of 17th International Conference on Conceptual Modeling, ER'98, pages 183-196, Singapure, November 1998.

[12] Rational Software Corporation. *UML notation guide. Versión 1.1*, September 1997. `http://www.rational.com/uml`

[13] Sánchez P., Letelier P., Ramos I. *Constructs for prototyping information systems with Object Petri Nets.* Proceedings of the IEEE International Conference on SMC'97, pages 4260-4265, Orlando, 1997.

[14] OBLOG Software S.A. *The OBLOG Software Development Approach (White Paper).* 1999, `http://www.oblog.pt/Download/Documentation.exe`

[15] Wegner P., Zdonik S. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like.* In Proc. of the 2th.European Conference on Object Oriented Programming, ECOOP'88, LNCS 322, pages 55-77, 1988.

[16] Wieringa R., Jonge W., Spruit P. *Using Dynamic Classes and Role Classes to Model Object Migration.* Theory and Practice of Object Systems, vol.1, num.1, pages 61-83, 1995.

Animation of Conceptual Models using two Concurrent Environments: an overview*

Patricio Letelier Pedro Sánchez Isidro Ramos
Department of Information Systems and Computation
Valencia University of Technology, 46020 Valencia
SPAIN
{letelier, ppalma, iramos}@dsic.upv.es
http://www.dsic.upv.es/users/oom

Rafael Corchuelo
Department of Languages and Information Systems
University of Sevilla, 41012 Sevilla
SPAIN
corchu@lsi.us.es

Abstract

Abstract: $\mathcal{OASIS}$ is a formal approach for the specification of object oriented conceptual models. In $\mathcal{OASIS}$ conceptual schemas of information systems are represented as societies of interacting concurrent objects. Animating such models in order to validate the specification of information systems is a topic of interest in requirements engineering. Concurrent Logic Programming and Petri nets are suitable models for distributed computation allowing a natural representation of concurrence. Using Concurrent Logic Programming or Petri Nets, $\mathcal{OASIS}$ specifications are animated according to $\mathcal{OASIS}$ execution model. In this paper, we show our work in translating $\mathcal{OASIS}$ concepts into concurrent environments. This work has been developed in the context of a CASE tool supporting the $\mathcal{OASIS}$ approach. Our aim is to build a module for animation and validation of specifications. A preliminary version of this module is presented.

Key-Words: Formal Specification, Animation of Specifications, Object-Oriented Models.

1 Introduction

Conceptual models, representing the functional requirements of information systems, are a key factor when linking the problem and solution domains. Building a conceptual model is a discovery process, not only for the analyst but also for the stakeholders. The most suitable strategy in this situation is to build the conceptual model in an iterative and incremental way, through analyst and stakeholder interaction. Conceptual modeling involves four activities: elicitation of requirements, modeling or specification, verification of quality and consistency, and eventually, validation.

Formal methods for conceptual modeling provide improvements in soundness and precision for specifications, simplifying their verification. However, when considering elicitation and requirements validation, prototyping techniques are more used. Hence, it is interesting to obtain a combination of both approaches.

This work uses $\mathcal{OASIS}$ [8] (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) as a formal approach for object-oriented conceptual specification of information systems. This is a step forward in a growing research field where validation

*This research is supported by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) through the MENHIR proyect (grant no. TIC97-0593-C05-01).

Letelier, P. Sánchez, P., Ramos, I. Animation of Conceptual Models using two Concurrent Environments: An overview. Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers. 1999.

of formal specifications through animation is being explored [14]. In this sense, some other proposals close in nature to $\mathcal{OASIS}$ are [6] and [5]. The differences, though, between these works and ours are basically determined by features of the underlying formalisms and the offered expressiveness. According to the presented results, the state of art is similar and is characterized by preliminary versions of animation environments.

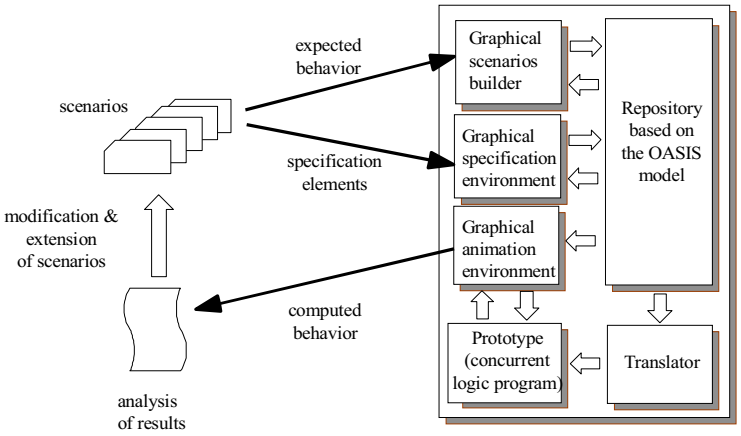


Figure 1: A framework for incremental specification of requirements

Fig.1 shows our framework for elicitation, modeling, verification and validation of requirements. Elicitation is achieved by using scenarios [12]. The expected behavior and elements of a given specification are extracted by the analyst from scenarios. The graphical scenario builder helps define scenarios in a suitable way. Functional requirements are modeled using a graphical specification module based on $\mathcal{OASIS}$. Conceptual models can be verified according to $\mathcal{OASIS}$ formal properties. At each stage of the requirements specification process it would be possible to validate the behavior of the associated prototype against the expected behavior. This comparison could lead to updates or extensions of existing scenarios. This cycle continues until the requirements are compliant with the proposed set of scenarios.

Experiments have been carried out using Object-Oriented Petri Nets [13] and Concurrent Logic Programming [9] as semantic domains for $\mathcal{OASIS}$ specifications. Correspondences between $\mathcal{OASIS}$ and these environments have been implemented in a translator program. The translator takes an $\mathcal{OASIS}$ specification stored in the repository and generates automatically a Concurrent Logic Program or a Petri Net that constitutes a prototype for the corresponding conceptual model. Furthermore, through a preliminary version of the graphical animation environment, the analyst can interact with the prototype in a suitable way. Regarding Concurrent Logic Programming, we have worked with the concurrent logic languages Parlog [2] and KL1 [1]. Regarding Petri Nets we have worked with Object-Oriented Timed Petri Nets [3] and Object Petri Nets [7].

2 Why a concurrence model?

Considering the utility of this work in requirements validation of $\mathcal{OASIS}$ specifications, it is essential to obtain the nearest implementation solution to the $\mathcal{OASIS}$ semantics. If so, we will have a way to provide formalization for the translation process. In this way, we think the best choice is to give each object an execution thread.

Fig.2 shows four ways of implementing concurrent object systems. The horizontal axis goes from purely sequential to purely concurrent. The vertical axis goes from a no object-oriented paradigm to a completely one. In three of these ways, a *monitor* should be necessary to control the activity of the system. The first (sequential, -OO) is

the simplest: there is only one monitor and there are objects whose state can be represented by a relational database, for instance. The second (concurrent, -OO) gains in concurrence by increasing the number of active processes. However there is no object representation. The third way considers each object as an independent unit of execution but there is a quantum of running time for each object. The fourth, which we chose in our model, is fully concurrent and each object is considered as an independent unit of execution. Concurrent Logic Programming and Petri Nets allow representing directly systems of this kind.

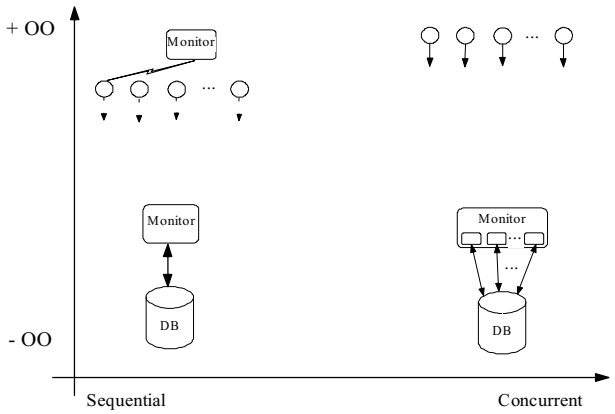


Figure 2: Implementation of concurrent object systems.

3 *OASIS*

OASIS is a formal object-oriented language for the specification of information systems at a high level of abstraction. An *OASIS* specification is a presentation of a theory in the used formal system and is expressed as a structured set of class definitions. Classes can be simple or complex. A complex class is defined in terms of other classes (simple or complex). Complex classes are defined by establishing relationships among classes. These relationships express aggregation or inheritance. A class has a name, one or more identification mechanisms for its instances (objects) and a type or template that is shared by every instance belonging to the class. The state of the object in a given instant will depend on the actions occurred in its lifetime until the moment during which it is observed. In this way, an object is seen as an observable process. Thus, an object has a life characterized by the occurrence of actions (that could be either requested or provided). In this way, an object is seen as client or server depending on whether the object is requesting or providing services.

Services provided in an atomic level by an object are referred as events. Each object has a creation event (starting its life) and optionally a destruction event (finishing its life). Events can be organized in a molecular level as processes. Beside the own semantics used when specifying processes, we add another one in order to make a distinction between prohibition (**protocol**) and obligation process (**operation**). An obligation process is a service offered by an object in a higher level. Particularly, a transaction is an obligation process having the implicit all or nothing policy. A prohibition process does not allow the execution of some sequences of actions in an object life. In *OASIS*, an information system is regarded as a society of autonomous and concurrent objects interacting with each other by the occurrence of actions. An action is a tuple including the client, the server and the requested service.

The visibility between objects is determined by a interfacing mechanism. Every object encapsulates its own state and behavior rules. As usual in object-oriented environments, objects can be seen from two points of view: static and dynamic. From the static perspective, the attributes are the set of properties describing the object structure.

The object state in a definite instant is the set of structural properties values. From the dynamic perspective, the evolution of objects is characterized by the "change of state" notion. The occurrence of actions implies changes (by means of **valuations** and **derivations**) in the values of the attributes. Object activity is determined by a set of rules: **preconditions**, **integrity constraints**, **triggers**, **protocols** and **operations**. A step is the set of actions executed at the same instant by the object. Every object has an unique identifier (*oid*) set by the system. Objects are referred by one or more identification mechanisms belonging to the problem space. An identification function sets a mapping between the identification mechanisms and the oid. The type or template describes the structure and behavior of every object.

$\mathcal{OASIS}$ is a formal specification language that allow defining conceptual schemas according to the object model that has been briefly presented.

3.1 $\mathcal{OASIS}$ semantics

The semantics of $\mathcal{OASIS}$ is defined by means of *Kripke* structures (W, τ, ρ) which include an universe of states w. To each class we associate an accessibility relation ρ in such a way that a pair of states (s, t) is in that relation if and only if there is a transition going from the state s to the state t. Let us call W the set of all possible worlds reachable by an object. Let us call F the set of state well formed formulae (wff) being evaluated over the current state of the object. Let A be the set of ground actions of the object template and $2^{\underline{A}}$ the set of all possible instantiated steps. The functions τ and ρ are defined as follows:

$$\tau : F \rightarrow 2^W$$
$$\rho : 2^{\underline{A}} \rightarrow (W \rightarrow W)$$

The function τ says in which worlds a state formula (in First Order Predicate Logic) is true. The function ρ is a binary relation between worlds (declarative semantics of the language). Given a step $\mu \in 2^{\underline{A}}$, $w, w' \in W$ then $(w, w') \in \rho(\mu)$ if and only if occurring μ implies the transition between the worlds w and w'.

3.2 $\mathcal{OASIS}$ expressed in DL

In [10] Deontic Logic is described as a variant of Dynamic Logic (DL) [4]. The definition of Deontic operators in Dynamic Logic is:

$\psi \rightarrow [a]false$	"the occurrence of a is forbidden in states where ψ is satisfied".
$\psi \rightarrow [\neg a]false$	"the occurrence of a is obligatory in states where ψ is satisfied".
$\psi \rightarrow [a]\phi$	"in states where ψ is satisfied, immediately after a occurrence, ϕ must be satisfied"

where ψ is a well formed formulae that characterizes an object state when the action a occurs and $\neg a$ represents the non-occurrence of the action a (i.e., only other actions different from a could occur). Furthermore, there is no state satisfying the atom $false$. This represents a state of system violation. Thus, one action is forbidden if its occurrence leads the system towards a violation state and one action is obligatory if its non-occurrence leads the system towards a violation state. The $\mathcal{OASIS}$ template is mapped to the formulae previously presented.

These formulae constitute a sublanguage of the language proposed and formalized in [15]. In [8] $\mathcal{OASIS}$ is presented as a specification language with a well defined syntax. Here is an example of part of a simple vending machine system using the $\mathcal{OASIS}$ syntax.

```
conceptual schema vending_machine_system
class vending_machine
identification
    number: (number);
constant attributes
    number: nat.
variable attributes
    n_chocs: nat;
    credit: nat(0);
    switch_on: bool(false);
events
    set new;
    coin_in;
    coin_out;
    choc;
    light_empty;
operations
    CANCEL:
      CANCEL1= {credit>1}::coin_out.CANCEL1
               + {credit<=1}::coin_out;
triggers
    ::light_empty when
        {nchocs= 0 and switch_on= false};
valuations
    [coin_in] credit=credit+1;
```

```
    [::coin_out] credit=credit-1;
    [choc] nchocs=nchocs-1, credit=credit-1;
    [::light_empty] switch_on=true.
preconditions
    choc if {credit>0 and nchocs>0};
    CANCEL if {credit>0};
protocols
    GETCHOC:
        GETCHOC1= coin_in.GETCHOC2;
        GETCHOC2= choc.GETCHOC1 +
                  coin_in.GETCHOC3 +
                  ::coin_out.GETCHOC1;
        GETCHOC3= choc.GETCHOC2 +
                  coin_in.GETCHOC4 +
                  ::coin_out.GETCHOC2;
        GETCHOC4= choc.GETCHOC3 +
                  ::coin_out.GETCHOC3;
end class

class customer
identification
    name:(name);
constant attributes
    name:string;
events
    add new;
    remove destroy;
end class


interface customer(someone)
        with vending_machine (someone)
    services(coin_in,choc);
end interface

interface vending_machine(someone) with self
    services(light_empty, coin_out);
end interface
end conceptual schema
```

In this example there are two classes: `customer` and `vending_machine`. The objects in both classes are active. A `vending_machine` object is forced to self-trigger an action with the event `light_empty` whenever the trigger condition is satisfied. Also, the `vending_machine` object has to return all coins when `cancel` button is pressed. Although `customer` objects do not have explicit triggers, they have an interface with `vending_machine` objects and this allows them to require actions associated with the visible events. Thus `customer` objects are active objects as well.

4 Concurrent Logic Programming and $\mathcal{OASIS}$

Concurrent logic languages arise as an attempt to improve the efficiency of logic languages by exploiting stream *AND* parallelism. Besides, they are high level programming languages and convenient for parallel and distributed systems.

We are interested in modeling objects as perpetual processes according to the $\mathcal{OASIS}$ execution model. The identity of the object is the name of an input stream argument of the process. Works in this direction are principally based on making OO extensions to concurrent logic languages. Although implementation aspects are common, our motivation is different, we want to generate automatically a concurrent logic program corresponding to an $\mathcal{OASIS}$ conceptual model.

It is said that an object is implemented as a perpetual process because among the subgoals in which an object is reduced the same object appears. This produces the effect of continuity in the object life. Whenever the object goal is thrown as subgoal, in the reduction some of its attributes may be modified. Thus a change of state due to the occurrence of the associate action is represented. The effect "to execute an action" is obtained in the reduction when the new input channel is used as the original one without considering the last executed action. The formulae that define the change of state when the event is executed are subgoals that assign new values to the attributes of the object. In [9] a set of mappings between $\mathcal{OASIS}$ and KL1 has been established.

Following theses mappings, a translator from $\mathcal{OASIS}$ to KL1 has been implemented. This translator takes as input a system specification from an $\mathcal{OASIS}$ repository and produces a KL1 program that is compiled in order to obtain the prototype. The translator and the prototype are programs running in a Unix workstation. The interface has been implemented in Tcl/Tk using the Tcl plug-in for Netscape.

In $\mathcal{OASIS}$ the system behavior is determined by the behavior of its objects. An object behavior can be observed by analyzing the actions occurred and the states reached by the object. Thus, the animation of an $\mathcal{OASIS}$ specification allows examining actions and states of the objects.



Figure 3: An animation session.

Fig.3 shows the interface of a prototype of our animation environment. The object society is drawn in the upper left area. Objects are selected by clicking on them. On the right the traces of actions of an object (or object group) are listed. The list of traces can be filtered according to the kind of actions (sent, received, in conflict, executed or rejected). In the state area the state of the object is presented (only when one object is selected). Buttons *play*, *pause*, *stop*, *forward* and *review* are provided in order to control the session of animation. When the animation is paused it is possible to explore the traces of actions and states at previous instants. Eventually, the two entry widgets allow building an external action sent by the analyst in representation of one object in the system.

5 Petri Nets and $\mathcal{OASIS}$

"Petri Nets are a graphical and mathematical modeling tool applicable to many systems. They are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic" [11]. Due to the mathematical background of Petri Nets, all aspects related to formalization are more feasible. Also, analysis techniques can be applied to get properties of liveliness, etc. The enhancements of Object Oriented Petri Nets (OOPN) include for instance token values to be identifiers, inheritance, test and inhibitor arcs, etc. Tokens can encapsulate the activity determined by another Petri Net. OOPN have a single hierarchy that includes both token types and subnet types, thereby allowing multiple levels of activity in the net. Furthermore, OOPN provides functions to evaluate the state of the net without changing its state. The possible use of superplaces and supertransitions makes OOPN suitable to model both synchronous and asynchronous interactions between objects. Each instance of an OOPN class could be an independent object passing on messages between objects through tokens. The notion of OOPN superplace permits synchronous interaction. One transition deposits (or extracts) a token in (from) the superplace. This operation is synchronous with an internal transition of the superplace that accepts (or produces) the token.

We have experimented using the CodeSign tool [3] in order to implement and animate $\mathcal{OASIS}$ specifications by implementing its execution model. How the main features of the $\mathcal{OASIS}$ can be naturally and directly represented in OOPN has been analyzed [13]. Object Oriented concepts (such as classes, instances, interaction between objects, encapsulation, etc.) and spe-

cific $\mathcal{OASIS}$ concepts (such as preconditions, valuations, triggers, protocols, and operations) have been addressed by using the properties of the OOPN.

The support of multiple levels of activity in the OOPN makes the design of architectures easy enough to prototype a society of concurrent objects. Now we are working to obtain CodeSign code automatically from an $\mathcal{OASIS}$ specification.

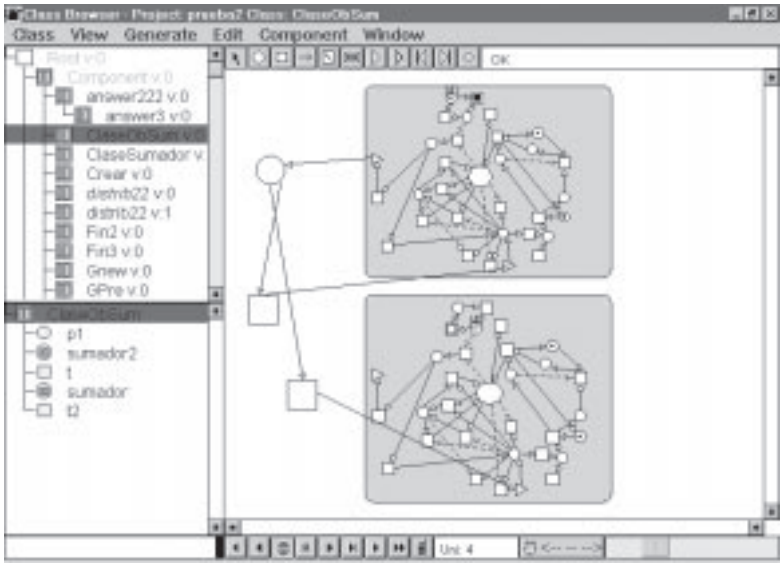


Figure 4: CodeSign Environment.

Fig.4 shows two $\mathcal{OASIS}$ classes (the grey boxes on the right) implemented in CodeSign using the Petri Net model. The bigger place in each class is the container of the instances of that class (notice that every object will have its own Petri Net instantiated.). The rest of the net is needed to direct the arrived actions towards the server object. The external net fragment is needed to establish the communication between the instances of both classes.

6 Conclusions

The main features of the $\mathcal{OASIS}$ language can be more naturally and directly represented in a concurrent environment. Using the execution model of $\mathcal{OASIS}$ as a guide we can obtain a useful animation of the $\mathcal{OASIS}$ specification. Our animation is only applied to purposes of requirements validation and do not claim to be the final software product.

We have built a translator program to obtain automatically both a concurrent logic program and a OOPN from $\mathcal{OASIS}$ specifications using the established correspondences. This work is being integrated into a CASE tool for system modeling supporting the $\mathcal{OASIS}$ approach. We have addressed the object-oriented concepts, and although we have focused on $\mathcal{OASIS}$, it could be extended to other similar languages.

The fidelity of the obtained concurrent implementation in relation to the $\mathcal{OASIS}$ system specification is a matter that is still being studied. In this case the verification and demonstration tasks would be supported by three important factors: firstly at the conceptual level the model is described in a formal language, secondly the abstract execution model is inspired by the semantics of that formal language, and eventually, concurrent programming languages and Petri Nets have a formal base. These factors do not determine the required justification for the translation process but give a way of formalization on which we are working.

We are currently considering the incorporation

of advanced concepts such as complex classes and complex communication mechanism.

References

[1] T. Chikayama. KLIC User's Manual. Institute for New Generation Computer Technology, Tokyo JAPAN, 1995.

[2] T. Conlon. Programming in PARLOG. Addisson-Wesley, 1989.

[3] R. Esser. An Object Oriented Petri Nets Approach to Embedded System Design, PhD Thesis, Swiss Federal Institute of Technology, Zurich, 1996.

[4] D. Harel. Dynamic Logic. In Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pages 497-694. Reidel 1984.

[5] P. Heymans. The Albert II Specification Animator. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm.

[6] R. Herzig and M. Gogolla. An animator for the object specification language TROLL light. In Proc. Colloq. on Object-Orientation in Databases and Software Engineering, Montreal 1994.

[7] C. Lakos, From Coloured Petri Nets to Object Petri Nets, Proceedings of the 16th International Conference on the Application and Theory of Petri Nets, LNCS 935, Torino, Italy, Springer-Verlag, 1995.

[8] P. Letelier, I. Ramos, P. Sánchez and O. Pastor. OASIS 3.0: A Formal Approach to Object-Oriented Conceptual Modeling. SPUPV-98.4011, Servicio de Publicaciones Universidad Politécnica de Valencia, 1998. (in spanish).

[9] P. Letelier, P. Sánchez , I. Ramos. G. Prototyping a requirements specification through an automatically generated concurrent logic program. Gupta (Ed.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science LNCS 1551, pp. 31-45, Springer-Verlag, 1998.

[10] J.-J.Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.

[11] T. Murata. Petri Nets: Properties, Analysis and Applications. Proceeding of the IEEE, Vol. 77, n°4, April 1989.

[12] C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyté, A. Sutcliffe, N.A.M. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois and P. Heymans. A Proposal for a Scenario Classification Framework, Technical Report CREWS 96-01, http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm.

[13] P. Sánchez, P. Letelier and I. Ramos. Constructs for Prototyping Information Systems using Object Petri Nets, Proc. of IEEE International Conference on System Man and Cybernetics, pages 4260-4265, Orlando, USA, 1997.

[14] J. Siddiqi, I.C. Morrey, C.R. Roast and M.B. Ozcan. Towards quality requirements via animated formal specifications. Annals of Software Engineering, n.3, 1997.

[15] Wieringa R.J. and Meyer J.-J.Ch. Actors, Actions and Initiative in Normative System Specification, Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

La evolución de modelos en el desarrollo de software: los enfoques OASIS y MEDES[1]

J. Parets[1], J. Carsí[2], J.H. Canós[2], A.Anaya[1], M.V. Hurtado[1], Penades, M.C[2], P.Paderewski[1], I. Ramos[2], M.J. Rodríguez[1]

[1] Departamento de Lenguajes y Sistemas Informáticos. ETS Ingeniería Informática. Universidad de Granada. Avda. Andalucía, 38 - 18071 - GRANADA. Telf: 958 243 180.
jparets@ugr.es

[2] Departamento de Sistemas Informáticos y Computación. Univ. Politécnica de Valencia
Camino de Vera s/n - 46020 - Valencia. Telf: 96 387 7734. Fax: + 34 96 387 7359
pcarsi@dsic.upv.es

Resumen. Dentro del proyecto MENHIR, los grupos de las Universidades Politécnica de Valencia y de Granada compartimos una preocupación especial por el modelado de la evolución del software. En este trabajo presentamos un estudio comparativo de nuestros enfoques que permita, en un futuro inmediato, disponer de un marco común de intercambio. Los resultados de este estudio muestran que ambas visiones de la evolución del software son complementarias y que hay distimtas posibilidades de sintonización que deben estudiarse en profundidad. **Palabras clave:** evolución del software, Ingeniería de Requisitos, métodos formales, prototipado.

1 Introducción

La evolución es un problema crucial en el desarrollo de software, que lleva estudiándose desde hace casi dos décadas y para el cual se han propuesto diversas soluciones. Una revisión de la literatura existente sobre el tema muestra que existen muy diferentes visiones, tanto en la propia definición del concepto como en los enfoques propuestos para abordar los aspectos evolutivos. En el contexto del proyecto MENHIR, hemos abordado la incorporación de la evolución en nuestros enfoques metodológicos: OASIS (en la UPV) y MEDES (en la UGR). En este trabajo común se persiguen dos objetivos fundamentales:

1. Clarificar las diferentes formas de abordar la evolución presentes en la literatura, de forma que sea posible encuadrar nuestra perspectiva del proceso evolutivo de software.
2. Realizar un estudio comparativo de los conceptos utilizados en OASIS y MEDES con la idea de proponer, en un futuro inmediato, un marco global común que permita la comunicación y el intercambio entre ellos.

[1] Este trabajo ha sido subvencionado por la CICYT a través de los subproyectos ESPILL y MEDES que se engloban dentro del proyecto coordinado MENHIR (TIC-97-0593-C05).

Parets, J., Carsí, J., Canós, J.H., Anaya, A., Hurtado, M.V., Penadés, M.C., Paderewski, P., Ramos, I., Rodriguez, M.J., La evolución de modelos en el desarrollo de software: los enfoques MEDES y OASIS. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos. 1999.

Para ello, en primer lugar presentaremos una clasificación de las tendencias en evolución tratando de sistematizar las diferentes formas de concebirla. A continuación resumiremos los enfoques de MEDES y OASIS, insistiendo en que ambos comparten una visión similar con matices susceptibles de encontrar aspectos compatibles. Finalmente, propondremos un marco común que permita guiar futuras investigaciones.

2 Tendencias de investigación en evolución del Software

En trabajos previos [32,33] hemos señalado que los métodos comerciales de desarrollo de software no consideran explícitamente el proceso evolutivo del software y, por tanto, no incluyen mecanismos ni conceptos que permitan abordarlo desde los propios métodos. Sin embargo, el problema de la evolución ha sido estudiado por muy diferentes autores bajo perspectivas distintas. Considerando los objetivos que se plantean estos autores podemos identificar seis tendencias:

2.1 Modelado de la dinámica del software

El trabajo pionero de Belady y Lehman [5] considera la evolución de software como "dinámica de la evolución de programas", identificando esta evolución con los continuos cambios sufridos por los programas, cambios que eran contemplados como fuente de problemas más que como una necesidad de adaptación de los programas al entorno. Posteriormente, Lehman [21] trata de establecer leyes cuantitativas sobre el comportamiento de un sistema software y su proceso de desarrollo. Estos trabajos se desarrollaron en la línea clásica de la dinámica de sistemas tratando de descubrir los parámetros de esta evolución concebida como dinámica parametrizable.

2.2 Modelado del proceso de desarrollo de software

Diferentes autores investigan sobre modelos de producción de software (modelos de ciclo de vida) durante la década de los 80. Estos modelos de ciclo de vida tratan de incorporar la iteración en el desarrollo. El sistema software se desarrolla haciendo uso de iteraciones, cada una de las cuales lleva asociada tareas de especificación, diseño, codificación o prueba. Dentro de este grupo de modelos cabe destacar el modelo de prototipos [1], el modelo en espiral de Boehm [7,8,9] y el modelo de Henderson [16].
La elaboración de este tipo de modelos, tratando de abandonar la pretendida perfección de un proceso secuencial, e incorporando la posibilidad de retroalimentación, ha servido de base para que se plantee la necesidad de representación, a ser posible automatizable, del proceso de desarrollo: el "software process" [44]. De los trabajos desarrollados en esta línea cabe destacar: el sistema SySL (System Structure Language) de Sommerville y Thomson [40], el modelo PRIMS de cambios [25,26] que recoge los cambios en el personal, política, leyes, recursos, procesos, resultados y sus consecuencias en el entorno de desarrollo, el modelo SPADE de evolución del proceso del software [3] y el marco conceptual propuesto en [14].

Otra línea de trabajo, en este mismo sentido, se centra en la gestión de la configuración y control de versiones [36, 24, 35].

2.3 Modelado de la dinámica del proceso de concepción-desarrollo de software

Desde esta perspectiva se concibe un sistema software como dotado de una estructura determinada que pasa por diferentes *estadios* de maduración. La evolución implica la transición desde un estadio a otro mediante la actividad del equipo de desarrollo. El estudio de la evolución implica, pues, la elaboración de modelos de estas transiciones estructurales.
En este campo cabe destacar los trabajos realizados por autores relacionados con el paradigma orientado a objetos. Banerjee [4] y Casais [12] consideran la evolución de una estructura de clases, las operaciones que modifican esa estructura, las restricciones a la hora de realizar esas modificaciones y los efectos de esas modificaciones en la estructura de clases y en las instancias existentes de esas clases. En el trabajo desarrollado por [22,23] dentro del proyecto DEMETER, se introduce la *historia de evolución de una estructura*, historia que describe una secuencia de fases de desarrollo y la noción de *programa adaptativo* que es capaz de amoldarse a ciertos cambios en la estructura de clases.

2.4 Transformación de programas

"*La mayor parte de la investigación en transformación asume que el programa transformado P' debe cumplir los requisitos del programa original P. Estas transformaciones se denominan transformaciones preservadoras del significado".* De esta forma definen Berzins et al. [6, p. 437] la forma más difundida de transformación de programas tal como la conciben, por ejemplo, Kozaczynski et al. [19], Merks et al. [28]). Sin embargo, Berzins y sus colaboradores consideran "*un enfoque transformacional en la construcción de software que no asume una especificación inicial fija y completamente correcta*" [p.437]. Su objetivo es "*crear modelos conceptuales y herramientas software que permitan la generación automática de variaciones en un sistema software de forma que el hombre sólo se deba ocuparse de las decisiones de alto nivel que deben cambiar de una versión a otra*" basándose en el estudio de transformaciones con cambio de significado. Una visión similar del proceso evolutivo es la que presenta Johnson et al. [17]. Aunque este enfoque podría considerarse similar al anterior, lo hemos separado por su mayor énfasis en la programación que en el resto del proceso de desarrollo.

2.5 Modelado de la dinámica en sistemas de información: modelado dinámico.

Tardieu [42] define los principios básicos del modelado dinámico:
"El modelado dinámico consiste en capturar los invariantes del sistema de información que no se expresan en el modelado estático [...] por oposición a la cinemática, no se concentra en las transiciones de estados, pero considera las causas de la transición [...] por oposición a la evolución, no cambia la estructura del sistema de información. Cualquier cambio es reversible. El beneficio del modelado dinámico resi-

de en el hecho de que describe los productos y los procesos, lo que conduce a una mejor comprensión de la organización modelada." [p.4]. Ejemplos típicos de esta tendencia son los diagramas de transición de estados utilizados en OMT [39] o los modelos de tratamiento característicos de MERISE [43] que, en definitiva, tratan de modelar la sucesión de los diferentes estados por los que circula un sistema y las acciones que producen esta circulación por los diferentes estados.

2.6 Otros enfoques

El término evolución se utiliza en otros contextos con significados adicionales. De hecho, es posible identificar otras perspectivas:

- La *incremental*: concibe el software como una pila que se puede incrementar por un proceso de adición simple, como es el caso de algunos enfoques formales que consideran las propiedades del software como una lista que puede ser fácilmente incrementada [27].
- La *simulativa*: incluye técnicas que, bajo la denominación de *algoritmos genéticos,* simulan la evolución de poblaciones cromosómicas bajo un modelo darwiniano. [37, 41].

Desde nuestro punto de vista, estos enfoques no resultan interesantes en el estudio de la evolución del software: el primero porque olvida los problemas de interacción y emergencia de propiedades entre los componentes de un sistema, y el segundo porque el proceso de evolución del software no es un proceso azaroso, sino dirigido por un equipo de desarrollo que trabaja bajo la presión del entorno.

3 Los enfoques MEDES y OASIS

Tanto MEDES como OASIS enfatizan la evolución de los modelos producidos durante el proceso de concepción y desarrollo del software. En este sentido, pueden incluirse en el apartado 4 del punto anterior: el equipo de desarrollo produce modelos del sistema software que, a lo largo del tiempo, irán refinándose. Cada uno de estos refinamientos implica un estadio en la estructura del sistema. La evolución se encontrará, pues, con dos problemas fundamentales a abordar:

1. La manipulación y gestión de estos cambios de un estadio a otro manteniendo la integridad y la consistencia de los modelos propuestos.
2. La propagación del cambio a las estructuras ya fijadas y en funcionamiento (en general, los datos manejados por el usuario).

En este apartado discutiremos cómo ambos enfoques abordan estos problemas. En general, podemos señalar que MEDES trata de plantearse la evolución desde una perspectiva más abstracta, en tanto que OASIS plantea la evolución en el marco de la orientación a objetos. Por ello presentaremos primero la evolución en MEDES y, posteriormente, en OASIS.

3.1 La evolución de modelos en MEDES

El punto de vista adoptado en MEDES es el de la Teoría del Sistema General [20]: en un sistema distinguimos entre *función* (actividad) y *estructura* (forma estable) partiendo del concepto de *procesador* que es capaz de desarrollar acciones sobre el entorno. En este contexto la noción de *historia* es fundamental. Se define la historia como *la trayectoria de la función y la estructura del sistema en el tiempo.* La representación concreta de esta trayectoria dependerá de la conceptualización que se haga de las nociones de función y estructura y, en el caso de los sistemas software, su evolución vendrá determinada por cambios de estructura realizados por un sistema externo, el *Sistema de Desarrollo* (o, si se prefiere, el equipo de desarrollo).
La función se representa a partir de los conceptos de *acción* y *evento*. Una acción es la posibilidad de realización de una actividad por parte del sistema, en tanto que un evento es la constancia memorizada de que una acción ha sido realizada. En tanto que la acción es *potencia* (en el sentido aristotélico), el *evento* es acto y la realización de un evento supone un cambio de estado en el sistema.
La estructura es una forma lo suficientemente estable como para ser reconocida por el Sistema de Desarrollo [20] y puede definirse como los "*patrones de comportamiento y decisión en un instante de tiempo. Patrones lo suficientemente estables como para ser modelados y para funcionar durante un período dado de tiempo"* [33].
Si asumimos que, desde un punto de vista funcional, el Sistema Software puede realizar acciones, desde un punto de vista estructural el Sistema Software puede sufrir cambios que alterarán su capacidad funcional. El realizar estos cambios estructurales será un privilegio del Sistema de Desarrollo y, por tanto, podemos representarlos como acciones que éste puede realizar y cuya realización se convertirá en evento memorizado.
La distinción entre estructura y función y su representación mediante eventos memorizados permite introducir dos conceptos importantes:

- *Historia Funcional del Sistema* (HFS): memoria de los eventos en un sistema a lo largo del tiempo. Representa la función del sistema.
- *Historia Estructural del Sistema* (HES): memoria de los eventos realizados por el Sistema de Desarrollo para transformar el sistema. Representa la evolución del sistema.

Desde un punto de vista abstracto, el proceso de desarrollo puede ser concebido tal como se muestra en la figura 1. La vida del sistema puede verse como una sucesión de períodos de estabilidad estructural, durante los cuales no se producen cambios en su estructura; en esos períodos el sistema está en funcionamiento. Cuando se produce algún cambio, el sistema pasa a un nuevo período de estabilidad. En los términos habituales en Ingeniería del Software cada período de estabilidad equivale a una *versión* del sistema.
Para representar y dar soporte a la evolución se introduce un sistema especial: el *Metasistema*. El Metasistema es utilizado por el Sistema de Desarrollo como una herramienta CASE y será el responsable de ejecutar las acciones que implican un cambio de estructura en el Sistema, es decir, podrá crear el Sistema y añadirle la funcionalidad requerida: nuevos procesadores, acciones y precondiciones para estas acciones. Además permitirá modificar estas características siguiendo ciertas reglas de integridad en las transformaciones.

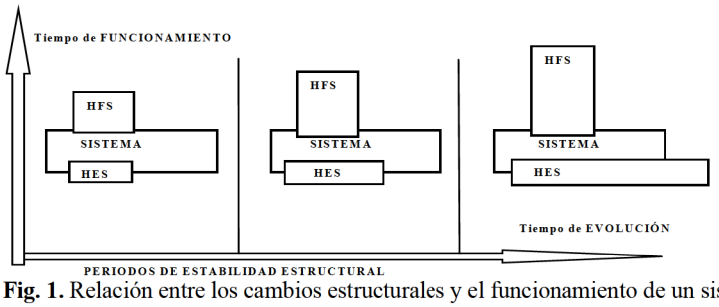


Fig. 1. Relación entre los cambios estructurales y el funcionamiento de un sistema.

La enorme ventaja de introducir el Metasistema reside en que los cambios de *estructura en el Sistema* serán la *funcionalidad normal del Metasistema*. Esto permite que la estructura general de Sistema y Metasistema sean isomorfas. Las actividades de evolución se realizan a través de una *Interfaz de Evolución*, siendo el *Subsistema Genético* (SG) el agente final de estas acciones, cuyos eventos resultantes se memorizan en la HES. La figura 2 representa la relación entre Sistema y Metasistema. El Metasistema es capaz de actualizar el Sistema a través de la *interfaz de evolución*. Puesto que éste también es un sistema, tendrá sus correspondientes historias: HFM (Historia Funcional del Metasistema) y HEM (Historia Estructural del Metasistema).

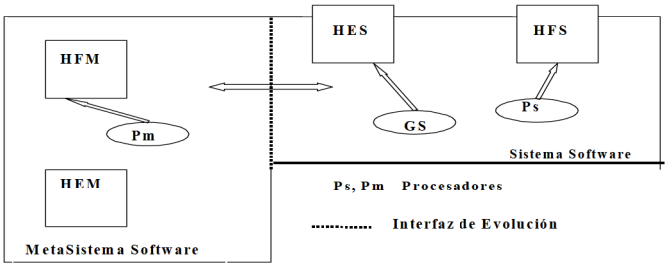


Fig. 2. Interacción entre Sistema y Metasistema.

La consecuencia más importante de esta estructura homogénea e isomorfa para Sistema y Metasistema es que la actividad funcional y la actividad estructural (que implica evolución) se pueden representar utilizando los mismos instrumentos. En el caso de MEDES esta representación se realiza del siguiente modo:

- Historias: Son estructuras temporales [15] que se pueden consultar y actualizar.
- Procesadores: son los agentes que realizan las acciones.
- Acciones: actividad implementada en un lenguaje de programación.
- Precondiciones: determinan cuándo se puede realizar una acción. En nuestro caso están escritas e implementadas en un lenguaje basado en Lógica Temporal de Pasado de primer orden y se evalúan sobre las historias mediante un motor temporal [2].
- Como consecuencia del isomorfismo entre Sistema y Metasistema las condiciones para la evolución del Sistema se expresan como precondiciones sobre las

acciones estructurales realizadas por el Metasistema, es decir, las restricciones de integridad en las transformaciones estructurales se expresan del mismo modo que las precondiciones de la actividad del sistema [38]

3.2 La evolución de modelos en OASIS

En el modelo OASIS [34], los sistemas son vistos como sociedades de *objetos* que interactúan; el *estado* de un objeto viene caracterizado en cada momento por el valor de sus *atributos*, y puede cambiar como consecuencia de la reacción del objeto a estímulos externos o internos llamados *eventos*. Cuando un evento ocurre en la vida de un objeto, éste reacciona de acuerdo a un patrón de *comportamiento* definido mediante un conjunto de reglas: para cada evento, una *precondición* define el subconjunto de estados del objeto en los que éste debe reaccionar; la reacción consiste en un cambio de estado definido en las *reglas de evaluación*, que indican cómo cambia el valor de los atributos, y las *restricciones de integridad*, que definen los estados válidos del objeto. Una vez el objeto ha cambiado a un estado válido, las *reglas de disparo* definen, mediante el patrón condición-acción, el comportamiento activo del objeto.

Objetos de propiedades similares se agrupan en *clases*, de las que aquéllos son *ejemplares*. La complejidad de los sistemas reales es descrita en OASIS a través de las relaciones entre clases, que incluyen la especialización y la agregación. Como ocurre en la mayoría de los lenguajes orientados a objetos, la especificación de un sistema (su esquema conceptual) consiste en un conjunto de clases descritas usando un determinado lenguaje (OASIS, en este caso); por ejemplo, en el caso de una biblioteca, el esquema estará compuesto por las clases libro, lector, bibliotecario, préstamo, etc. En el esquema están descritas tanto la estructura como la función del sistema.

La aproximación de OASIS a la evolución se basa, al igual que MEDES, en la posibilidad de variar la estructura del sistema de modo que su función también cambie [10, 11]; dentro del modelo OO, lo que MEDES denomina el Metasistema viene representado por un conjunto de meta-objetos cuyo estado codifica la estructura del sistema, y cuyos servicios permiten variarla. Algunos de los atributos de esos meta-objetos son lista-atributos, lista-eventos, precondiciones, restricciones, y disparos. Entre los servicios, por su parte, aparecen añadir-atributo, eliminar-atributo, añadir-evento, y otros. Un esquema OASIS se puede ver como un meta-objeto compuesto por otros meta-objetos, uno por cada una de las clases del esquema. El estado de cada meta-objeto, define la estructura y el comportamiento de todos los posibles ejemplares de la clase asociada, que pueden ser cambiados a través de los servicios del mismo.

Además, cada metaobjeto posee servicios de creación y destrucción de ejemplares (new y destroy respectivamente). Cuando a un metaobjeto se le invoca el evento new, se crea un nuevo objeto cuyos atributos vienen definidos por el valor del atributo lista-atributos del metaobjeto en ese momento; sus eventos por el valor de lista-eventos, y así sucesivamente (ver Figura 3). Finalmente, un atributo población permite a los metaobjetos saber qué objetos ha creado y no ha destruido.

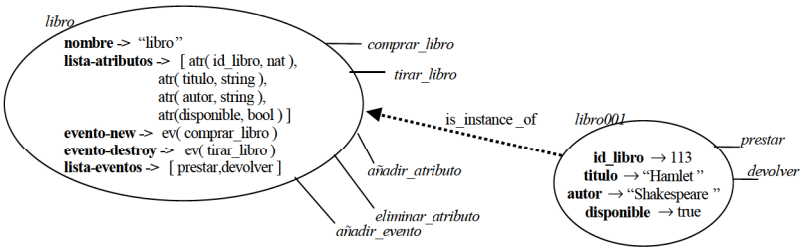


Fig. 3. Relación *is_instance_of* entre el metaobjeto libro y su ejemplar libro001.

Nótese que la noción de meta-objeto coincide con la noción de clase en algunas aproximaciones OO, fundamentalmente en el ámbito de las bases de datos [13], y por ello a partir de ahora usaremos indistintamente ambos términos. La visión objetual de las clases permite a OASIS tratar la evolución de una manera homogénea, puesto que el patrón de comportamiento de los metaobjetos es exactamente el mismo que el de los objetos OASIS. De hecho, incluso los metaobjetos son ejemplares de la *metaclase*, un metaobjeto que, con el fin de cerrar la circularidad, se define como ejemplar de sí mismo. El estado del metaobjeto que define la metaclase codifica la estructura y comportamiento de las clases OASIS (ver figura 3).

Actualmente, y porque así se ha decidido, la estructura de la metaclase es invariable (esto es, las precondiciones deshabilitan los eventos de modificación) y define el modelo OASIS; si se permitiera variar su estructura, estaríamos ante un caso de evolución no ya de un sistema, sino del propio modelo OASIS.

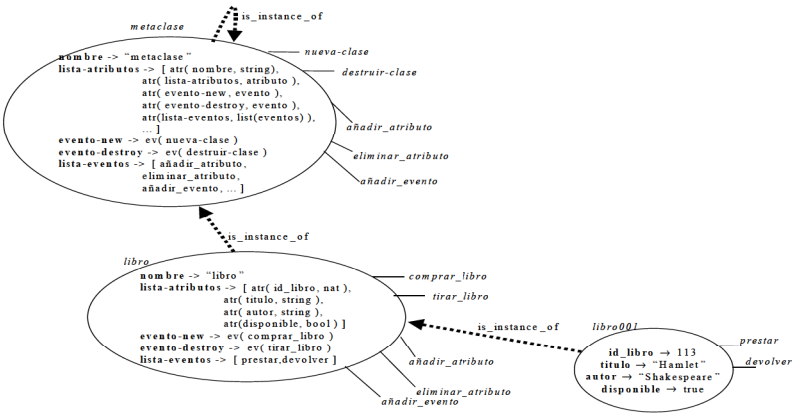


Fig. 4. Relación *is_instance_of* entre las metaclases, las clases y los objetos.

4 Un marco común OASIS-MEDES

En cualquier desarrollo de software se manejan distintas representaciones del conocimiento del dominio, que poseen grados diferentes de formalización Durante la Ingeniería de Requisitos el conocimiento del dominio generalmente se captura en lenguaje natural. Posteriormente esta representación se estructura y (en ocasiones) se formaliza hasta llegar a una versión ejecutable del sistema. En general, se realiza un proceso de traducción entre distintas formas de representación que conduce a tener que enfrentarse con dos situaciones concretas [29]:

1. La evolución de los modelos, es decir, cambios en los modelos del sistema sin un cambio de lenguaje, es decir, sin un cambio de meta-modelo.
2. La evolución del meta-modelo: cambios en el modelo de sistema que requieren un cambio de lenguaje (meta-modelo).

En ambos casos aparecen dos problemas:

1. La traducibilidad, es decir, la posibilidad de derivar un modelo de otro modelo expresado en un lenguaje diferente.
2. La trazabilidad: la posibilidad de navegación entre construcciones semánticas equivalentes en diferentes modelos.

La traducibilidad es el problema fundamental en la evolución del meta-modelo y la trazabilidad es importante en ambas situaciones. Es decir, durante el proceso de desarrollo se elaboran modelos informales, semiformales y formales. Estos modelos evolucionan y, además, es conveniente que exista trazabilidad entre ellos y que la traducibilidad sea lo mayor posible.

La exposición de las características de OASIS y MEDES muestra que ambos métodos proponen un enfoque formal que permite un proceso de razonamiento para asegurar la consistencia de los sistemas especificados.

OASIS enfatiza un enfoque orientado a objetos en el que la estructura y el comportamiento de los objetos se establece formalmente mediante Transaction-Frame Logic [18]. La evolución se permite mediante el concepto de metaclase

MEDES selecciona un conjunto más restringido de conceptos y otro enfoque lógico (el temporal) debido a que el objetivo es proporcionar conceptos cercanos a la visión funcional del usuario.

Siguiendo el esquema propuesto en el apartado anterior, deberíamos considerar los problemas de trazabilidad y traducibilidad entre ambos teniendo en cuenta el diferente nivel de abstracción que utilizan para modelizar sistemas software:

1.*Trazabilidad.*

1.1. Trazabilidad interna: OASIS y MEDES aseguran la trazabilidad interna entre los conceptos usados en sus modelos.

1.2. De OASIS a MEDES, y viceversa: puesto que OASIS presenta un conjunto de conceptos más variado y dado que el enfoque lógico es diferente, habrá que estudiar el grado de trazabilidad entre los conceptos de OASIS y los conceptos MEDES.

2.*Traducibilidad.*

La diferencia más radical entre ambos es que MEDES no toma ninguna decisión acerca de la representación del estado del sistema en forma de objetos, es decir, adopta una visión funcional del sistema: los procesadores equivalen a actores que realizan acciones, sin tomar ninguna decisión acerca de la estructura de datos del

sistema, lo cual, inicialmente, permite prototipar a un nivel de abstracción bastante alto. Ello, además, permite tratar los problemas de propagación del cambio de forma integrada con la evolución de los componentes de un sistema.
Por otra parte, todos los componentes del modelo de evolución de MEDES tienen su reflejo en uno o más componentes del modelo OASIS. Así, se puede pensar en los metaobjetos como procesadores del metasistema, la HFS está recogida en las trazas (secuencias de eventos ocurridos a lo largo del tiempo) de los ejemplares de las clases del esquema; por su parte, las trazas de los metaobjetos recogen la HES. La funcionalidad del SG está recogida en el comportamiento de las clases que, como se ha dicho anteriormente, permiten crear y destruir ejemplares, permiten cambiar la estructura y comportamiento de los mismos dinámicamente, y además soportan procesos de migración de ejemplares para adecuarlos a su nueva estructura y/o comportamiento. Parece, pues, que caben diferentes posibilidades de equivalencia entre ambos:

1. Utilizar OASIS como una de las implementaciones de MEDES, es decir, utilizar OASIS como lenguaje objeto de MEDES. Posiblemente no aporte ningún elemento interesante ya que supondría buscar una representación del modelo MEDES en OASIS sin cambiar el nivel de abstracción.
2. Construir un esquema de traducción de sistemas MEDES a sistemas OASIS. Ello requeriría un proceso de estatización (determinación de los objetos que representan estado) con pérdida de ciertas propiedades evolutivas en el nivel de MEDES, es decir, una vez traducida una especificacion a OASIS la evolución habría que realizarla en el propio OASIS.
3. Tratar de obtener de MEDES elementos directamente traducibles a OASIS, como la especificación de los procesos. A partir de ellas se podrían completar los modelos OASIS añadiendo aspectos relacionados con el estado.

5 Conclusiones y trabajo futuro

Este estudio inicial de comparación de los enfoques de OASIS y MEDES en el modelado de sistemas software muestra que hay puntos de compatibilidad y de complementariedad entre ambos, que pueden aprovechar diferentes puntos de vista en la realización de prototipos ejecutables de sistemas software.
Actualmente se está trabajando en la valoración de las estrategias citadas en el punto anterior, para determinar cuál de ellas resulta más favorable desde la perspectiva del desarrollo de software capaz de evolucionar. Como trabajo futuro aparece la posibilidad de emplear la expresividad de MEDES para el modelado funcional como punto de entrada al modelado de sistemas mediante OASIS. En concreto, MEDES podría utilizarse en la simulación de casos de uso a partir de los cuales obtener una traducción a la especificación de procesos en OASIS.

Bibliografía

1. Agresti, W.W.: Tutorial: New Paradigms for Software Development, Los Angeles.CA IEEE Computer Society Press (1986)
2. Anaya, A. M.J. Rodríguez, J. Parets: Representation and Management of Memory and Decision in Evolving Sofware Systems, LNCS, Vol 1333, 71-82 (1997)
3. Bandinelli, S.C., Fuggetta, A., Ghezzi, C.: Software Process Model Evolution in the SPADE Environment, IEEE Trans.S.E. Vol. 19.12, 1128-1144 (1993)
4. Banerjee J., Kim, W., Kim, H.K., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases, In Proc. Of ACM-SIGMOD International Conference on Management of Data, San Francisco CA, Mayo (1987)
5. Belady, L.A., Lehman, M.M.: A Model of Large Program Development, IBM SYST.J. Vol. 15.3, 225-252. (1976)
6. Berzins, V., Luqi, Yehudai, A:. Using Transformations in Specification-Based Prototyping, IEEE Trans.S.E. Vol. 19.5, 436-452 (1993.)
7. Boehm, B.W.: A Spiral Model of Software Development and Enhancement, ACM SIGSOFT S.E.NOTES. AUGUST Vol. 11, 14-24 (1986)
8. Boehm, B.W., Egyed, A., Kwan, J., Port, D., Shah A., Madachy, R.: Using the WinWin Spiral Model: A Case Study, IEEE Computer, 33-44, July (1998)
9. Bohem, B., In, H.: Identifying quality requirement conflicts. IEEE-SOFTWARE, Vol. 13(2), 25ss. (1996)
10. Carsí J.A., Ramos I., Canós J.H.: La Metaclase OASIS, IDEAS'98, Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software, Torres-BRASIL, 12 pág., Abril (1998)
11. Carsí J.A., Ramos I., Penadés M.C., Pelechano V.: Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic, JIS'96, I Jornadas de Trabajo en Ingeniería del Software, Sevilla, 12 pág., Noviembre (1996)
12. Casais, E.: Managing Class Evolution in Object-Oriented Systems. In: Object Management. Tsichritzis, D. GENEVE. Centre Universitaire d'Informatique. 133-195 (1990)
13. Catell R.: The Object Database Standard: ODMG-93, Release 1.1, Morgan Kaufmann Publishers (1994)
14. Conradi, R., Fernström, C., Fuggetta, A.: A Conceptual Framework for Evolving Software Processes, ACM SIGSOFT SEN Vol. 18.4. 26-35 (1993)
15. Gabbay, D.M.,Reynolds, M.: Towards a computational treatment of time. In: D.M.Gabbay, C.J.Hogger: Handbook of logic in Artificial Intelligence and Logic Programming. Volume 4: Epistemic and Temporal Reasoning, Oxford Science Publications, Oxford: Clarendon press , 351-431 (1994)
16. Henderson-Sellers, B., Edwards, J.L.: The Object-Oriented Systems Life Cycle, CACM 33.9. p.143-159. (1990)
17. Johnson, W.L , Feather, M.S. , Harris, D.R.: Representation and Presentation of Requirements Knowledge. IEEE Trans.S.E. 18.10, 853-869 (1992)
18. Kifer M., Deductive and Object Data Languages: A Quest for Integration, Keynote address at the 4-th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, December Springer's LNCS 1013 (1995)
19. Kozaczynsky, W., Ning, J., Engberts, A.: Program Concept Recognition and Transformation, IEEE Trans.S.E. 18,12. p.1065-1075 (1992)
20. Le Moigne, J-L: La théorie du système général. Théorie de la modélisation, PARIS, Presses Universitaires de France (1977-1990)
21. Lehman, M.M.: Programs, Life Cycles and the Laws of Software Evolution, PROCEED.IEEE 68.9, 1060-1076, (1980)

22. Lieberherr, K.J., Silva-Lepe, I., Xiao, C.: Adaptative Object-Oriented Programming Using Graph-Based Customization, CACM Vol. 37.5, 94-101 (1994)
23. Lieberherr, K.J., Xiao, C.: Object-Oriented Software Evolution, IEEE Trans.S.E., Vol. 19.4, 313-343 (1993)
24. Luqi. A Graph Model for Software Evolution, IEEE Trans.S.E., Vol. 16.8, 917-927 (1992)
25. Madhavji, N., Schäfer, W. Prism - Methodology and Process-Oriented Environment. IEEE Trans.S.E. Vol. 17.12, 1270-1283 (1991)
26. Madhavji, N.H. Environment Evolution: The Prism Model of Changes, IEEE Trans.S.E. Vol. 18.5, 380-392 (1992)
27. Manna, Z. , Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems Specification, Springer-Verlag, NY (1992)
28. Merks, E.A.T., Dyck, J.M., Cameron, R.D.: Language Design for Program Manipulation, IEEE Trans.S.E., Vol. 18.1, 19-32 (1992)
29. Parets, J. Grünbacher, P.: Capturing, Negotiating, and Evolving System Requirements: Bridging WinWin and the UML, Aceptado en EUROMICRO99. Septiembre (1999)
30. Parets, J., Anaya, A., Rodriguez, M.J., Paderewski, P.: A Representation of Software Systems Evolution Based on the Theory of the General System, In: Computer Aided Systems Theory - EUROCAST'93', Pichler, F., Moreno-Diaz, R. (Eds.). LNCS 763, Springer-Verlag, 96-109 (1994)
31. Parets, J., Torres, J.C.: A Language for Describing Complex-Evolutive Software Systems In: Computer Aided Systems Theory - EUROCAST'95, Pichler, F., Moreno-Diaz, R. (Eds.). LNCS 1030, Springer-Verlag, 181-199 (1996)
32. Parets-Llorca, J.: On Some Epistemological Challenges of Object-Oriented Software Engineering: If Objects were Systems, 2ème. Congres Europeen de Systemique Vol. 2, 447-456 (1993)
33. Parets-Llorca, J.: Reflexiones sobre el proceso de concepción de sistemas complejos. MEDES: un método de especificación, desarrollo y evolución de sistemas software, Tesis Doctoral, Universidad de Granada (1995)
34. Pastor, O. et al., OASIS version 2 (2.2): A Class-Definition Language to Model Information Systems using an Object-Oriented Approach, SPUPV-95.788, Universitat Politècnica de València (1995)
35. Plaice, J., Wadge, W. A: New Approach to Version Control, IEEE Trans.S.E., Vol. 19.3, 268-276 (1993)
36. Ramamoorthy, C.V., Usuda, Y., Prakash, A., Tsai, W.T.: The Evolution Support Environment System, IEEE Trans.S.E., Vol. 16.11, 1225-1234 (1990)
37. Ribeiro Filho, J.L. , Treleaven, P.C. , Alippi, C.: Genetic-Algorithm Programming Environments. Computer, Vol. 27.7, 28-43 (1994)
38. Rodriguez, M.J., Parets, J.: Cambio en la estructura de decisión en un sistema evolutivo, II Jornadas de Ingeniería del Software, JIS'97, San Sebastián , Septiembre (1997)
39. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F.: Object-Oriented Modeling and Design, Prentice-Hall, NJ (1991)
40. Sommerville, I., Thomson, R.: An Approach to the Support of Software Evolution, THE COMPUTER JOURNAL , Vol. 32.5, 386-398 (1989)
41. Srinivas, M. , Patnaik, L.M.: Genetic Algorithms: A Survey. Computer, Vol. 27.6, 17-26 (1994)
42. Tardieu, H.: Issues for Dynamic Modelling through Recent Developments in European Methods In: Dynamic Modelling of Information Systems, II. Sol, H.G., Crosslin, R.L. (Eds.), North-Holland, Amsterdam, 3-23 (1992)
43. Tardieu, H., Rochfeld, A., Coletti, R.: La méthode Merise. Principes et outils, Editions d'Organisation, Paris (1983)
44. Tully, C.: Representing and Enacting the Software Process: Proceds. of the 4th International Software Process Workshop, ACM SIGSOFT S.E.NOTES, Vol. 14.4 (1989)

Computación y Sistemas Vol. 3 No. 4 pp. 287 - 295
© 2000, CIC - IPN. ISSN 1405-5546 Impreso en México

Una Interpretación de la Herencia de *OASIS* en Lógica Dinámica*

Pedro Sánchez **Patricio Letelier** **Isidro Ramos**
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, España
Telef. +346387357
{ppalma, letelier, iramos}@dsic.upv.es

Artículo recibido el 17 de marzo, 1999; aceptado el 10 de febrero, 2000

Resumen

OASIS es un lenguaje formal para la especificación de requisitos de sistemas de información que sigue el enfoque orientado a objetos. La herencia en OASIS se trata taxonómicamente definiendo particiones estáticas y dinámicas. Las primeras suponen un particionamiento del espacio total de objetos. Las segundas representan un particionamiento del espacio de estados de los objetos y se produce por la ocurrencia de acciones o por la caracterización del estado del objeto a través de valores de atributos. En OASIS el formalismo utilizado para dar cuenta del comportamiento del objeto es una variante de Lógica Dinámica que incluye los operadores deónticos de permiso, prohibición y obligación. Se ha establecido un marco de trabajo para animar especificaciones OASIS dado el interés existente en la validación temprana de requisitos. Este trabajo presenta un procedimiento automatizable para interpretar las particiones estáticas y dinámicas de OASIS como fórmulas en la variante de Lógica Dinámica utilizada. Así, es posible animar (de forma homogénea y usando el marco establecido) especificaciones OASIS con herencia.

Palabras clave: Ingeniería de requisitos, métodos orientados a objetos, herencia, lógica dinámica.

*Este trabajo ha sido parcialmente financiado por la "Comisión Interministerial de Ciencia y Tecnología" (CICYT) a través del proyecto con referencia TIC 97-0593-C05-01.

1 Introducción

Un aspecto importante en el modelado conceptual de sistemas de información es el hecho de incorporar tanto la perspectiva estructural como la de comportamiento. La aparición del enfoque orientado a objetos ha contribuido enormemente a la integración de estos dos aspectos. Distintos enfoques abordan el modelado conceptual, bien de manera semi-formal como la notación estándar UML (Rumbaugh et al., 1997), bien formalmente mediante lenguajes como TROLL (Grau et al., 1998), Albert II (Du bois et al., 1997), OBLOG (OBLOG Sw., 1999). En esta línea formal, *OASIS* 3.0 (Letelier et al., 1998) provee un enfoque OO y formal para la especificación de modelos conceptuales.

Paralelamente, la animación automática de especificaciones formales (Siddiqi et al., 1997) permite la captación incremental de requisitos y su validación temprana. Para ello, es necesario disponer de un modelo formal, suficientemente expresivo y factible de ser incorporado en una herramienta que dé soporte a la animación de especificaciones de requisitos (Sánchez et al., 1999). La animación de especificaciones *OASIS* de manera fiel a la semántica del modelo e integrada en un entorno CASE nos ha impulsado a la búsqueda de un formalismo homogéneo a partir del cual pueda definirse e implantarse un modelo de ejecución (Letelier et al., 1999c). Dentro de los trabajos de animación con *OASIS* pueden destacarse los relativos a Redes de Petri Objetuales (Sánchez et al., 1997; Letelier et al., 1999d) y a Programación Lógica Concurrente (Letelier et al., 1999a; Letelier et al., 1999d).

En *OASIS*, el modelado del comportamiento puede realizarse desde dos perspectivas complementarias en cuanto a expresividad. Parte del comportamiento se especifica localmente en cada clase mediante evaluaciones, precondiciones, disparos, operaciones, protocolos y restricciones de integridad. Adicionalmente, mediante las relaciones de especialización entre clases se hereda el comportamiento desde las superclases en jerarquías de

Sánchez, P., Letelier, P., Ramos, I. Una Interpretación de la Herencia de OASIS en Lógica Dinámica. Computación y Sistemas, vol. 4, núm.4. 2000.

herencia. La **herencia** es una característica fundamental del paradigma objetual que describe cómo una clase hereda propiedades desde otra aunque no existe un consenso total en cuanto a su semántica (Taivalsaari, 1996). En particular, su interpretación en el *modelado conceptual* dista bastante de la que se asume en los lenguajes de programación orientados a objetos. Usualmente, la especialización usada en un modelo conceptual se traslada a herencia en algún lenguaje de programación mediante el uso de patrones de diseño, o bien, se utiliza en un mismo nivel de abstracción la herencia del lenguaje.

La especialización de *OASIS* se puede caracterizar como un uso disciplinado de la herencia, al incluir una serie de constructores específicos que facilitan el modelado y determinan una semántica precisa. Este trabajo presenta una procedimiento automatizable para trasladar la especialización de *OASIS* (*particiones estáticas* y *dinámicas*) a un conjunto de fórmulas en la variante de Lógica Dinámica utilizada (que se describe en la sección 3). Así, desde una perspectiva global, la especificación de una clase *OASIS* y la jerarquía de herencia modelada pueden ser vistas como un conjunto de fórmulas en dicha lógica. La estrategia que aquí se presenta es obtener una especificación *OASIS* equivalente a la original de manera que no incluya relaciones de especialización entre las clases (realizando transformaciones en el formalismo subyacente). En este sentido, merece destacarse que el interés está en poder animar de manera fiel a la semántica del modelo y no el producir código final. Muchos aspectos del procedimiento descrito en este trabajo pueden ser aprovechados por otros trabajos de animación de modelos conceptuales que deseen seguir una estrategia similar.

La sección 2 de este trabajo presenta el marco formal en el que se basa la especialización de *OASIS*. La sección 3 introduce los conceptos de Lógica Dinámica utilizados. La sección 4 establece el mecanismo de representación para cualquier tipo de particiones y un resumen del proceso de traducción. En la sección 5 se presentan las conclusiones y el trabajo futuro.

2 Herencia en *OASIS*

Las clases poseen un tipo concreto que establece el conjunto de propiedades de todas sus instancias posibles.

La plantilla (conjunto de propiedades) de una **subclase** se define extendiendo y/o modificando las plantillas de otras clases llamadas **superclases** y a partir de las cuales hereda.

En términos generales, la clase que hereda dispone de las propiedades de otra clase y puede añadir o cambiar algunas de éstas. En *OASIS*, la herencia está inspirada en los trabajos de (Wieringa et al., 1995) y se expresa mediante particiones y grupos de rol que establecen mecanismos de modificación incremental de propiedades y relaciones de orden entre los tipos o plantillas de las clases.

2.1 Clasificación de objetos

Una clase tiene asociado un conjunto potencial de objetos que llamamos instancias de la clase. Para cada clase C, se introducen los siguientes conceptos:

- La **intensión** de una clase, $int(C)$, es el conjunto de todas las propiedades que son compartidas por todas sus instancias (representa la plantilla de la clase).
- La **extensión** de una clase, $ext(C)$, es el conjunto de todas las **posibles** instancias de la clase.

Siendo C_1 y C_2 dos clases, si $ext(C_1) \subseteq ext(C_2)$, entonces diremos que C_1 es subclase de C_2 y además se cumple que $int(C_2) \subseteq int(C_1)$.

2.2 Particiones estáticas

Siendo $C_1, ..., C_n$ subclases de otra clase C_0, decimos que $\{C_1, ..., C_n\}$ es una **partición estática** de C_0 si se cumple que:

$$ext(C_0) = \cup ext(C_i), \forall i = 1, ..., n$$
$$ext(C_i) \cap ext(C_j) = \emptyset, \forall i \neq j, \forall i, j = 1, ..., n$$

Se puede particularizar a cualquier instante t del sistema:

$$ext_t(C_0) = \cup ext_t(C_i), \forall i = 1, ..., n$$
$$ext_t(C_i) \cap ext_t(C_j) = \emptyset, \forall i \neq j, \forall i, j = 1, ..., n$$

De esta forma, existe una cardinalidad implícita asociada a cada partición por la que cada instancia de la subclase está relacionada con exactamente una instancia de la superclase y viceversa. Así, si o es una instancia de C_0, entonces es instancia de exactamente una subclase de cada partición[1] existente.

Ejemplo 1 *Dos particiones estáticas de una misma clase en OASIS:*

```
camion, coche, otro_vehiculo
    static specialization of vehiculo;
gasolina, diesel, otro_tipo
    static specialization of vehiculo;
```

[1] Nótese que se pueden definir varias particiones para una misma clase

2.3 Particiones dinámicas

Siendo $C_1, ..., C_n$ subclases de otra clase C_0, decimos que $\{C_1, ..., C_n\}$ es una **partición dinámica** de C_0 si para cada instante t se cumple que:

$$ext_t(C_0) = \cup ext_t(C_i), \forall i = 1, ..., n$$
$$ext_t(C_i) \cap ext_t(C_j) = \emptyset, \forall i \neq j, \forall i, j = 1, ..., n$$

Sin embargo, puede haber instantes del sistema t_1 y t_2, con $t_1 \neq t_2$, y también i, j con $i \neq j$, tales que:

$$ext_{t_1}(C_i) \cap ext_{t_2}(C_j) \neq \emptyset$$

Y por lo tanto, se tendrá que:

$$ext(C_i) = ext(C_j), \forall i, j = 0, ...n$$

Las dos primeras fórmulas son idénticas a las que satisfacen las particiones estáticas. La tercera fórmula implica que pueden existir al menos dos instantes distintos tal que al cambiar del uno al otro, al menos un objeto "se mueve" desde una subclase a otra. La cuarta fórmula es consecuencia de que, en principio, cualquier objeto puede llegar a estar en cualquiera de las subclases de la partición y la igualdad se mantiene también respecto de la superclase C_0. En el ejemplo anterior de la partición estática, una instancia de la clase `camión` no puede pasar a serlo de la clase `coche`, o viceversa.

Ejemplo 2 *Una especialización dinámica en OASIS de la clase `coche` determinada por la ocurrencia de las acciones `crear_coche`, `reparar` y `estropear`:*

```
funcionando, estropeado dynamic
specialization of coche
    migration relation is
coche = crear_coche.funcionando;
funcionando = estropear.estropeado;
estropeado = reparar.funcionando;
```

Según lo anterior, la creación de una instancia de `coche` implica comenzar perteneciendo a la subclase `funcionando`. Cuando sea una instancia de `funcionando` le ocurrirán acciones de la signatura de `coche` más, posiblemente, otras de la subclase `funcionando`. Entre ellas está el evento `estropear` que pertenece a la signatura de `funcionando`. Su ocurrencia implica salir de la subclase que representa a `funcionando` y entrar en la indicada en el proceso migratorio (la subclase `estropeado`). Desde un punto de vista teórico, el proceso que representa la vida de una instancia de `coche` es la composición de los distintos subprocesos de las subclases y los puntos de enlace entre ellos vienen dados por las acciones indicadas en el proceso migratorio.

Ejemplo 3 *Una partición dinámica en OASIS de la clase `persona` en función de sus atributos:*

```
niño where {edad<14},
adolescente where {14<=edad and edad<18},
adulto where {18>=edad}
    dynamic specialization of persona;
```

Dicha jerarquía de herencia tiene como atributo relevante `edad` de la clase `persona` cuyo valor se calcula como diferencia entre la fecha actual y la fecha de nacimiento. En este caso, independientemente de las acciones que lleven a la modificación del atributo `edad`, se ha preferido representar la migración en función de dicho atributo y no de acciones que acontezcan.

2.4 Especies y herencia múltiple en *OASIS*

A partir de las posibles combinaciones de subclases no particionadas se obtiene un conjunto de subclases, llamadas **especies**, con las siguientes propiedades:

- Son disjuntas entre ellas.
- Cubren todas las posibles situaciones del objeto en la jerarquía (dando solución a la herencia múltiple).

El concepto de especie aporta ventajas de tipo metodológico dado que el modelo se hace más claro y reduce las posibilidades de error.

Ejemplo 4 *Algunas especies para la jerarquía de la clase vehículo:*

```
camion*diesel, coche*gasolina,
otro_vehiculo*diesel, ...
```

Ejemplo 5 *La clase `camión*diesel` puede tener como propiedad emergente un atributo que represente la fecha del último cambio de filtro de combustible. Este atributo debe añadirse a la especificación de dicha especie. Para ello, es necesario explicitar la plantilla de la clase `camión*diesel` (como para el resto de clases).*

Se asume que para las especies sin propiedades emergentes no será preciso definir su plantilla.

3 *OASIS* expresado en Lógica Dinámica

Como se expone en (Letelier et al., 1998) toda la plantilla de una clase en *OASIS* es representable como un conjunto de fórmulas en Lógica Dinámica. Es decir, para cada evaluación, derivación, precondición, disparo, restricción de integridad, proceso de obligación y proceso de permiso existirá un conjunto de fórmulas en

Lógica Dinámica que capta (parcialmente) la semántica del lenguaje.

El objetivo de este apartado es dar las pautas necesarias para considerar la relación de especialización establecida entre las clases en el proceso de traducción a Lógica Dinámica. Una vez más, todos los aspectos relativos a la especialización de *OASIS* son trasladables a alguno de los tres tipos de fórmulas consideradas: de cambio de estado, de permiso y de obligación. Dicho de otro modo, a partir de las fórmulas que caracterizan el estado del objeto $\{\psi, \phi, ...\}$ se usarán los siguientes tipos de fórmulas en Lógica Dinámica (siendo a una acción):

$\psi \rightarrow [a]false$	ocurrencia de a prohibida en estados que satisfacen ψ
$\psi \rightarrow [\neg a]false$	ocurrencia de a obligatoria en estados que satisfacen ψ
$\psi \rightarrow [a]\phi$	en estados que satisfacen ψ, tras la ocurrencia de a ϕ debe satisfacerse

Estas fórmulas constituyen un sublenguaje del lenguaje de Lógica Dinámica propuesto y formalizado en (Meyer, 1998).

En *OASIS*, la plantilla de clase puede verse formada por la tupla $\langle Atr, Ev, Form, Procesos\rangle$, es decir, por *atributos*, *eventos*, *fórmulas* en Lógica Dinámica y *procesos*. Con las consideraciones expuestas en (Letelier et al., 1998) es posible simplificar la plantilla a la tupla $\langle Atr, Ev, FormLD\rangle$ siendo $FormLD$ el conjunto de fórmulas en lógica dinámica correspondiente a $Form$ y $Procesos$. Para distinguir las propiedades de distintas clases se indicará con un subíndice el nombre de la clases a la que pertenecen. Así, Atr_i representa los atributos de la clase C_i.

En la siguiente sección se detalla cómo la expresividad incluida en la especificación de particiones estáticas y dinámicas puede ser descrita mediante fórmulas adicionales en la Lógica Dinámica utilizada.

4 Jerarquía de Herencia

Una especie involucra a clases de la misma jerarquía de especialización. Además, las clases que forman la especie tienen una clase como ancestro común que está particionada en más de una partición. Dada la jerarquía de especialización de la Figura 1, en la que no distinguimos si las particiones son estáticas o dinámicas, las especies posibles son:

C_8

$C_3 * C_1$ $\quad C_3 * C_6$ $\quad C_3 * C_7$

$C_4 * C_1$ $\quad C_4 * C_6$ $\quad C_4 * C_7$

$C_5 * C_1$ $\quad C_5 * C_6$ $\quad C_5 * C_7$

Por el contrario, no son especies $C_3 * C_4, C_0 * C_7$, etc.

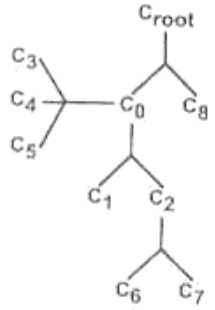


Figure 1: Una jerarquía de especialización.

El interés de este trabajo está en determinar las propiedades que posee una especie dado que los objetos en el modelo *OASIS* son instancias de alguna de las especies posibles. Para calcular estas propiedades se presentan de forma simplificada algunos casos típicos que contengan particiones estáticas y dinámicas y se extiende el procedimiento para cualquier jerarquía de herencia.

A continuación se presenta la codificación en lógica dinámica para el caso en el que sólo hay particiones estáticas.

4.1 Jerarquía de particiones estáticas

Dada una jerarquía de especialización que contenga únicamente particiones estáticas, el conjunto de propiedades que describe el comportamiento de una clase hoja en la jerarquía es la unión de las propiedades (explicitadas en las clases) desde la clase raíz de la jerarquía hasta dicha clase hoja.

El objetivo es establecer las propiedades que por herencia están presentes en las clases hoja de la jerarquía pues son éstas las únicas que forman parte de cualquier especie. Dado que en el modelo de herencia de *OASIS* se mantiene la compatibilidad de comportamiento para las particiones estáticas y dinámicas, las propiedades de toda clase hoja $C_i$de la jerarquía vienen dadas por:

i) El conjunto[2] de atributos es igual a:

$$Atr_i = \bigcup_{C_k=C_i}^{raiz} Atr_k$$

ii) El conjunto de eventos es igual a:

$$Ev_i = \bigcup_{C_k=C_i}^{raiz} Ev_k$$

iii) Dado que las evaluaciones en una subclase sólo pueden modificar atributos emergentes, el conjunto de evaluaciones es la unión de las evaluaciones desde la clase hoja hasta la clase raíz de la jerarquía (es decir, la unión de fórmulas en Lógica Dinámica para evaluaciones).

iv) Dado que si una prohibición se modifica en la subclase la nueva condición debe implicar lógicamente a la condición de la superclase, el conjunto de prohibiciones es la unión de las prohibiciones desde la clase raíz hasta la clase hoja de la jerarquía y sobreescribiendo las prohibiciones para la misma acción.

v) Para las obligaciones el cálculo es el mismo que el indicado en las prohibiciones.

Una vez que se ha determinado el conjunto de propiedades tanto estructurales como de comportamiento, correspondiente a cada clase hoja de la jerarquía de herencia, es inmediato el cálculo de las propiedades de cualquier especie.

La plantilla correspondiente a una especie $(C_0 * C_1 * ... * C_n)$ viene dada por la tupla:

$$Atr_0 \cup ... \cup Atr_n \cup Atr_e,$$
$$Ev_0 \cup ... \cup Ev_n \cup Ev_e,$$
$$FormLD_0 \cup ... \cup FormLD_n \cup FormLD_e$$

donde Atr_i, Ev_i y $FormLD_i$ son los atributos, eventos y fórmulas en Lógica Dinámica de la clase C_i (respectivamente) y Atr_e, Ev_e y $FormLD_e$ representan los atributos, eventos y fórmulas en Lógica Dinámica (respectivamente) emergentes y especificados para dicha especie $(C_0 * C_1 * ... * C_n)$. Para el caso en que se solapen las propiedades, el cálculo de las fórmulas sigue las mismas restricciones anteriores de manera que se imponen las condiciones más restrictivas de aplicación de los permisos y las obligaciones y sobre todas ellas las especificadas como emergentes desde la especie.

[2]Un posible conflicto de nombre debería evitarse reescribiendo el atributo de manera que refleje la clase de la que proviene.

4.2 Jerarquía de particiones dinámicas

Dado que el modelo de herencia de *OASIS* no permite particionar de forma estática una subclase que provenga de una partición dinámica, las particiones dinámicas ocupan subárboles que llegan hasta las clases hoja de la jerarquía de especialización. La Figura ?? muestra una jerarquía ejemplo en la que a la clase C_2 se le ha asociado una partición dinámica.

El tratamiento para determinar las propiedades de las especies es distinto cuando aparecen particiones dinámicas. Como se ha comentado anteriormente, una partición dinámica posibilita un proceso migratorio especificado bien como un proceso (basado en la ocurrencia de acciones), bien mediante valores de atributos. Para el caso del proceso migratorio se usa en *OASIS* la misma sintaxis que para la especificación de los protocolos. Para el caso de particionamiento en función de los valores de los atributos se caracteriza la partición mediante fórmulas en lógica ecuacional de primer orden. En ambos casos, las particiones dinámicas constituyen un particionamiento del espacio de estado de los objetos.

La idea es trasladar de manera "ascendente" las propiedades de las subclases que forman las particiones dinámicas a la superclase particionada. Para el caso de la Figura 2, se pretende ampliar el conjunto de fórmulas dinámicas de la clase C_2 distinguiendo la ampliación de C_2 por incorporar los aspectos de C_6 respecto de la ampliación de C_2 por incorporar los aspectos de C_7.

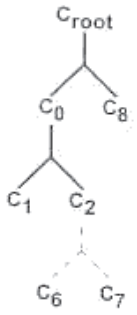


Figure 2: Partición dinámica de una partición estática.

A continuación se describe para cada tipo de partición dinámica las modificaciones en el conjunto de fórmulas en lógica dinámica de la clase particionada.

Migración por ocurrencia de acciones Siendo $\{C_1, ..., C_n\}$ una partición dinámica de la clase C_j, la plantilla correspondiente a la clase C_ja la que se le incorporan las propiedades definidas en las subclases C_i de la partición (con $i = 1, ..., n$), viene determinada por:

i) Conjunto de atributos: $Atr = Atr_j \cup Atr_1 \cup ... \cup Atr_n$.

ii) Conjunto de eventos: $Ev = Ev_j \cup Ev_1 \cup ... \cup Ev_n$.

iii) El conjunto de fórmulas de evaluación es la unión de las fórmulas de evaluación de C_j y desde C_1 hasta C_n.

iv) El conjunto de fórmulas de prohibición es la unión de las fórmulas de prohibición de C_j y desde C_1 hasta C_n. Dada la restricción de compatibilidad de comportamiento, si C_i redefine una prohibición de C_j prevalece entonces la prohibición de C_i.

v) Las fórmulas de obligación se calculan de manera análoga a las fórmulas de prohibición.

vi) El proceso migratorio definido en la partición pasa a formar parte de la clase C_j como protocolo que define secuencias válidas en el acontecer de las acciones[3]. Las fórmulas por permisos se incorporan directamente a las fórmulas por permisos que posea la clase C_j. Cada una de las propiedades de comportamiento de cada clase C_i debe añadir como condición de aplicación de la fórmula un predicado de ocurrencia que caracterice la constante agente del estado en el que se encuentra el objeto.

vii) Repetir el proceso i) a vi) para la siguiente partición dinámica de C_j considerando como propiedades las recientemente calculadas.

Ejemplo 6 *Dada la partición dinámica definida por la especificación OASIS:*

```
C6, C7
dynamic specialization of C2
migration relation is
    C2 = new.C6
    C6 = a6.C7
    C7 = a7.C6
```

donde además se tienen las propiedades en Lógica Dinámica siguientes:

$$\Phi_2 \rightarrow [a_2]\Psi_2 \quad \text{en la clase } C_2$$
$$\Phi_6 \rightarrow [a_6]\Psi_6 \quad \text{en la clase } C_6$$
$$\Phi_7 \rightarrow [a_7]\Psi_7 \quad \text{en la clase } C_7$$

entonces, al trasladar estas propiedades a la clase C_2las fórmulas anteriores se sustituyen por las siguientes[4]:

$$\Phi_2 \rightarrow [a_2]\Psi_2$$
$$((p = C_6) \wedge \Phi_6) \rightarrow [a_6]\Psi_6$$
$$((p = C_7) \wedge \Phi_7) \rightarrow [a_7]\Psi_7$$

y además se incluyen las fórmulas siguientes correspondientes al protocolo como proceso de prohibición:

$$(p = C_6) \rightarrow [a_6](p = C_7)$$
$$(p = C_7) \rightarrow [a_7](p = C_6)$$
$$\neg(p = C_6) \rightarrow [a_6]false$$
$$\neg(p = C_7) \rightarrow [a_7]false$$

Migración por valores de atributo Cuando la partición dinámica se expresa como un particionamiento del estado en función de los valores de los atributos, el proceso de cálculo de las propiedades asociadas a la superclase viene dado por:

i) Conjunto de atributos: $Atr = Atr_j \cup Atr_1 \cup ... \cup Atr_n$.

ii) Conjunto de eventos: $Ev = Ev_j \cup Ev_1 \cup ... \cup Ev_n$. Dado que un evento emergente en una subclase no debe estar accesible si el objeto no es instancia de dicha subclase de la partición dinámica, es necesario añadir un conjunto de fórmulas dinámicas que deshabiliten la ocurrencia de dicho evento (véase el siguiente ejemplo).

iii) El conjunto de fórmulas de evaluación es la unión de las fórmulas de evaluación de C_j y desde C_1 hasta C_n.

iv) El conjunto de fórmulas de prohibición es la unión de las fórmulas de prohibición de C_j y desde C_1 hasta C_n. Dada la restricción de compatibilidad de comportamiento, si C_i redefine una prohibición de C_j entonces prevalece la prohibición de C_i.

v) Las fórmulas de obligación se calculan de manera análoga a las fórmulas de prohibición.

vi) Cada una de las propiedades de comportamiento de cada clase C_i debe añadir, como condición de aplicación de la fórmula, la fórmula en lógica ecuacional usada en la definición de la partición dinámica (véase el ejemplo 7).

[3] En (Letelier et al., 1998) se ofrece con detalle el proceso de representación de los protocolos como fórmulas en lógica dinámica. Por un lado se obtienen fórmulas que determinan la transición entre estados del proceso. Por otro lado, se derivan fórmulas de permisos que establecen las secuencias válidas de acciones.

[4] Siendo p la variable que representa el estado actual del proceso migratorio y que se supone se ve modificada por fórmulas de transición.

vii) Repetir el proceso i) a vi) para la siguiente partición dinámica de C_j considerando como propiedades las recientemente calculadas.

Ejemplo 7 *Dada la partición dinámica definida por la especificación OASIS:*

```
C6 where {atr1<0},
C7 where {atr10}
dynamic specialization of C2
```

donde además se tienen las propiedades en lógica dinámica siguientes:

$$\Phi_2 \rightarrow [a_2]\Psi_2 \qquad \text{para la clase } C_2$$
$$\Phi_6 \rightarrow [a_6]\Psi_6 \qquad \text{para la clase } C_6$$
$$\Phi_7 \rightarrow [a_7]\Psi_7 \qquad \text{para la clase } C_7$$

entonces, al trasladar estas propiedades a la clase C_2las fórmulas anteriores se sustituyen por las siguientes:

$$\Phi_2 \rightarrow [a_2]\Psi_2$$
$$((atr_1 < 0) \wedge \Phi_6) \rightarrow [a_6]\Psi_6$$
$$((atr_1 0) \wedge \Phi_7) \rightarrow [a_7]\Psi_7$$

y además se añaden las fórmulas que deshabilitan cada uno de los eventos emergentes cuando el objeto no es instancia de la subclase dinámica en la que están definidos, es decir:

$$\neg(atr_1 < 0) \rightarrow [a_6]false$$
$$\neg(atr_1 0) \rightarrow [a_7]false$$

4.3 Resumen proceso de traducción

El proceso de traducción a fórmulas en Lógica Dinámica para las especificaciones de clase de *OASIS* (que incorporen particiones estáticas y dinámicas) se resume en los siguientes pasos:

- El proceso se aplica a una jerarquía completa de herencia.
- Se comienza por las particiones dinámicas (si existen) de más bajo nivel en la jerarquía. Se trasladan las propiedades especificadas en las subclases de la partición dinámica a la superclase y así, sucesivamente, hasta llegar a una superclase que sea subclase de una partición estática o bien sea la raíz de la jerarquía (en cuyo caso termina el proceso).
- Cuando todas las particiones dinámicas han sido sustituidas entonces se determinan las propiedades totales para cada clase hoja de la jerarquía de herencia que sólo contiene particiones estáticas (o bien la clase raíz).
- A partir de las propiedades de las clases hoja se determinan las propiedades de cualquier especie perteneciente a la jerarquía. Este aplanamiento de toda la jerarquía en especies individuales es la representación final por considerar dichas especies como clases independientes al mismo nivel que el resto de clases del esquema conceptual.

5 Conclusión y trabajo futuro

OASIS es un lenguaje de especificación de sistemas de información que sigue el enfoque orientado a objetos. Mediante la herencia podemos especializar las propiedades definidas en las clases. Utilizamos particiones estáticas en aquellas especializaciones que son fijas desde la creación del objeto. Por otro lado, definimos particiones dinámicas para expresar que la partición a la que pertenece el objeto no es fija y que depende de la ocurrencia de eventos específicos o de los valores de los atributos en un determinado instante. Los cambios de una instancia entre subclases de particiones dinámicas corresponde con lo que se denomina proceso de migración.

Este trabajo presenta las pautas para trasladar las particiones definidas en una jerarquía de herencia a clases de más bajo nivel (llamadas especies) en las que no existe subclasificación alguna. Para el proceso de traducción se utiliza el mismo marco formal de Lógica Dinámica en el que está basado *OASIS*.

Actualmente se dispone de un traductor de los conceptos de clase de *OASIS* a entornos de concurrencia (Programación Lógica Concurrente y Redes de Petri). Dicho traductor está integrado en un ambiente de animación (Letelier et al., 1999b) donde el proceso de captación de requisitos está basado en especificación y validación incremental de escenarios (Rolland et al., 1996). A partir de las pautas presentadas en este trabajo la idea es ampliar el proceso de traducción incluyendo herencia y reutilizando el entorno de animación comentado.

En *OASIS* se pueden definir roles asociados a una clase. Dado que el rol de un objeto es un nuevo objeto, que la clase de rol representa la raíz de una nueva jerarquía de herencia y que existe compatibilidad de signatura (al menos) entre el rol y quien lo desempeña, el tratamiento de los roles no es tan inmediato. Se hace necesario incluir mecanismos de comunicación entre el rol y el *player*, de manera que se contemple la concurrencia (intra e inter-objeto) inherente a los objetos del modelo. Pensamos que la solución a los roles vendrá junto con el tratamiento de clases agregadas, donde se hace necesaria la comunicación síncrona entre objetos. En cualquier caso existirá un vínculo de comunicación entre el rol y el *player* (que supone el sustento básico para abordar la delegación inherente al mecanismo de

rol). Otro aspecto importante es ofrecer un soporte adecuado para considerar el polimorfismo existente en el modelo. Dicho de otro modo, una acción en una especificación *OASIS* puede hacer referencia a una clase servidora que no sea necesariamente una especie. En este caso el entorno debe determinar las propiedades que a más bajo nivel tiene el servidor.

References

Du Bois P. The Albert II Reference Manual Language, Constructs and Informal Semantics. Research Report RR-97-002, Computer Science Department, University of Namur, 1997. ftp://ftp.info.fundp.ac.be

Grau A., Kuester Filipe J., Kowsari M., Eckstein S., Pinger R., Ehrich H.-D. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In Ling T.W., Ram S. and Lee M.L. (Eds.) Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore, pages 277-290, Springer, LNCS 1507, November 16-19, 1998.

Harel D. Dynamic Logic in Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner, pages 497-694, Reidel 1984.

Letelier P., Ramos I., Sánchez P., Pastor O. *OASIS* 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto, Servicio de Publicaciones de la Universidad Politécnica de Valencia (SPUPV-98.4011), ISBN 84-7721-663-0, Universidad Politécnica de Valencia, 1998.

Letelier P., Sánchez P., Ramos I. Prototyping a requirements specification through an automatically generated concurrent logic program. First International Workshop on Practical Aspects of Declarative Languages (PADL'99), Springer-Verlag, LNCS 1551, pages 31-45, Texas (USA), January 1999.

Letelier P., Sánchez P., Ramos I. Un Ambiente para Especificacion Incremental y Validacion de Modelos Conceptuales. Actas del 2°Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software, IDEAS'99, páginas 216-228, san José (Costa Rica), Marzo de 1999.

Letelier P., Sánchez P., Ramos I. Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales, VI Congreso Internacional de Investigación en Ciencias Computacionales, páginas 80-91, Cancún (Méjico), Septiembre de 1999.

Letelier P., Sánchez P., Ramos I. Animation of Conceptuals Models using two Concurrent Environments: An overview. In Software and Hardware Engineering for the 21st Century, pages 389-396, Ed. World Scientific and Engineering Society Press, Massachussets (USA), 1999.

Meyer J.-J.Ch. A Different Approach to Deontic Logic: Deontic Logic viewed as a Variant of Dynamic Logic. In Notre Dame Journal of Formal Logic, vol. 29, pages 109-136, 1988.

OBLOG Software S.A. The OBLOG Software Development Approach (White Paper), 1999. http://www.oblog.pt

Pernici B. Objects with Roles. In EEE/ACM Conference on Office Information Systems, Cambridge, Mass., 1990.

Rolland C., Ben Achour C., Cauvet C., Ralytè J., Sutcliffe A., Maiden N.A.M., Jarke M., Haumer P., Pohl K., Dubois E., Heymans P. A Proposal for a Scenario Classification Framework, Technical Report CREWS 96-01. http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm.

Rumbaugh J., Jacobson I., Booch G. Unified Modeling Language Reference Manual, Addison Wesley, 1997.

Sánchez P., Letelier P., Ramos I. Constructs for prototyping information systems with object Petri Nets. Proceeding of the IEEE International Conference on SMC'97, vol. 5, pages 4260-4265, Orlando (USA), 1997.

Sánchez P., Letelier P., Ramos I., Pastor Ó. Modelado Conceptual con un Lenguaje Formal y Orientado a Objeto. Actas de la XXV Conferencia Latinoamericana de Informática (CLEI'99), Asunción, páginas 417-430, Agosto de 1999.

Siddiqi J., Morrey I.C., Roast C.R., Ozcan M.B. Towards quality requirements via animated formal specifications. Annals of Software Engineering, num. 3, 1997.

Taivalsaari A. On the Notion of Inheritance. ACM Computing Surveys, vol. 28, num. 3, pages 438-479, September, 1996.

Wieringa R., Jonge W., Spruit P. Using Dynamic Classes and Role Classes to Model Object Migration. Theory and Practice of Object Systems, vol. 1, num. 1, pages 61-83, 1995.



Pedro Sánchez Palma es Licenciado en Informática por la Universidad de Málaga (España) en 1992. Doctor en Informática por la Universidad Politécnica de Valencia (España) en el 2000. Profesor Titular del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia (UPV) y miembro del Grupo de Investigación "Programación Lógica e Ingeniería del Software" de la UPV. Sus áreas de interés son ingeniería del software y modelado orientado a objetos.



Patricio Letelier Torres es Ingeniero Civil en Informática por la Universidad Técnica Federico Santa María de Valparaíso (Chile) en 1990. Doctor Ingeniero en Informática por la Universidad Politécnica de Valencia (España) en 1999. Profesor de la Universidad Politécnica de Valencia (UPV) en el Departamento Sistemas Informáticos y Computación, siendo miembro del Grupo de Investigación "Programación Lógica e Ingeniería del Software" de la UPV. Sus áreas de interés son la ingeniería de requisitos y modelado orientado a objetos.



Isidro Ramos Salavert es Licenciado en Ciencias Físicas por la Universidad de Valencia (España) en 1966, Licenciado en Informática por la Universidad Politécnica de Madrid (España) en 1972, Doctor en Ciencias Físicas por la Universidad Complutense de Madrid (España) en 1971. Área de interés: modelado conceptual de sistemas de información. Actualmente es Catedrático de Universidad y director del Grupo de Investigación "Programación Lógica e Ingeniería del Software" de la Universidad Politécnica de Valencia.

Animating formal specifications with inheritance in a DL-based framework[1]

Pedro Sánchez **Patricio Letelier** **Isidro Ramos**

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera, s/n, 46071 Valencia
email {ppalma | letelier | iramos}@dsic.upv.es

http://www.dsic.upv.es/users/oom
Keywords: Requirements engineering, object oriented methods, inheritance, animation of specifications.

Abstract

Dynamic Logic (DL) provides a suitable formal framework to model actions and reasoning about them. OASIS is a language for the specification of object oriented conceptual models. In our model, specialization is a relation between classes that defines an inheritance mechanism through static and dynamic partitions. A variant of DL (including the Deontic operators for permission, prohibition and obligation) is the formalism used in OASIS to deal with changes of state, triggers, preconditions, protocols and operations. The animation of conceptual models in order to validate the specification is an interesting topic. We have worked on translating OASIS specifications automatically to concurrent environments in order to obtain a prototype useful to validate specifications by animation. The aim of this paper is to show that it is feasible to translate static and dynamic partitions automatically into dynamic logic formulae. Thus, using the same developed schema of animation it is possible to execute OASIS specifications including inheritance.

1 Introduction

Conceptual models, representing the functional requirements of information systems, are a key factor when linking problem and solution domains. Building a conceptual model is a discovery process, not only for analysts but also for stakeholders. The most suitable strategy in this situation is to build conceptual models in an iterative and incremental way, through analyst and stakeholder interaction. Conceptual modeling involves four activities: elicitation of requirements, modeling or specification, verification of quality and consistency, and eventually, validation.

[1] This research is supported by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) through the MENHIR proyect (grant no. TIC97-0593-C05-01).

Formal methods for conceptual modeling provide improvements in soundness and precision for specifications and help in their verification. However, when considering elicitation and requirements validation, prototyping techniques are more often used. Hence, it is interesting to obtain a combination of both approaches.

This work uses *OASIS* [10,7] (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems), a formal approach for object oriented conceptual specification of information systems. This is a step forward in a growing research field where the validation of formal specifications through animation is being explored [16]. In this sense, some other proposals close in nature to *OASIS* and to the work we are carrying out using *OASIS* are TROLL [5], ALBERT [6] and OBLOG [13]. The differences between these works and ours are basically determined by features of the underlying formalisms and the offered expressiveness. According to the presented results, the state of art is similar and is characterized by preliminary versions of animation environments.

Validation and verification play an important role in the quality of the final product. By means of validating formal specifications stakeholders can fasten the correspondence between formal specifications and user prospects. The verification process allows to verifying the correctness between implementations and specification. Errors can be propagated towards the design and implementation process if the validation and/or verification are not done exhaustively. Basically, validation and verification techniques can be classified in two groups:

- Statics: reasoning about system properties regarding a set of predefined rules.
- Dynamics: executing some kind of implementation of the system.

As Feenstra points out in [1] two approaches are considered when animating specifications: to reason about scenarios (what requires an automatically generated prototype obtained from formal specifications) and to reason about reachability properties (what requires software capable of solving queries automatically). In our work the animation process focuses on the first approach showing the effects of actions in the system by testing scenarios.

Figure 1 shows a framework based on *OASIS* for elicitation, modeling, verification and validation of requirements. Elicitation is achieved by using scenarios [14]. The elements and expected behavior of a given specification are extracted by analysts from scenarios. Functional requirements are modeled using a graphical specification module based on *OASIS*. Conceptual models can be verified according to *OASIS* formal properties. At each stage of the requirements specification process it would be possible to validate the behavior of the associated prototype against the expected behavior. This comparison could lead to updates or extensions of existing scenarios. This cycle continues until the requirements are compliant with the proposed set of scenarios.
Experiments have been carried out using Object-Oriented Petri Nets [15] and Concurrent Logic Programming [8] as concurrent environments for *OASIS* specifications. Correspondences between *OASIS* and these environments have been included in a translator program. This translator takes an *OASIS* specification stored in the repository and generates automatically a Concurrent Logic Program or a Petri Net that constitute a prototype for the corresponding conceptual model. Furthermore, through a preliminary version of the graphical animation environment, the analyst can interact with the prototype in a suitable way. We obtain CodeSign code automatically

from *OASIS* specifications. This work is being integrated into a CASE tool for system modeling supporting the *OASIS* approach. We have addressed validation through animation focusing on *OASIS* but this work could be extended to other similar languages. Basically, the obtained prototypes allow to animate sets of a variant of Dynamic Logic formulae [12] (including permissions, valuations and obligations) representing *OASIS* specifications. Our aim is to be able to animate (using the same translator) *OASIS* specifications including inheritance.

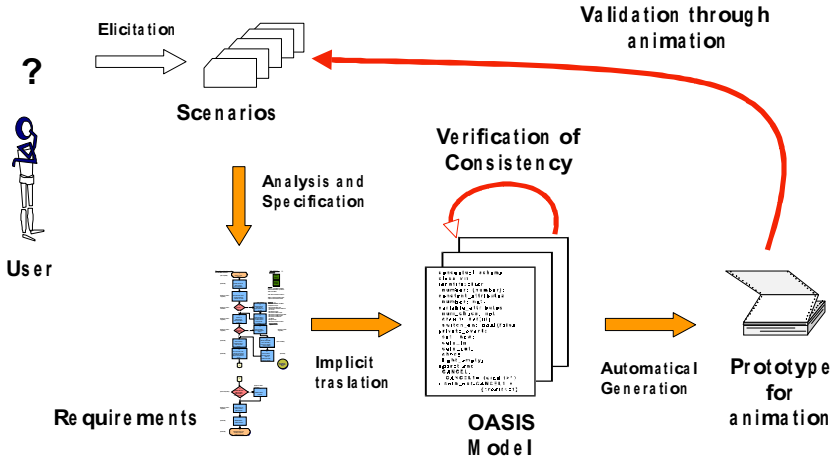


Figure 1: Animation Environment.

Nowadays, an increasing work in validation through animation of conceptual models has been developed. Many CASE tools have integrated an special module for animation. Most of tools related to requirements validation do not offer much help for animation. OBLOG[2] has an animation module but it is not still available. In Rhapsody[3] and ObjectTime[4] a graphical representation (a sequence diagram) is included as a result of the animation and interaction between objects. In this way, Aonix[5] offers a module called "Animator" in which the edition and execution of scenarios are supported. In BridgePoint[6] validation is done by monitoring in several levels the generated prototype but it is not easy to contrast between the obtained and expected results. In [3] an environment about TROLL in which is possible to study the change in the state of objects is showed. In a similar way, in [6] the architecture and functionality of an animation module for ALBERT specifications is presented. In all these proposals near to *OASIS*, the state of the art is clearly characterized by preliminary versions of animation modules. The aim of this paper is to show that is feasible to translate automatically the *OASIS* specialization in *DL* formulae. Thus using the same developed schema of animation it is possible to execute *OASIS* specifications including inheritance.

[2] http://www.oblog.pt

[3] http://www.ilogix.com

[4] http://www.objectime.com

[5] http://www.aonix.com

[6] http://www.projtech.com

The rest of this paper is organized as follows: in section 2 we introduce the basic concepts of *OASIS*. Section 3 gives the formal framework of specialization in *OASIS*. Section 4 gives a short description of the used *DL*. Section 5 establishes a representation mechanism for specialization. Section 5 summarizes the paper and outlines further work. Finally, an appendix introduces a full case study.

2 Basic concepts of *OASIS*

An *OASIS* specification is a presentation of a theory in the used formal system and is expressed as a structured set of **class** definitions. Classes can be simple or complex. A complex class is defined in terms of other classes (simple or complex) by establishing relationships among classes. These relationships provide **aggregation** or **inheritance** mechanisms. A class has a name, one or more identification mechanisms for its instances (**objects**) and a type or template that is shared by every instance belonging to the class. Each object has an unique identifier (*oid*) set by the system, however, objects are referred by their identification mechanisms belonging to the problem space. A function establishes a mapping between the identification mechanisms and the *oid*. The type or template describes the structure and behavior of every object.

Thus each object encapsulates its own state and behavior rules. As usual in object oriented environments, objects can be seen from two points of view: static and dynamic. From the static perspective, the attributes are properties describing the object structure. The object state in a given instant is the set of structural properties values. From the dynamic perspective, the evolution of objects is characterized by the "change of state" notion. The occurrence of actions implies changes (by means of **valuations** and **derivations**) in the values of the attributes. Object activity is determined by a set of rules: **preconditions** (as forbidden actions in certain states), **triggers** (as obligations to be fulfilled in certain states), **protocols** (as allowed sequences of actions in object life) and **operations** (as obligated sequences of actions). A **step** is the set of actions executed at the same instant by the object.

Inheritance is a mandatory characteristic of the object oriented paradigm. Inheritance is a mechanism through which subclasses inherit properties of superclasses. The inheritance mechanism is an unquestionable help in incremental construction and reuse, but researches rarely agree on its meaning and usage [17]. In *OASIS*, inheritance is used in a disciplined way under the concept of **specialization**.

3 Specialization in *OASIS*

In *OASIS* to specialize a class means to create one or more partitions for it. Each partition is a set of new classes that divides the original one taking into account some criterions, thus many partitions can coexist. One object, in a given instant, is instance of only one subclass in every partition. Next we give the characteristics of each kind of specialization in *OASIS*, it is, **static partitions** and **dynamic partitions**.

Definition 1 ***Subclasses and superclasses.*** *When a subclass is a specilization of another class, the former is said to be a subclass and the latter a superclass. The template of a subclass is derived from the templates of other classes (superclasses).*

3.1 Static partitions

A static partition [18] divides completely the possible instances set of the partitioned class into disjoints subsets. Thus, static partitions divide the space of objects. Each object is permanently created as an instance of a given subclass in static partitions.

Each instance of a given subclass is at the same time one instance of its superclass and vice versa.

Example 1 Two static partitions of class `vehicle`:

```
truck, car, other
       static specialization of vehicle;
gas, diesel, other
       static specialization of vehicle;
```

3.2 Dynamic partitions

A dynamic partition [18] includes subclasses to which an object can belong during its lifetime. Object migration between classes is produced by the occurrence of actions. Thus, a dynamic partition divides the possible states of objects allowing to objects to change their subclasses.

From the point of view of the *OASIS* language, we can specify the transition between subclasses in a partition in two manners (being equivalent): by means of a migration process (related to occurring actions) or by attribute values.

Example 2 A dynamic specialization of class `car` produced by the occurrence of the actions `new_car`, and `repair_car` in *OASIS*:

```
working, broken_down dynamic
specialization of car
       migration relation is
  car = new_car.working;
  working = break_down.broken_down;
  broken_down = repair.working;
```

The creation of a `car` instance implies that it starts belonging to the class `working`. As an instance of `working`, actions from `car` and `working` templates can be recognized. The action `break_down` implies to leave the subclass `working` and to migrate towards `broken_down` class.

From a theoretical point of view, the process representing the life of an instance of `car` is the joint of processes defined in every subclass. The connections among subclasses are given by the actions included in the migratory process.

Example 3 A dynamic partition of the class `person` under the attribute approach:

```
child where {age < 13}
teenager where {13 < = age and age < = 19}
```

```
adult where {19 < age}
dynamic specialization of person;
```

In this case, whenever the attribute `age` changes, depending on the established conditions one instance of person could migrate between subclasses.

Eventually, in *OASIS* is not allowed to define static partitions from dynamic ones.

3.3 Species and multiple inheritance in *OASIS*

The Cartesian product between leaf classes of a given specialization hierarchy gives the set of **species** mixing all properties of those classes. Furthermore, emergent properties can be specified in their templates.

Example 4 Species of the hierarchy `vehicle`:

```
truck*diesel, car*gas, car*diesel
other*diesel, other*gas, truck*gas
```

Example 5 The class `truck*diesel` may have an emergent attribute named `tonnage`. This attribute must be specified in the `truck*diesel` template.

4 *OASIS* formalized in *DL*

Dynamic logic (DL) [4] has been traditionally considered to describe and reason about dynamic systems. The aim of this logic is the study of mathematical properties of programs and their behaviour. An object in *OASIS* is able to pass from one state to another. Due to the state change, the truth values of the formulas describing the state also change. The objective in our work is to define a logical basis to be able to express our reasoning about *OASIS* specifications. To each *OASIS* object we associate an accessibility relation in such a way that a pair of state (s,t) is in that relation if and only if there is a computation of the program (execution of a set of actions by the object) transforming the state s into the state t.

In [12] Deontic Logic is described as a variant of *DL* [4]. The definition of deontic operators in *DL* is:

$O(a) \Leftrightarrow [\neg a]$ false	"the occurrence of a is obligatory".
$F(a) \Leftrightarrow [a]$ false	"the occurrence of a is forbidden".
$P(a) \Leftrightarrow \neg F(a)$	"a is permitted if and only if a is not forbidden".

Where *false* denotes no reachable world. It is usual to find in related works the special atom V of violation representing a problematic or "non-ideal" situation. In our work the atom false expresses the impossibility of performing a transition.

We are using a sublanguage of the language presented by Meyer in [12] including the following kind of formulae:

$\psi \rightarrow [a]false$	"the occurrence of *a* is forbidden
	in states where ψ is satisfied"
$\psi \rightarrow [\neg a]false$	"the occurrence of *a* is obligatory
	in states where ψ is satisfied"
$\psi \rightarrow [a]\phi$	"immediately after *a* occurrence,
	ϕ must be satisfied
	in states where ψ is satisfied"

These formulae are named **prohibition**, **obligation** and **valuation**, respectively. Additionally, ψ is a well-formed formula that characterizes the state of an object when the action *a* occurs and $\neg a$ represents the non-occurrence of the action *a* (i.e. only other actions different from *a* could occur). Furthermore, there is no state satisfying the atom *false*. Thus, one action is forbidden if its occurrence leads the system towards a violation state, and one action is obligatory if its non-occurrence leads the system towards a violation state.

An *OASIS* class template is represented by the tuple $\langle Atr, Ev, Formulae, Processes \rangle$, that is, attributes, events, *DL* formulae (valuations, prohibitions and obligations) and process specifications (protocols and operations). In [9] it is showed how Processes can also be interpreted as a set of *DL* formulae. Thus, the class template can be seen as $\langle Atr, Ev, DLForm \rangle$ where *DLForm* includes all *DL* formulae establishing the behavior of objects. Next section details how, through a translation process, static and dynamic partitions can be integrated in the same formal framework of *DL*.

5 Specialization in *DL*

As mentioned in previous sections species involve classes in the same hierarchy. Likewise, the classes constituting species have a common ancestor class which has been partitioned more than once. Figure shows a specialization hierarchy with the following species:

$C_3 * C_1$ $C_3 * C_6$ $C_3 * C_7$
$C_4 * C_1$ $C_4 * C_6$ $C_4 * C_7$
$C_5 * C_1$ $C_5 * C_6$ $C_5 * C_7$

But, for instance, the following are not species: $C_3 * C_4$,$C_1 * C_7$, $C_3 * C_5 * C_1$, etc.

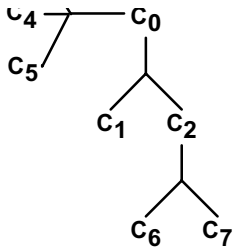


Figure 2: Specialization hierarchy.

We are interested in establishing the properties of species because in *OASIS* objects are always instances of species. In order to calculate these properties we will see some simplified case studies. Next we will analyze the equivalent in *DL* of any static partition.

5.1 Static partition hierarchies

Given a specialization hierarchy with only static partitions the set of properties describing the behavior of any leaf class in the hierarchy is the union of properties from root class to the leaf class. In static and dynamic partitions of *OASIS* we have behavioral compatibility between superclasses and subclasses. Thus, the set of properties of any leaf class C_i is given as follows:

i. Set of attributes:

$$Atr_i = \cup Atr_k \qquad \forall C_k = C_i \text{ hasta } C_k = C_{root}$$

ii. Set of events:

$$Ev_i = Ev_k \qquad \forall C_k = C_i \text{ hasta } C_k = C_{root}$$

iii. Valuation formulae in subclasses can only modify emergent attributes[7] (and also attributes without valuations in superclasses), thus the set of valuation formulae is the union of valuations from leaf class to root class (i.e. the *DL* formulae union).

iv. If a prohibition formula is modified in the subclass then the new condition must imply the superclass condition, thus the set of properties is the union of the properties from leaf class to root class (i.e. the *DL* formulae union). In this way, only are considered the most specific preconditions.

v. Obligation formulae are established in the same way as prohibition formulae.

After deriving both structural and behavioral properties from every leaf class, it is immediate to calculate the properties of species.

Definition 2 ***Template of a species*** *$(C_0 * ... * C_n)$ is a class template with the following elements:*

$$Atr_0 \cup ... \cup Atr_n \cup Atr_e$$

[7] That is, attributes defined in the subclass specification.

$$Ev_0 \cup ... \cup Ev_n \cup Ev_e$$

$$DLForm_0 \cup ... \cup DLForm_n \cup DLForm_e$$

where Atr_i, Ev_i and $DLForm_i$ are attributes, events and *DL* formulae of class C_i; Atr_e, Ev_e and $DLForm_e$ are the added attributes, events and *DL* formulae being specified for the species $(C_0 * ... * C_n)$. In case of overlapping of properties the calculus of properties follows the constraints previously mentioned.

5.2 Dynamic partition hierarchies

Whenever dynamic partitions are included, the process to determine properties of species is slightly different.

The idea is to translate all the subclass properties in every dynamic partition into an ascending direction. Figure 3 shows an example in which class C_2 has been dynamically partitioned. Thus we want to extend the dynamic formulae of class C_2 of Figure 3 separating the formulae due to C_6 from those due to C_7. Next specific considerations when deriving the *DL* formulae set are described here.

Migration by action occurrence

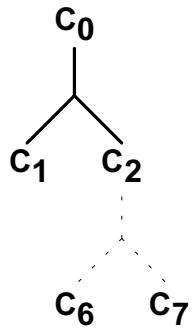


Figure 3: Dynamic partition.

Given the dynamic partition $\{C_1,...,C_n\}$ of the class C_j, then the template of C_j including properties of subclasses for each class C_i belonging to the partition, is given as:

i. Set of attributes: $Atr = Atr_j \cup Atr_1 \cup ... \cup Atr_n$.
ii. Set of events: $Ev = Ev_j \cup Ev_1 \cup ... \cup Ev_n$.
iii. The set of valuations formulae is the union of valuation formulae of classes C_j and C_1 to C_n. Each valuation must have a condition that controls when it is used.
iv. The set of prohibition formulae is the union of prohibition formulae from the classes C_j and C_1 to C_n. Taking into account the behavior compatibility restriction, if C_i redefines a prohibition of C_j then the condition of C_i prevails.
v. Obligation formulae are established in the same way as prohibition formulae.
vi. Migration processes defined in partitions will be included in class C_j as a protocol containing valid action sequences. Prohibition formulae are directly incorporated into the set of prohibition formulae of C_j. However the behavior

property of the class C_i must add an occurrence predicate that evaluates the current state of the process. Thus, this such a protocol will be translated into prohibition formulae and valuation formulae in the same way as protocols.

vii. Repeat the procedure from i) to vi) for each dynamic partition of C_j.

Example 6 Given the following dynamic partition in *OASIS*:

```
C6, C7
dynamic specialization of C2
migration relation is
  C2new.C6
  C6a6.C7
  C7a7.C6
```

where we also have the following valuation formulae:

$\Phi_2 \rightarrow [a_2]\Psi_2$ in class C_2
$\Phi_6 \rightarrow [a_6]\Psi_6$ in class C_6
$\Phi_7 \rightarrow [a_7]\Psi_7$ in class C_7

then, when translating these properties into the class C_2 , the previous formulae are substituted by the following ones[8]:

$\Phi_2 \rightarrow [a_2]\Psi_2$
$((p = C_6) \wedge \Phi_6) \rightarrow [a_6]\Psi_6$
$((p = C_7) \wedge \Phi_7) \rightarrow [a_7]\Psi_7$

and the following formulae are included:

$(p = C_6) \rightarrow [a_6](p = C_7)$
$(p = C_7) \rightarrow [a_7](p = C_6)$
$\neg(p = C_6) \rightarrow [a_6]false$
$\neg(p = C_7) \rightarrow [a_7]false$

Migration by attribute value

In this case *DL* formulae are now defined as follows:

i. Set of attributes: $Atr = Atr_j \cup Atr_1 \cup ... \cup Atr_n$.
ii. Set of events: $Ev = Ev_j \cup Ev_1 \cup ... \cup Ev_n$. Taking into account that an emergent event of a subclass must not be available if the object is not instance of that subclass, then it is necessary to add a *DL* formulae that disable the occurrence of that event (see next example).
iii. The set of valuation formulae is the union of valuation formulae of classes C_j and C_1 to C_n.

[8] Where p is a variable representing the current state of the migration process. Thus p is modified by valuation formulae.

iv. The set of prohibition formulae is the union of prohibition formulae of classes C_j and C_1 to C_n. Taking into account the behavior compatibility restriction, if C_i redefines a prohibition of C_j then the condition of C_i prevails.
v. Obligation formulae are established in the same way as prohibition formulae.
vi. Each formula obtained from C_i must add an occurrence predicate referring to the current state of the object.
vii. Repeat from i) to vi) for each dynamic partition of C_j.

Given the following dynamic partition in *OASIS*:

```
C6 where {atr1 < 0},
C7 where {atr1 >= 0}
dynamic specialization of C2
```

where we also have the following valuation formulae:

$\Phi_2 \rightarrow [a_2]\Psi_2$ in class C_2
$\Phi_6 \rightarrow [a_6]\Psi_6$ in class C_6
$\Phi_7 \rightarrow [a_7]\Psi_7$ in class C_7

then, when translating these properties into the class C_2, the previous formulae are substituted by the following ones:

$\Phi_2 \rightarrow [a_2]\Psi_2$
$((atr_1 < 0) \wedge \Phi_6) \rightarrow [a_6]\Psi_6$
$((atr_1 >= 0) \wedge \Phi_7) \rightarrow [a_7]\Psi_7$

and the following formulae are included:

$\neg(atr_1 < 0) \rightarrow [a_6]false$
$\neg(atr_1 >= 0) \rightarrow [a_7]false$

5.3 Translation process summary

The translation process from class specification with specialization to *DL* formulae is summarized in the following steps:

- The translation process is applied for each specialization hierarchy.
- The translation process in dynamic partitions (if they exist) starts from the bottom level. All properties defined in dynamic subclasses are added to the superclass. The translation process continues until reaching a static partition or until reaching the top class in the hierarchy (in this case, the translation process finishes).
- When all dynamic partitions have been substituted the properties for each leaf class containing only static partitions are determined.
- Eventually, properties for species can be determined from leaf class properties. This hierarchy flattening in separate species is the final representation. Thus, all species will be independent classes at the same level and without specialization relationships.

5.4 Animation support

From an ideal point of view the representation in *DL* formulae finishes when all properties of species are specified. However, we also have to give a suitable support for taking into account the existing polymorphism in our *OASIS* model. Thus, an action in *OASIS* may reference a server class not being a species. In this situation the system must determine the full properties at the lower level. This implies a separation in two levels in our animation environment. On the one hand we have objects whose behavior is given by one species in which they were created. On the other hand we also have a class representing each original hierarchy class. In this additional class we keep a reference for each object belonging to it[9].

Figure 4 shows how every class is considered at the same level (without hierarchy) and objects are associated to species. However, there are mappings from each class to their object instances. For example obj_1 is an instance of class A and B and it is also instance of all species like B∗, etc.

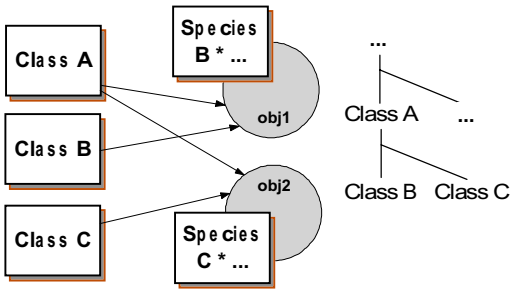


Figure 4: Representation example.

6 Conclusion

OASIS is a formal approach for the specification of object oriented conceptual models. In *OASIS* conceptual schemes of information systems are represented as societies of interacting concurrent objects. Animating such models in order to validate the specification of information systems is a topic of interest in requirements engineering.

Using inheritance we can specialize (or generalize) properties defined in classes. Specialization is an important modeling mechanism. There are two kinds of specializations in *OASIS*: by defining static and dynamic partitions. Objects in static partitions belong to a given specialized class during their whole lifetime. Besides, objects in dynamic partitions can migrate from one subclass to another one. The migration between subclasses of the same partition may be due to action occurrence or change in attribute values. A process migration represents the possible transitions among subclasses that an object can do. In addition, class templates of *OASIS* can be expressed as a set of *DL* formulae.

Static and dynamic specialization constructs in conceptual modeling allow expressing directly some patterns. For example, in [2] the dynamic classification is implemented

[9] A mapping between identifying mechanisms and *oids*.

using the pattern named *state* and the role mechanism may be implemented using the pattern named *role object*.

This paper gives the required steps to translate static and dynamic partitions in the same framework of *DL* used for class template. Thus, an *OASIS* specification with specialization can be interpreted automatically as an equivalent specification without specialization in our *DL*-based framework. In this way, the current animation environment, working with valuation, prohibition and obligation formulae, can be used to animate specifications with specialization.

We have built a translator program to obtain a prototype from *OASIS* specifications automatically. The translator should be extended in order to include the mappings established in this paper. This work is being integrated into a CASE tool for system modeling supporting the *OASIS* approach and the methodology OO-METHOD [11] defined for *OASIS*.

References

[1] Feenstra R.B. and Wieringa R.J. *Validating Specifications of Dynamic Systems using Automated Reasoning Techniques*. In J.C. Bioch and Y.-H. Tan (Eds.), Proceedings of the Seventh Dutch Conference on Artificial Intelligence, NAIC'95, pages 105-114, 1995.

[2] Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns: elements of Reusable Object Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[3] Grau A. and Kowsari M. A. *Validation System for Object-Oriented Specifications of Information Systems*. In Manthey R. and Wolfengagen V. (Eds.), Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97), St. Petersburg, Electronic Workshops in Computing, Springer, September 2-5, 1997.

[4] Harel D. *Dynamic Logic*. In Handbook of Philosophical Logic II, editors D.M. Gabbay, F. Guenthner, pages 497-694, Reidel 1984.

[5] Grau A., Kuester Filipe J., Kowsari M., Eckstein S., Pinger R. and Ehrich H.-D. *The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools*. In Ling T.W., Ram S. and Lee M.L. (Eds.) Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore, pages 277-290, Springer, LNCS 1507, November 16-19, 1998.

[6] Heymans P. *The Albert II Specification Animator*. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm.

[7] Letelier P., Ramos I., Sánchez P. and Pastor Ó. *OASIS 3.0: A Formal Approach to Object-Oriented Conceptual Modeling*. Servicio de Publicaciones de la Universidad Politécnica de Valencia (SPUPV-98.4011), ISBN 84-7721-663-0, 1998. (in Spanish)

[8] Letelier P., Sánchez P. and Ramos I. *Prototyping a requirements specification through an automatically generated concurrent logic program*. Gupta (Ed.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science LNCS 1551, pages 31-45, Springer-Verlag, 1998.

[9] Letelier P., Sánchez P. and Ramos I. *Process Specification for Objects interpreted into Dynamic Logic*. III Jornadas de Ingeniería de Software (JIS'98), Murcia, pages 281-292, 1998. (in Spanish)

[10] Pastor O. and Ramos I. *OASIS versión2(2.2): A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*. Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV-95.788, 1995.

[11] Pastor O., Insfrán E., Pelechano V., Romero J., and Merseguer J. *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods*. Proceedings of Conference on Advanced Information Systems Engineering, CAiSE '97, pages 145-158, Barcelona, 1997.

[12] Meyer J.-J.Ch. *A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic*. In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.

[13] OBLOG Software S.A. *The OBLOG Software Development Approach*. (White Paper). 1999, http://www.oblog.pt/Download/Documentation.exe

[14] Rolland C., Ben Achour C., Cauvet C., Ralyté J., Sutcliffe A., Maiden N.A.M., Jarke M., Haumer P., Pohl K., Dubois E. and Heymans P. *A Proposal for a Scenario Classification Framework*. Technical Report CREWS 96-01, http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm.

[15] Sánchez P., Letelier P. and Ramos I. *Constructs for prototyping information systems with object Petri Nets*. Proceeding of the IEEE International Conference on SMC'97, pages 4260-4265, Orlando, 1997.

[16] Siddiqi J., Morrey I.C., Roast C.R. and Ozcan M.B. *Towards quality requirements via animated formal specifications*. Annals of Software Engineering, n.3, 1997.

[17] Taivalsaari A. *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, N. 3, September 1996.

[18] Wieringa R., Jonge W. and Spruit P. *Roles and dynamic subclasses: a modal logic approach*. Vrije U., The Netherlands, 1995.

A An example

In this example a vending machine (vm) accepts coins (coin_in) increasing the customer credit (credit). This credit decreases (coin_out) when a product is given (give). The machine also allows canceling the operation (CANCEL) and then returning all the stored coins. The machine has a warning light (light_empty) that is switched on if the action switch_on occurs. The *OASIS* 3.0 specification of vm is the following:

```
class vm
identification
 number: (number);
constant attributes
 number :nat;
variable attributes
 credit :nat(0);
 light_empty :bool(false);
events
 set new;
 coin_in;
 coin_out;
 light_empty;
 give;
valuations
 [::switch_on]light_empty:=true;
 [coin_in]credit:=credit+1;
 [::coin_out]credit:=credit-1;
 [give]credit:=credit-1;
preconditions
 CANCEL if {credit > 0};
 give if {credit > 0}
operations
 CANCEL:
 CANCEL1= {credit > 1}::coin_out.CANCEL1
  + {credit=1}::coin_out;
end class
```

This machine is specialized in chocolate machines (choc_vm) and others (other_vm). This is a static partitioning, thus the *OASIS* specification is the following:

```
choc_vm, other_vm
        static specialization of vm;
```

An object belonging to the class choc_vm has chocolates as products and their quantity is num_chocs. This quantity is reduced by one unit when giving a chocolate (give). It is not possible (a prohibition) to obtain a chocolate if there is neither credit nor chocolate. In *OASIS* we have:

```
class choc_vm
variable attributes
 num_chocs nat(0);
valuations
 [give] num_chocs:=num_chocs-1;
preconditions
 give if {credit > 0 and num_chocs > 0}[10]
end class
```

We will now consider a dynamic partitioning. The idea is to make a distinction between chocolate machines with products (with_choc) and without products (no_choc). A machine of type no_choc must switch on the warning (light_empty). In *OASIS* we have:

```
no_choc {num_chocs=0},
with_choc {num_chocs > 0}
 dynamic specialization of choc_vm;
class no_choc
triggers
 ::switch_on when {light_empty=false};
end class
```

We will also include a static partition of class choc_vm. Thus we make a distinction between bounded[11] (bounded_choc) and unbounded (unbounded_choc). The following *OASIS* specification gives the detailed information of this partition:

```
bounded_choc, unbounded_choc
 static specialization of choc_vm;

class unbounded
protocols
 GETCHOC:
  GETCHOC1=coin_in.GETCHOC2;
  GETCHOC2={credit=1}give.GETCHOC1 + {credit > 1}give.GETCHOC2
      + coin_in.GETCHOC2 +   {credit=1}::coin_out.GETCHOC1
      + {credit > 1}::coin_out.GETCHOC2;
end class

class bounded
protocols
 GETCHOC:
 GETCHOC1=coin_in.GETCHOC2;
 GETCHOC2=give.GETCHOC1+ coin_in.GETCHOC3 +::coin_out.GETCHOC1;
```

[10] It is mandatory in *OASIS* to repeat the full inherited condition and then to expend the formula in the subclass specialization.

[11] A bounded machine has a limited admission of credit.

```
 GETCHOC3=give.GETCHOC2+ coin_in.GETCHOC4 +::coin_out.GETCHOC2;
 GETCHOC4=give.GETCHOC3 + ::coin_out.GETCHOC3;
end class
```

Eventually, vm is dynamically partitioned in working and broken_down. After the creation action (set) the machine starts in the working state. A machine may be repaired (repair) to reach the working state. When the action break_down occurs, the machines moves to the state broken_down where the events cancel, coin_in and give are not habilitated. The specification in *OASIS* (see the Figure 5) is the following one:

```
broken_down, working
 dynamic specialization of vm
 migration relation is
        vm = set.working;
        working = break_down.broken_down;
        broken_down = repair.working;

class broken_down
events
 repair;
preconditions
 coin_in if {false};
 CANCEL if {false};
 give if {false};
end class

class working
events
 break_down;
end class
```

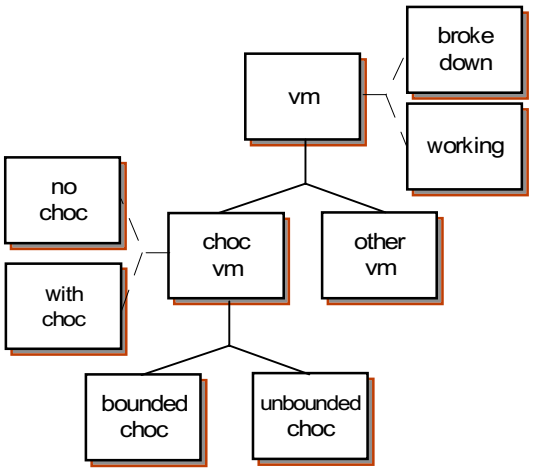


Figure 5: Specialization hierarchy.

A.1 *DL* for each *OASIS* class

We give here a *DL* formulae set for each class mentioned above:

class vm

```
credit=N →[coin_in] credit=N+1
credit=N →[::coin_out] credit=N-1
```

```
credit=N →[give] credit=N-1
[::switch_on] light_empty=true
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1 →[::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1 →[::coin_out] cancel=0
¬(credit > 0) →[give] false
¬(credit > 0) →[CANCEL] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit=1)
→[¬::coin_out] false
```

class choc_vm

```
num_chocs=N →[give] num_chocs=N-1
¬(num_chocs > 0 ∧ credit > 0) →[give] false
```

class bounded_choc

```
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 →[coin_in] getchoc=GETCHOC3
getchoc=GETCHOC3 →[coin_in] getchoc=GETCHOC4
getchoc=GETCHOC2 →[choc] getchoc=GETCHOC1
getchoc=GETCHOC3 →[choc] getchoc=GETCHOC2
getchoc=GETCHOC4 →[choc] getchoc=GETCHOC3
getchoc=GETCHOC2 →[::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC3 →[::coin_out] getchoc=GETCHOC2
getchoc=GETCHOC4 →[::coin_out] getchoc=GETCHOC3
¬((getchoc=GETCHOC1)∨ (getchoc=GETCHOC2)
∨ (getchoc=GETCHOC3))→[coin_in] false
¬((getchoc=GETCHOC2) ∨ (getchoc=GETCHOC3) ∨ (getchoc=GETCHOC4))→[choc]
false
¬((getchoc=GETCHOC2)∨ (getchoc=GETCHOC3)
∨ (getchoc=GETCHOC4))→[::coin_out] false
```

class unbounded_choc

```
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[give] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[give] getchoc=GETCHOC2[12]
getchoc=GETCHOC2 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[::coin_out] getchoc=GETCHOC2
¬((getchoc=GETCHOC1) ∨ (getchoc=GETCHOC2))→[coin_in] false
¬((getchoc=GETCHOC2) →[give] false
¬((getchoc=GETCHOC2) →[::coin_out] false
```

class no_choc

```
light_empty=false →[¬::switch_on] false
```

---

[12] This formula and next one may be deleted due to not modifying the variable value of the process.

class broken_down

```
¬(false) →[coin_in] false
¬(false) →[CANCEL] false
¬(false) →[give] false
```

A.2 Representation of dynamic partitions

As mentioned above, properties of dynamic partitions need to be included in the partitioned class. The properties of the class choc_vm after incorporating the properties of classes no_choc and with_choc are the following ones:

```
num_chocs=0 ∧ light_empty=false →[¬::switch_on] false[13]
num_chocs=N →[give] num_chocs=N-1
¬(num_chocs > 0 ∧ credit > 0) →[give] false
```

The inclusion of properties of the subclasses broke_down and working into the class vm plus the migratory process, gives the following set of *DL* formulae:

```
credit=N →[coin_in] credit=N+1
credit=N →[::coin_out] credit=N-1
credit=N →[give] credit=N-1
[::switch_on] light_empty=true
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1→[::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1→[::coin_out] cancel=0
¬(vm=working) →[give] false[14]
¬(vm=working) →[CANCEL] false
¬(vm=working) →[coin_in] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit < =1)
→[::coin_out] false
[set] vm=working[15]
vm=working ∧¬(credit > 0)→ [give] false
vm=working ∧ ¬(credit > 0)→ [CANCEL] false
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
```

A.3 Representation of static partitions

After dealing with dynamic partitions we need to derive the *DL* formulae for each path from each leaf class to the top class.

---

[13] This formula is moved here from the dynamic subclass.

[14] This permission and the next two ones replace the prohibitions of vm due to be more restrictive.

[15] This formula and the following one are related to the migratory process of the dynamic partition

class vm - choc_vm - bounded

num_chocs=N → [give] num_chocs=N-1
credit=N → [coin_in] credit=N+1
credit=N → [::coin_out] credit=N-1
credit=N → [give] credit=N-1
[::switch_on] light_empty=true
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 → [coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 → [coin_in] getchoc=GETCHOC3
getchoc=GETCHOC3 → [coin_in] getchoc=GETCHOC4
getchoc=GETCHOC2 → [choc] getchoc=GETCHOC1
getchoc=GETCHOC3 → [choc] getchoc=GETCHOC2
getchoc=GETCHOC4 → [choc] getchoc=GETCHOC3
getchoc=GETCHOC2 → [::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC3 → [::coin_out] getchoc=GETCHOC2
getchoc=GETCHOC4 → [::coin_out] getchoc=GETCHOC3
[set] vm=working
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1 → [::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1 → [::coin_out] cancel=0
¬(vm=working) → [give] false
¬(vm=working) →[CANCEL] false
¬(vm=working) → [coin_in] false
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
¬((getchoc=GETCHOC1)∨ (getchoc=GETCHOC2)
∨ (getchoc=GETCHOC3))→[coin_in] false
¬((getchoc=GETCHOC2)∨ (getchoc=GETCHOC3) ∨ (getchoc=GETCHOC4))→ [choc]
false
¬((getchoc=GETCHOC2)∨ (getchoc=GETCHOC3)
∨ (getchoc=GETCHOC4))→[::coin_out] false
¬(num_chocs > 0 ∧ credit > 0) → [give] false[16]
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit=1)
→ [::coin_out] false
num_chocs=0 ∧ light_empty=false → [¬::switch_on] false

class vm - choc_vm - unbounded

num_chocs=N → [give] num_chocs=N-1
credit=N → [coin_in] credit=N+1
credit=N → [::coin_out] credit=N-1
credit=N → [give] credit=N-1
[::switch_on] light_empty=true
[set] vm=working
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
[set] cancel=0
[CANCEL] cancel=CANCEL1

---

16 This prohibition formulae associated to get event what have just been mentioned just above can be mixed into one formula by using the logic connector '∨'.

```
cancel=CANCEL1 ∧ credit > 1→ [::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1→ [::coin_out] cancel=0
¬(vm=working) → [give] false
¬(vm=working) →[CANCEL] false
¬(vm=working) → [coin_in] false
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
¬(num_chocs > 0 ∧ credit > 0) → [give] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit=1)
→ [::coin_out] false
num_chocs=0 ∧ switch_on=false → [¬::light_empty] false
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[give] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[give] getchoc=GETCHOC2[17]
getchoc=GETCHOC2 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[::coin_out] getchoc=GETCHOC2
¬((getchoc=GETCHOC1) ∨ (getchoc=GETCHOC2))→[coin_in] false
¬(getchoc=GETCHOC2) →[give] false
¬(getchoc=GETCHOC2) →[::coin_out] false
¬(getchoc=GETCHOC1) ∨ (getchoc=GETCHOC2) →[coin_in] false
¬(getchoc=GETCHOC2) → [give] false
¬(getchoc=GETCHOC2) → [::coin_out] false
```

class vm - other_vm

```
credit=N →[coin_in] credit=N+1
credit=N →[::coin_out] credit=N-1
credit=N →[give] credit=N-1
[::switch_on] light_empty=true
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1→[::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1→[::coin_out] cancel=0
¬(vm=working) →[give] false
¬(vm=working) →[CANCEL] false
¬(vm=working) →[coin_in] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit < =1)
→[::coin_out] false
[set] vm=working
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
```

A.4 Properties of species

[17] This formula and the next one can be deleted although we prefer to keep them in order to make the translation process clearer.

As mentioned above, it is possible to deduce the properties (as *DL* formulae) for each species as the union of the respective properties. In the example we distinguish the following species involving only static partitions[18]:

```
other_vm
bounded_choc
unbounded_choc
```

Theses species correspond with each class mentioned above. In order to complete the calculus of properties we present here the set of identification mechanisms, attributes and events deduced from the specialization hierarchy of our example.

A.4.1 Identification mechanisms

The unique available identification mechanism (number) has been defined in the class vm. This mechanism is inherited by every species.

A.4.2 Attributes and events

We present the attribute and event sets for each species. We assume that the possible conflicts caused by coincidental names have been previously adjusted as a part of the consistency verification process of the specification.

Specie vm - choc_vm - bounded

Atr={number, credit, switch_on, num_chocs}
Ev={set, coin_in, coin_out, light_empty, give, CANCEL, repair, break_down}

Specie vm - choc_vm - unbounded

Atr={number, credit, switch_on, num_chocs}
Ev={set, coin_in, coin_out, light_empty, give, CANCEL, repair, break_down}

Especie vm - other_vm

Atr={number, credit, switch_on}
Ev={set, coin_in, coin_out, light_empty, give, CANCEL, repair, break_down}

[18] In this example, taking into account that there is only one hierarchy and that there is only one double static partition of the same class, species are formed by only one class. In a more general case it should be necessary to connect the properties of each class.

Un lenguaje formal para especificación y refinamiento de Objetos de Negocio

Raquel Anaya, Isidro Ramos

Universidad EAFIT. Medellín, Colombia
ranaya@sigma eafit edu.co

Universidad Politécnica de Valencia. Valencia. España
iramos@dsic upv es

Resumen. La descomposición, tanto horizontal como vertical, es una estrategia para enfrentar la complejidad del problema y construír componentes software reusables. El lenguaje extendido de Oasis permitirá formalizar el proceso de especificación y refinamiento de los componentes incorporando dos tipos de clases : Las clases tipo actividad que establecen la comunicación desde y hacia el entorno, implementan las reglas del negocio del comportamiento global y definen el mecanismo de sincronización de servicios. Las clases tipo recurso que describen la estructura y el compromiso (rol) de los objetos en un comportamiento. Según el nivel de abstracción, estos dos tipos representan conceptos abstractos en el dominio del problema o conceptos específicos en el dominio de la solución. El proceso de refinamiento representa un cambio de granularidad que transforma un requerimiento abstracto en un diseño concreto, proceso formalmente soportado por la técnica de reificación

Introducción

El reuso es una estrategia prometedora para enfrentar la demanda de sistemas de información cada vez más complejos que apoyen las operaciones del negocio en entornos organizacionales altamente competitivos y dinámicos [Pri93] [FF95]. La descomposición es uno de los principios básicos de la reutilización ; para que una solución software construida para satisfacer una necesidad actual, pueda ser reutilizada en un necesidad futura, se requiere desarrollar piezas de software flexibles y con el nivel de granularidad adecuado para representar una funcionalidad relevante en el dominio del problema. Dichas piezas ("*features*") deben permitir adecuadamente su integración posterior ("*feature integration*") en otro sistema, sin producir problemas ("*feature interaction problem*") [FIRE]. Se evoluciona entonces de una aproximación orientada al desarrollo de un sistema completo, a una aproximación orientada al desarrollo de componentes [Gog86] [NT95].

La descomposición del espacio del problema puede realizarse de manera horizontal y vertical. La perspectiva *horizontal* esta apoyada en el viejo principio de

Anaya, R., Ramos, I. Un lenguaje formal para especificación y refinamiento de Objetos de Negocio. Ingeniería del software y bases de datos: tendencias actuales. Universidad de Castilla-La Mancha, 2000.

modularización en el que la funcionalidad total del sistema se descompone en funcionalidades menores en un mismo nivel de abstracción La perspectiva *vertical* hace énfasis en el aspecto de refinamiento por pasos que involucra más de un nivel de abstracción [Kru92] [ED93][Eck98b]

Las metodologías tradicionales OO permiten la descomposición horizontal y vertical del problema [Rum91][SM91][Jac93][UML] La descomposición horizontal a través de técnicas como los Casos de Uso y los Diagramas de Colaboración, facilitan desde una perspectiva funcional, el modelado de los objetos del negocio de cara a una funcionalidad útil en el espacio del problema [Fow97]. La descomposición vertical se presenta de manera intuitiva mediante el enfoque iterativo del análisis OO que parte de unas clases abstractas y llega a clases de diseño e implementación Sin embargo, el tratamiento de estos dos tipos de descomposición se realiza de manera informal ; los entornos de desarrollo generalmente no proveen mecanismos para conservar versiones relevantes del proceso de refinamiento o de la descomposición por funciones, en los que se maneje de forma automática la trazabilidad en un mismo nivel de abstracción (integración de aspectos) o en diferentes niveles de abstracción (de los objetos de negocio a los objetos de diseño)

Por otra parte, existen tendencias importantes de acercamiento de los lenguajes de especificación formal a modelos gráficos convencionales [JWH+95] [MC94] [BW94] [PIP+97]. Esta aproximación logra entornos de desarrollo de software robustos con una semántica y una sintaxis bien definidas y a la vez facilitan un acercamiento amigable a la especificación del problema. OASIS y su extensión metodológica OO-Method, se encuentran en esta línea La plantilla de definición de clases en el lenguaje Oasis, se traduce a un conjunto de fórmulas en lógica dinámica, con una semántica declarativa dada por una estructura Kripke [LRS+98].

El objetivo de este trabajo es presentar una versión extendida de Oasis 3 0, para adecuarlo a la construcción de componentes reutilizables que representen diferentes niveles de abstracción y que describan el compromiso que los objetos del negocio adquieren para satisfacer una funcionalidad útil en el espacio del problema.

La segunda parte presenta el aspecto de descomposición horizontal a través de clases tipo actividad en la que se modelan objetos de negocio desde una perspectiva funcional, la tercera parte presenta el aspecto de refinamiento formalmente soportado por el principio de reificación En cada uno de estos tópicos se relacionan los principales trabajos existentes ; las extensiones propuestas al lenguaje se ilustran con el caso de estudio de alquiler de vehículos; en la presentación del lenguaje se omiten propiedades no relevantes para el propósito de este trabajo La cuarta parte, a manera de conclusión, sintetiza los principales aportes de este trabajo e introduce el formalismo gráfico que será utilizado para dirigir el proceso de recopilación de requisitos y refinamiento de los objetos de negocio.

Aproximación funcional para el modelado de objetos de negocio

Trabajos en el área

Son diversos los lenguajes formales que tratan el problema del comportamiento cooperativo de objetos desde una perspectiva funcional. Liu et al utiliza la lógica temporal de primer orden para definir la actividad como un constructor básico que expresa patrones de comportamiento interobjetuales y sus mecanismos de agregación y composición [LM96]. El trabajo de [MKB97] define primitivas para expresar colaboraciones entre objetos en un lenguaje de definición (DDL) por medio de clases de comportamiento (*bclass*). Los trabajos de [HHG90] y [Hol92] introducen el concepto de *contrato* como los constructores de alto nivel para especificar interacciones entre grupos de objetos. Los contratos definen unos participantes con sus obligaciones contractuales (métodos) e incluyen una declaración de los invariantes o reglas que condicionan su comportamiento.

Otros trabajos enfatizan el concepto de rol del objeto. Un rol provee una vista del comportamiento de un objeto en un contexto particular [BFP95]. Aproximaciones medodológicas como IOOM [DP95], propuestas formales como [AR92], entornos complejos de producción de software como [BGK+97] y proyectos grandes de reutilización como Ithaca [BFP95], introducen este concepto como estrategia de modelado para representar objetos con comportamiento complejo ; el rol provee un nivel de granularidad más fino que el objeto mismo, aumentando la potencialidad de reuso. Desde esta perspectiva, un objeto de negocio puede ser definido como una agregación compleja de estructuras y comportamientos no disjuntos construído de manera incremental y conservativa.

Aproximación funcional en Oasis

Se propone diferenciar entre dos tipos de clases : Clases que representan recursos y clases que representan actividades. Una clase tipo actividad sirve para declarar la colaboración que se establece entre objetos participantes; una clase tipo recurso es la plantilla que describe la estructura y comportamiento de un objeto participante. Estos dos tipos de clase podrían ser declarados con la semántica básica que maneja OASIS que a través del mecanismo de agregación puede dar cuenta de objetos que participan en un comportamiento complejo ; desde nuestro enfoque, más que una agregación se trata de una relación de inclusión. Las extensiones que se proponen, ya sea desde el punto de vista sintáctico o semántico, tienen por objetivo dar mayor riqueza expresiva al lenguaje desde la perspectiva de desarrollo de componentes reusables.

Clase tipo actividad

Un tipo actividad puede definirse como la composición concurrente de una sociedad de objetos para un comportamiento específico, en la cual la signatura de los objetos participantes es visible completamente. Esta clase cumple tres funciones básicas : Primera, establecer la comunicación desde y hacia el entorno de cara a este comportamiento; segunda, implementar las reglas del negocio que rigen el comportamiento global y tercero, servir como mecanismo coordinador de los servicios que requieren sincronización Los objetos tipo actividad son objetos dotados de la noción de estado que representan instancias de ejecución de una actividad global que para nuestro propósito se corresponde con un caso de uso[Ana99]

El cuadro no 1 presenta la especificación de la actividad AlquilarVehículo Se tiene un bloque para declaración de los objetos participantes. Cada participante es definido con un nombre de variable, nombre de la clase a la que corresponde y el nombre del rol. Los participantes de esta actividad son los objetos de negocio cliente y vehículo, con los roles de alquilador y objeto_alquiler, respectivamente. SolicitarVehiculo y RecibirVehiculo son estímulos recibidos del entorno que aparecen en la signatura de la clase, los cuales originan una solicitud de servicio a los objetos participantes (a) y (b).

Los atributos constantes de esta clase representan los parámetros generales asociados al comportamiento. En el ejemplo, el número de días límite que puede estar un vehículo alquilado (PlazoLimAlquiler) y el número máximo de vehículos que puede tener un cliente a cargo (NroVehCliente) son parámetros definidos para cada instancia de ejecución de la actividad en el momento de su creación Estos parámetros describen propiedades volátiles de la actividad que generalmente se asocian a las reglas de negocio que rigen el comportamiento global Las reglas del negocio pueden ser declaradas a través de precondiciones, restricciones de integridad, etc ; en este caso la precondición indica que se puede ejecutar el servicio SolicitarVehiculo si el cliente no ha llegado al límite de vehículos que puede tener a cargo (c).

```
Class AlquilarVehiculo
participants
        c : Cliente01 as alquilador;
        h : Vehiculo01 as objeto_alquiler;
constants attributes
        PlazoLimAlquiler : nat ;
        NroVehCliente : nat ;
variable attributes
events
        alta new
        SolicitarVehiculo calling with members
        (a)
                c.SolicitarVeh;

                h.EntregarVeh;
```

RecibirVehiculo calling with members

(b)

h.RecibirVeh;

c.DevolverVeh;

preconditions

SolicitarVehiculo if c.TotalContratos < NroVehCliente

(c)

End Class AlquilarVehiculo

Cuadro no. 1 declaración de la actividad abstracta *AlquilarVehículo*

Clases tipo recurso

Son las clases tradicionales que se define en una especificación OO y que normalmente se representan en esquemas de una base de datos relacional. Para aquellas clases que representen entidades del dominio del problema con comportamientos diversos y complejos en más de un contexto (producto, cliente, venta, etc) se introduce el concepto de rol o aspecto de la clase. Es decir, la definición de tipo declara las propiedades del objeto relevantes para el comportamiento y guarda silencio acerca de propiedades no relevantes [AR92]

El cuadro no. 2 presentan la especificación para los objetos de negocio que participan en el comportamiento *AlquilarVehículo*. La declaración de rol se hace explícita en el encabezado de la clase agregando la palabra clave played by, seguido por el nombre de la clase base de la cual forma parte y el nombre del rol que desempeña en este comportamiento. Por ejemplo *Vehiculo01*, está asociado a la clase base Vehículo y describe la estructura y comportamiento relevante para el alquiler en el que desempeña el rol de *objeto_alquiler*.

Class Vehiculo01 **played by** Vehiculo (objeto_alquiler)

variable attributes

tarifa : nat;

estado : string;

condiciones : string

events

EntregarVeh ;

RecibirVeh ;

valuations

[EntregarVeh] estado = 'alquilado';

[RecibirVeh] estado = 'devuelto'

preconditions

EntregarVeh if {estado='disponible' };

end Class VehiculoRol01

```
Class Cliente01 played by Cliente (alquilador)
....
variable attributes
        totalcontratos : nat(0);
events
        SolicitarVeh;
        DevolverVeh;
valuations
        [SolicitarVeh] totalcontratos = totalcontratos + 1;
        [DevolverVeh] totalcontratos = totalcontratos – 1;
end Class Cliente01
```

Cuadro no 2 Declaración abstracta de *Cliente* y *Vehículo*

Modelado del comportamiento interobjetual

El tipo actividad representa una sociedad o sistema de objetos que existen concurrentemente Cada objeto participante posee un ciclo de vida local que para propósitos de este comportamiento debe sincronizarse en diferentes puntos del tiempo para formar un ciclo de vida global distribuído [DE95a]

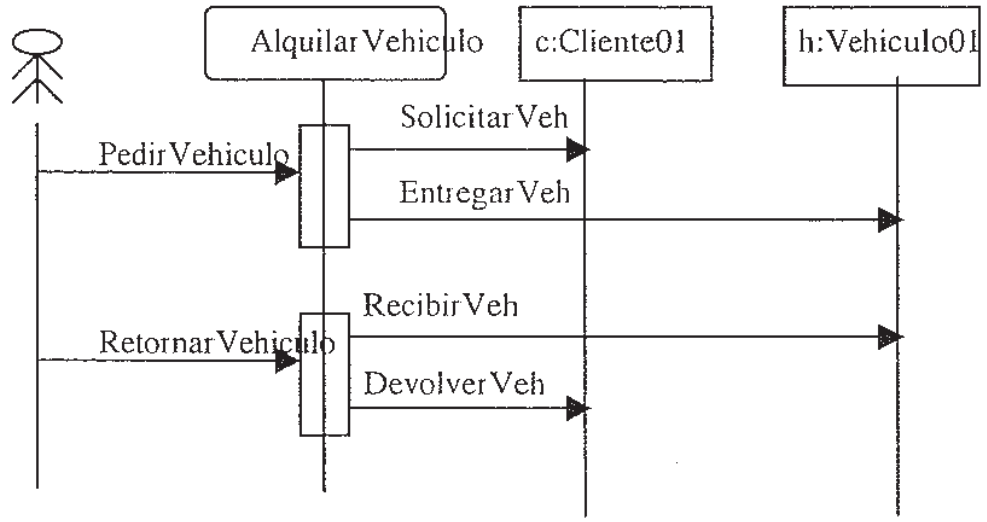


Figura no 1 Sincronización abstracta para *AlquilarVehiculo*

Los diagramas de secuencia, sirven como formalismo gráfico para capturar los puntos en el tiempo donde los ciclos de vida de la sociedad de objetos deben sincronizarse, como resultado de los requisitos del domino del problema y de la semántica asociada a los respectivos servicios. La figura no 1 presenta parte del Diagrama de Secuencia para la actividad *AlquilarVehiculo* : cargar un vehículo a un cliente (c *SolicitarVeh*) debe sincronizarse con modificar el estado del vehículo (h *EntregarVeh*). En este caso la comunicación es sincrónica de ocurrencia forzada (*event calling*), puesto que los servicios deben ocurrir en los dos objetos o en ninguno de ellos La coordinación de la comunicación sincrónica es responsabilidad de la clase actividad : los

participantes de la actividad pueden a la vez establecer entre ellos comunicaciones de tipo asincrónico (*action waiting, action sending*) [LRS+98].

Hasta este momento, se tiene una especificación, correcta y completa, de la especificación en un primer nivel de abstracción en el que se describe el papel que los objetos del negocio *cliente* y *vehículo* desempeñan en la funcionalidad *AlquilarVehículo* Se trata ahora de describir el proceso de refinamiento de la especificación y los principios en los cuales se fundamenta

Refinamiento de la especificación

Trabajos en el área

La reificación es una técnica formal que estudia la reducción de la complejidad de software y que está siendo investigada bajo la aproximación de orientación por objetos La propuesta de refinamiento presentada en este trabajo está apoyada básicamente en la propuesta de Denker et al [ED93], [Den95], [Den95a], [Den96], [HWD96]. El uso del término **reificación** más que refinamiento o implementación hace énfasis en el *cambio de granularidad* de una especificacion Significa partir de una especificación inicial para construír una secuencia de especificaciones que van reemplazando conceptos abstractos de la especificación por conceptos más cercanos a un programa concreto

El proceso de reificación puede verse como una aproximación de tres niveles en la que se tienen tres tipos de objetos : Objetos abstractos (*abstract object*), que para nuestro caso se corresponde con los objetos del negocio, que representan comportamientos no deterministicos del dominio del problema, objetos base (*building block*) que describen los comportamientos básicos del problema y los objetos refinados (*middle object*) que son una composición de los objetos abstractos y objetos base, enriquecidos con una especificación de cómo los atributos abstractos son refinados a estructuras de base de datos (reificación de datos) y cómo las acciones abstractas son refinadas a transacciones base (reificación de procesos) [ED93] La función de reificación establece una correspondencia de los eventos abstractos a procesos de secuencialidad u opcionalidad sobre eventos concretos (base). Un evento que era atómico desde el punto de vista abstracto, viene a ser compuesto desde el punto de vista reificado. Asociado al efecto de cambio de granularidad, surge entonces el concepto de **transacción**, como una unidad lógica de un nivel de abstracción específico, compuesto de varios eventos simples a un nivel más concreto [Den96]

Los trabajos de Denker et al desarrollan formalmente esta propuesta de reificación en el contexto de la lógica temporal de eventos (DTL) teniendo como marco de interpretación la estructura de eventos y utilizando Troll como lenguaje formal de especificación [Den96]. En [Den95] se establece el criterio de correctitud de la reificación, desde el punto de vista semántico : un objeto refinado es una reificación

correcta de un objeto abstracto sobre un conjunto de objetos base cuando se cumplen dos condiciones : Primera, el objeto reificado sólo puede adicionar restricciones al comportamiento del objeto base, pero no puede modificar su comportamiento y segunda, los objetos reificados pueden ser utilizados sin pérdida de información en lugar de los objetos base Estas dos condiciones se expresan con la semántica de la estructura de eventos para significar que el conjunto de ciclos de vida que representan el comportamiento del objeto reificado, debe proyectarse tanto en la estructura abstracta como en las estructuras base. Desde el punto de vista sintáctico [DE95a] la prueba de reificación se establece cuando las fórmulas que se derivan de la reificación deben ser satisfechas tanto para los objetos abstractos como para los objetos concretos a través de una sustitución sintáctica.

Refinamiento de la clase recurso en Oasis

El refinamiento se describe por la palabra **reified** y representa el mecanismo de transformación de objetos abstractos en objetos concretos, tanto en su estructura como en su comportamiento El cuadro no. 3 presenta una vista simplificada de la reificación de Vehiculo01 El tratamiento formal y exhaustivo del refinamiento en el marco de la Lógica Dinámica y con la semántica de intrepretación de la estructura Kripke, se encuentra fuera del objetivo de este trabajo.

```
Class Vehiculo01R reified Vehiculo01 aggregation of
        static inclusive Contrato toward(0 1) from( 1,1) ;
        static relational Tarifa toward(posee : 1,1) from(aplicada_a : 0,*) ;
....
 variable attributes
        tarifa : nat derived;                          /* tarifa pasa a ser un atributo
derivado
  estado : string;
  combustible : nat;
  kilometros : nat;
  condiciones = {combustible : nat , kilometros : nat}  /*equivalencia del atributo abstracto
   events                                                                    /* en
atributos concretos
  CrearContrato calling to members              /* el objeto se encarga de sincronizar los
  servicios
                  Contrato Crear;                        /* de sus componentes
  PagarFianza calling to members
        Contrato.Pagar;
  DarEstadoAlq;
  PagarSaldo calling to members
        Contrato Pagar
  DarEstadoDev;
  EntregarVeh process ENTREGARVEH;      /* ecuaciones de equivalencia de eventos
  abstractos
  RecibirVeh   process RECIBIRVEH;      /* en procesos concretos
```

```
   valuations
        [DarEstadoAlq] estado = 'alquilado';
        [RecibirVeh] estado = 'devuelto';
   derivations
        tarifa = posee.precio;                    /* fórmula de derivación del
atributo refinado
   preconditions
        DarEstadoAlq if {estado='disponible'};
   operations
   ENTREGARVEH :                                  /* descripción de procesos que
han sido refinados
   EntregarVeh = CrearContrato.VEH0;
VEH0        = PagarFianza.VEH1;
   VEH1              = ::DarEstAlq;

   RECIBIRVEH transaction:

RecibirVeh = PagarSaldo.REC0;
   REC0 = ::DarEstadoDev;
   end Class Vehiculo01R
```

Cuadro no. 3 Declaración refinada de *Vehiculo01*

```
   Class Contrato
   .....
   constant attributes
        numero : string ;
        fecha_desde : date;
        nro_dias : nat
   variable attributes
        totalpagado : nat(0);
        valorcontrato : nat;
        fecha_hasta : date;
   events
        Crear  new;
   Pagar ;
        Cerrar  destroy;
   valuations
        [Pagar(cant)] total_pagado = total_pagado + cant;
   end class Contrato

   Class Tarifa aggregation of
        static relational  Vehiculo01R toward(aplicada_a : 0,*) from(posee : 1,1) ;
   identification
        codigo : (codigo)
   constant attributes
        codigo : nat;
        descripción : nat
```

```
variable attributes
        precio ;
events
        crear_tarifa new ;
        cambiar_precio (cant nat) ;
        borrar_tarifa destroy;
"""
valuations
        [cambiar_precio(cant)] precio_alq = cant;
protocols
TARIFA :
        TARIFA0 =
                crear_tarifa TARIFA1
        TARIFA1 = cambiar_precio TARIFA1 +
                borrar_tarifa;
end Class Tarifa
```

Cuadro no 4 Especificación de objetos base para *Vehiculo01R*

En la propuesta de Denker et al la transacción es el concepto que representa un cambio de granularidad ; una transacción comprende un conjunto de acciones ejecutadas consecutivamente sin observabilidad intermedia [Den96] Es importante resaltar la riqueza semántica y expresiva que posee Oasis para describir el objeto como un proceso observable [RPC+93] [LRS+98]. La versión 3 establece diferencia entre procesos que *pueden* ocurrir en la traza del objeto (protocolos), procesos que *deben* ocurrir (operaciones) y procesos que tienen una ocurrencia atómica sin observabilidad intermedia (transacción) ; utiliza un subconjunto de CCS como lenguaje para especificar la diversidad de procesos De esta manera se pueden establecer funciones de reificación más ricas que en la propuesta original, facilitando el modelado de problemas reales de gestión de objetos de negocio.

Las clases *Contrato* y *Tarifa* surgen en el proceso de refinamiento de *Vehiculo01* (cuadro no.4). *Contrato* es un objeto agregado de *Vehiculo01*, lo cual indica que todos sus servicios serán coordinados a través de Vehículo01. *Tarifa* es un objeto secundario asociado con *Vehiculo* que se maneja en Oasis como una agregación de tipo relacional ; este objeto puede suscribir en su signatura local, eventos con el entorno expresados en su sección de protocolo.

3.3 Refinamiento de la clase actividad en Oasis

Vale la pena considerar ahora. el efecto que el proceso de reificación tiene en la sincronización que se declaró a nivel abstracto Como ya fue mencionado, la clase actividad será la encargada de establecer la coordinación de eventos entre el objeto complejo (actividad) y los objetos agregados (participantes) a través de una

comunicación sincrónica y de ocurrencia forzada Teniendo en cuenta que los eventos a nivel abstracto se reifican como procesos (operaciones, protocolos, transacciones) en el nivel básico, se hace necesario analizar el efecto que el refinamiento tiene sobre esta sincronización inicial

En la propuesta original de reificación, Denker et al establece una regla que declara que la especificación de una interacción abstracta significa que hay un 'punto de comunicación' durante la ejecución de las acciones [Den96]. Los planteamientos dados por Denker et.al, están orientados a establecer modelos de sincronización en los que se optimiza la concurrencia de acciones involucradas en una transacción considerando aspectos de control de concurrencia y paralelismo de transacciones Para el propósito práctico que persigue este trabajo se hace necesario establecer los principios que determinen en qué momento es relevante redefinir la sincronización sobre los objetos reificados

En situaciones donde las acciones abstractas se refinan en procesos complejos y específicamente cuando los pasos del proceso contemplen estímulos del entorno (existe observabilidad intermedia) , es deseable que el diseñador pueda establecer los puntos de sincronización adecuados ; esperar que éstos se sincronicen en su punto de finalización podría ser una restricción fuerte para el modelo de comunicación. A continuación se intenta explicar mediante el ejemplo en estudio.

La clase *AlquilerVehículo* declara dos casos de sincronización :

(1) happens c SolicitarVeh $\Rightarrow$ happens h.EntregarVeh

(2) happens h RecibirVeh $\Rightarrow$ happens c.DevolverVeh

y dos funciones de refinamiento :

ρ (EntregarVeh) = Operation{CrearContrato, PagarFianza, DarEstAlq}

ρ (RecibirVeh) = Transaction{PagarSaldo, DarEstadoDev}

Para el caso (2) donde *RecibirVeh* se convierte en transacción y *DevolverVeh* no se descompone, la sincronización se establece en el punto de *commit* de RecibirVeh con el evento DevolverVeh, es decir, la declaración de sincronización no se modifica Para el caso (1) donde *EntregarVeh* se convierte en una operación podrían presentarse alternativas de sincronización válidas y significativas desde el dominio de la aplicación : En qué momento se hace efectivo el aumento de vehículos a cargo del cliente (SolicitarVeh) ? Al crear el contrato o cuando el cliente haya pagado la fianza ? En este caso se seleccionó la primera alternativa.

El cuadro no. 5 presenta la especificación refinada de *AlquilarVehiculo* En esta especificación ya refinada se puede realizar, una declaración precisa de los axiomas que rigen las reglas del negocio Es decir, la especificación refinada de un tipo actividad puede establecer limitaciones de comportamiento sobre los objetos base. La especificación refinada de la actividad se realiza de manera asistida a través de un Diagrama de Secuencia similar al mostrado en el figura no. 1.

```
Class AlquilarVehiculoR reified AlquilarVehiculo
participants
        c : Cliente01 as alquilador;
        h : Vehiculo01R as objeto_alquiler;        /* referencia el tipo refinado de
ehiculo
        t : Tarifa;                                /* Tarifa es un agente secundario
ıe surge al
constants attributes                               /* reficar Vehiculo01
plazo_lim_alquiler : nat ;
nro_veh_cliente : nat ;
variable attributes
events
        alta new
        SolicitarVehiculo calling with members     /* nueva declaración de
ncronización sobre
                c.SolicitarVeh;                    /* los eventos concretos

                h.CrearContrato;

        RecibirVehiculo calling with members

                h RecibirVeh;

                c.DevolverVeh;
        CrearContrato calling with members         /* nuevos servicios que pueden
:r activados
                h.CrearContrato;                   /* por el entorno
        PagarFianza calling with members
                h.PagarSaldo;
        PagarSaldo calling with members
                h.PagarSaldo ;
        CrearTarifa calling with members
                t crear_tarifa ;

valuations
preconditions
        SolicitarVehiculo if c.totalcontratos < nro_veh_cliente
        (c)
End Class AlquilarVehiculoR
```

Cuadro no. 5 Especificación refinada de *AlquilarVehiculo*

Conclusiones

Se ha presentado una extensión al lenguaje formal Oasis para especificación de componentes abstractos que representen una funcionalidad relevante en el espacio del problema (partición horizontal) y que a la vez permita el refinamiento de los objetos de negocio en objetos de diseño (partición vertical)

Este trabajo aumenta la expresividad de un lenguaje formal de especificación para tratar de manera homogénea aspectos que son considerados por los trabajos relacionados de manera independiente. Las extensiones propuestas habilitan el lenguaje como soporte formal para el proceso de desarrollo de componentes en modelos del dominio construidos pensando en su reutilización posterior.

La formalización permite determinar los principios y heurísticos básicos que serán utilizadas en el entorno gráfico, utilizando la notación UML, para capturar los requisitos y refinamiento de los componentes Una actividad se corresponde con el concepto de Caso de Uso, el vocabulario que permite describir la funcionalidad se define a través del Diagramas de Clase particulares a cada comportamiento (modelo de roles) y la sincronización de acciones y responsabilidades de los objetos participantes se describe por medio de Diagramas de Secuencia [Ana99].

Aunque Denker et al propone un marco de formalización del refinamiento uniforme desde el punto de vista estructural y de comportamiento, para efectos prácticos este trabajo retoma el aspecto de refinamiento del comportamiento y utiliza principios del área de modelización semántica para el refinamiento estructural [DP95].

Bibliografía.

[Ana99] Anaya, R *Un Modelo Para el Desarrollo de Componentes Reusables*. Memorias 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software. San José de Costa Rica Marzo de 1 999 Por aparecer

[AR92] Andersen, E ; Reenskaug, T *System Design by Composing Structures of Interacting Objects* ECOOP 92 European Conference on Object Oriented Programming Ed. Springer-Verlag, 1 992

[BGK+97] Baumer, D ; Gryczan G ; Knoll, R ; Lilienthal, C ; Riehle, D ; Zullighoven, H *Framework Development for Large Systems* Comunications of the ACM. Volumen 40 no 10, Octubre de 1 997.

[BFP95] Bellinzona, R ; Fugini, M ; Pernici B *Reusing Specifications in OO Applications* IEEE Software, Marzo de 1 995

[BW94] Briggs, T ; Werth J *A Specification Languaje for Object Oriented Analysis and Design* Lectures Notes in Computer Science ECOOP94

[Den95] Denker G *Transactions in Object-Oriented Specification*. http://www cs tu-bs dc/idb/publications/

[DE95a] Denker G ; Ehrich, H *Action Reification in Object-Oriented Specification* http://www.cs tu-bs de/idb/publications/

[Den96] Denker, G. *Reification - Changing Viewpoint but Preserving Truth* http://www cs tu-bs de/idb/publications/

[DP95] De Antonellis V ; Pernici B *Reusing Specifications Through Refinements Levels*. Data and Konwledge Engineering Volumen 15. 1.995.

[Eck98b] Eckstein, S Modules for Object Oriented Specification Languages: A Bipartite Approach http://www cs tu-bs.de/idb/publications/

[ED93] Ehrich H.; Denker G *Constructing Systems as Object Communities*. http://www cs tu-bs de/idb/publications/

[FF95] Frakes, W. ; Fox. C *Sixteen Questions About Software Reuse*, ACM Comunications. Volumen 38 no. 6. Junio de 1 995

[FIRE] Feature Integration in Requirements Engineering http://www cs bham ac uk/~mcp/fireworks/

[Fow97] Fowler. M *UML Distilled Applying the Standars Object Modeling* Languaje Addison Wesley. 1.997

[Gog86] Goguen. Joseph A *Reusing and Interconnecting Software Components* En Tutorial Software Reusability The Computer Society of the IEEE. 1 986

[HHG90] Helm, R.; Holland. R ; Gangopadhyyay. D *Contract - Specifying Behavioral Composition in Object-Oriented Systems*. Memorias ECOOP'90 ACM Press

[Hol92] Holland I *Specifying reusable components using Contracs* Memorias ECOOP'92 Springer-Verlag

[HWD96] Huhn, M ; Wehrheim. H ; Denker, G *Action Refinement in System Specification - Comparing a Process Algebraic and an Object-Oriented Aproach* http://www cs tu-bs.de/idb/publications/.

[Jac93] Jacobson, Ivar, et.al Object Oriented Software Engineering : A Use Case Driven Aproach. . Wesley Publishing, 1993

[JWH+94] Jungclaus. R ; Wieringa, R ; Hartel. P ; Saake, G ; Hartmann, T. *Combining TROLL with the Object Modeling Technique* http://www cs tu-bs de/idb/publications/

[Kru92] Krueger, C. W : *Software Reuse*, ACM Computing Surveys Volumen 24 no 2 Junio de 1.992

[LM96] Liu. L.; Meersman. R. *The Building Blocks for Specifying Communication Behavior of Complex Objects : An Activity-Driven Approach* ACM Transaction on Dtabase Systems. Vol 21. No. 2. 1 996

[LRS+98] Letelier P ; Ramos. I ; Sánchez, P.; Pastor O *OASIS Versión 3 0 Un Enfoque formal para el modelado conceptual Orientado a Objetos* Departamento de Sistemas Informáticos y Computación Facultad de Informática. Universidad Politécnica de Valencia, 1.998

[MC94] Moreira. A ; Clark. R. *Combining Object-Oriented Analysis and Formal Description Techniques*. Lectures Notes in Computer Science ECOOP94

[MKB97] Marshall, B ; Kennedy, J ; Barclay. P. *Bclass A construct and method for modelling co-operative object behavior* Information and Software Technology. Vol 39 1.997

[NT95] Nierstrasz, O ; Tsichritzis D. *Object Oriented Software Composition*. Prentice Hall. 1995

[PIP+97] Pastor. O ; Insfrán, E ; Pelechano, V ; Romero, J ; Merseguer J *OO-Method. An OO Software Production Environment Combining Conventional and Formal Method* Advanced Information System Engineering CAISE 97 Eds. Antoni Olivé, Joan Antoni Pastor Barcelona España, 1 997

[Pri93] Prieto-Diaz. Ruben *Status Report - Software Reusability* IEEE Software Mayo 1 993

[RPC+93] Ramos, I ; Pastor O ; Cuevas J ; Devesa J *Objects as Observable Processes*. Actas del 3rd Workshop on the Deductive Approach to Information System Design Cataluña 1.993

[Rum91] Rumbaugh James et al *Object-Oriented Modeling and Design* Prentice Hall. 1991

[SM91] Shaler, S ; Mellor S.J *Object-Oriented Systems Analysis · Modeling the worl in Data* New Jersey Prentice Hall. 1 991

[UML] Unified Unified Modeling Language http://www.rational.com

AR_2CA: Una Herramienta para la Construcción de Componentes Reutilizables a través de Niveles de Refinamiento

Raquel Anaya
Universidad EAFIT. Medellín, Colombia
ranaya@eafit.edu.co

Isidro Ramos
Universidad Politécnica de Valencia. Valencia, España
iramos@dsic.upv.es

Resumen

AR_2CA es una herramienta que permite la construcción, clasificación y recuperación de componentes ***para*** *reutilizar. La aproximación metodológica que subyace en AR_2CA define un proceso de desarrollo descendente, en el que claramente se distinguen los niveles de abstracción por los que evoluciona el componente en su construcción: requisitos, análisis, diseño y ejecución. El proceso de derivación de un nivel de abstracción a otro se apoya en primitivas de refinamiento estructural derivadas del área de modelización semántica y en primitivas de refinamiento funcional apoyadas en el principio de* ***reificación****. En el nivel de ejecución el componente presenta una interfaz homogénea como componente visible en el entorno de JavaBeans que facilita la validación de la especificación y el entendimiento del componente con el propósito de reutilizarlo.*

Palabras claves: Refinamiento de Especificaciones Desarrollo Basado en Componentes, Reutilización

Introducción

El desarrollo basado en componentes (CBD) es una de las áreas que ha recibido considerable atención como una de las aproximaciones que favorece el desarrollo de aplicaciones abiertas y distribuidas con el objetivo de soportar la gestión de las operaciones globales y dinámicas de la organización actual [NT95]. Una aproximación CBD percibe el desarrollo de aplicaciones como un proceso de selección, adaptación y composición a partir de componentes que han sido previamente construidos, clasificados y almacenados en una biblioteca con el propósito de ser reutilizados.

Para adoptar una aproximación CBD como estrategia de desarrollo, se requiere la consideración de aspectos tanto tecnológicos como metodológicos. La tecnología de componentes ha impulsado considerablemente este enfoque definiendo modelos computacionales que integran en una arquitectura software una diversidad de servicios que garantizan componentes *run-time* portables, distribuidos y heterogéneos [OH98]. El aspecto metodológico hace énfasis en la capacidad de composición del componente; en esta línea se estudian modelos y métodos [DP95][DC98], lenguajes de especificación de arquitecturas

Anaya, R., Ramos, I. AR2CA: Una Herramienta para la Construcción de componentes reutilizables a través de niveles de refinamiento. Actas de 3as Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software (IDEAS2000). 2000.

[MT97][CPT99] y lenguajes visuales [FNP92] que facilitan la representación, construcción, especificación e integración del componente.

Este trabajo presenta una primera versión de AR_2CA (Arquitectura para Refinamiento y Reutilización de Componentes de Análisis), una herramienta que apoya el proceso de construcción, clasificación y recuperación de componentes que representan una funcionalidad relevante en el espacio del problema [Ana99]. La aproximación metodológica que subyace en AR_2CA sigue un enfoque descendente en la construcción de componentes *para* reutilizar; se utilizan algunos de los formalismos gráficos propuestos por UML y se tiene como soporte formal Oasis [AR99], un lenguaje de especificación orientado a objetos y fundamentado en una variante de la lógica dinámica [LRS+98]. Las características de AR_2CA como herramienta de clasificación y recuperación de componentes, se encuentran fuera del objetivo del presente trabajo.

De la sección 1 a la sección 3 se describen los niveles de requisitos, análisis y diseño; la sección 4 describe el nivel de ejecución del componente en el entorno de *JavaBeans*; finalmente, se presentan las conclusiones y trabajos futuros.

1. El Nivel de Requisitos

El modelo que permite describir un componente al nivel más alto de abstracción, recibe el nombre de *modelo ontológico*. Su nombre se deriva del término *ontología* que es definido en el área de intercambio de conocimiento (*knowledge sharing*), como la "*especificación de una conceptualización*" [Gru]. Una ontología es el conjunto de términos de interés en un dominio de información particular y las relaciones entre ellos. La ontología establece un acuerdo en el uso de un vocabulario de manera consistente (aunque no completa) para una comunidad de agentes, que recibe el nombre de compromiso ontológico (*ontological commitmens*).

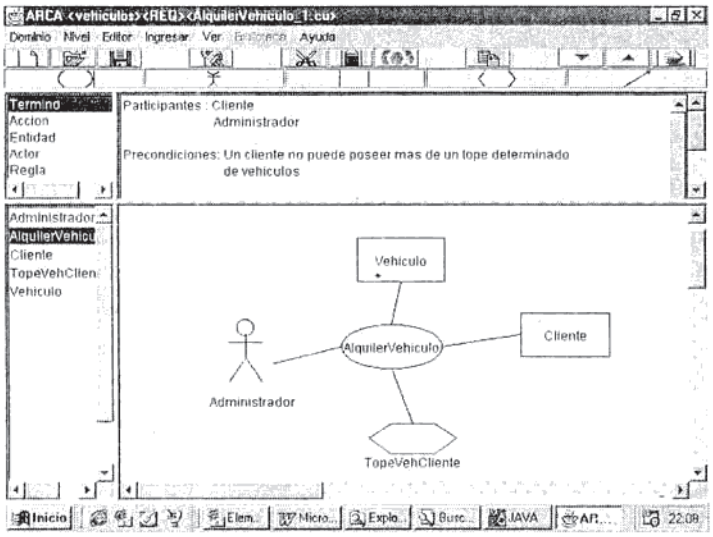


Figura no.1 Nivel de requisitos del componente

Se ha utilizado una notación extendida de los casos de uso [Jac93] para capturar el compromiso ontológico que adquiere el componente dentro de un dominio del problema. El diagrama de caso de uso cumple dos finalidades básicas: en primer lugar, sirve como vehículo

amigable para capturar progresivamente el vocabulario asociado a un dominio específico; en segundo lugar, sirve como estrategia para particionar el espacio del problema en unidades funcionales significativas para el usuario, las cuales definen el nivel de granularidad del componente; el caso de uso representa una acción abstracta compartida a partir de la cual se inicia el proceso de refinamiento del componente [DC98].

La figura no. 1 presenta la vista del caso de uso para la funcionalidad llamada *AlquilerVehiculo*. El modelo ontológico está representado por una *acción*, denominada *acción base*, ejecutada por un/unos *actor/ actores*, con el concurso de unas *entidades* y reglamentada por unas *reglas*. La acción representa un proceso elemental del negocio con un sentido relevante al usuario; los actores son los agente de la acción; las entidades son objetos claramente definidos en el espacio del problema que desempeñan un papel activo dentro de la acción, denominados *actuandos* [FHL+96] y las reglas son las políticas del negocio que restringen, condicionan o permiten la ejecución de la acción y representan las propiedades volátiles que el componente debe implementar.

2. El Nivel de Análisis

En este nivel se realiza una transición de los términos del modelo ontológico a estructuras de objetos y colaboraciones entre éstos. Este nivel está representado por dos modelos: el *modelo de colaboraciones* que describe la aspecto de comportamiento del componente y el *modelo de roles* que describe su aspecto estructural.

2.1 El modelo de colaboraciones

Se utiliza el diagrama de secuencia que cumple dos finalidades básicas: en un primer acercamiento, especifica el protocolo de interacción entre el componente y el entorno para posteriormente definir las responsabilidades de los objetos participantes en la ejecución de esta acción. Un modelo ontológico, definido en el nivel anterior, puede ser refinado en diferentes modelos de colaboración dando lugar a diferentes versiones de análisis para un mismo modelo de requisitos.

La figura no. 2 representa el escenario de *AlquilerVehículo* para el caso de uso definido anteriormente. Nótese que el objeto *unAlquiler* representa una instancia del objeto *AlquilerVehiculo* y sirve como objeto coordinador que establece la comunicación entre el usuario y el componente y encapsula su estructura y comportamiento interno. La entrada por parte de un actor recibe el nombre de *estímulo* [DIP96]; estos estímulos invocan los servicios con el mismo nombre que presta el objeto instancia de la clase coordinadora.

Una vez definidos los estímulos que implementará el componente, se identifican los objetos del negocio participantes y se establecen sus responsabilidades de cara a esta colaboración. La asignación de una responsabilidad en el objeto participante puede significar la creación de un nuevo método o 'puerto de entrada' en la plantilla de la clase de la cual el objeto es instancia o la reutilización de un 'puerto de entrada' que ya había sido definido sobre esta clase. En este ejemplo, para satisfacer el estímulo *SolicitarVehiculo*, se invoca sobre la instancia *unCliente* el mensaje *cargarVehiculo* y sobre la instancia *unVehículo* el mensaje *entregarVehiculo* que, en este caso, crea servicios en las clases respectivas con el mismo nombre del mensaje.

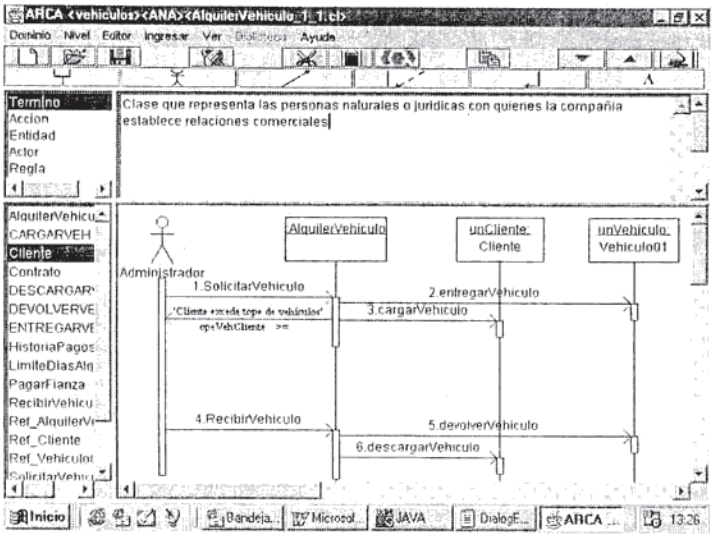


Figura no.2 Modelo abstracto de colaboraciones

2.2 El modelo de roles

El modelo de roles (figura no.3) está representado por un diagrama de clases que sigue la notación UML y que permite definir los atributos y métodos de los componentes que participan en el comportamiento y sus correspondientes relaciones de especialización, agregación y asociación. Se le denomina *modelo de roles* [AR92], puesto que el diagrama representa especificaciones parciales de los objetos del negocio; estos objetos son estructuras complejas que describen comportamientos diversos dentro de un dominio, siguiendo el patrón estructural propuesto en [BGK+97].

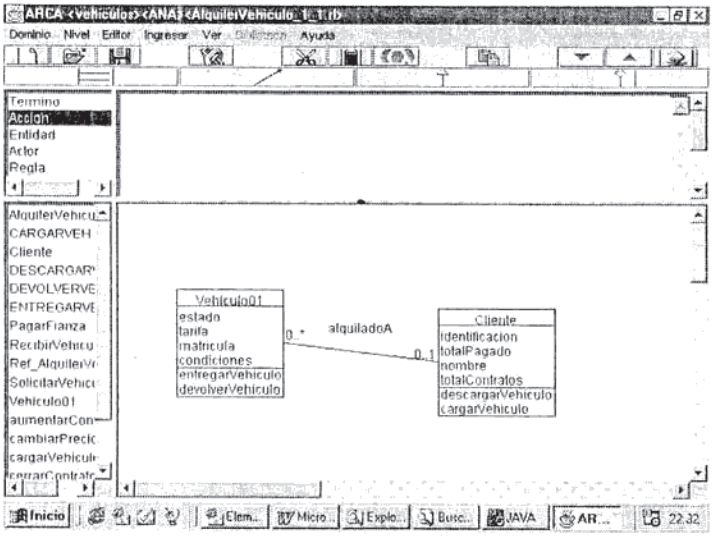


Figura no.3 Modelo abstracto de roles

El modelo de colaboraciones y el modelo de roles son complementarios y consistentes en todo momento. Por ejemplo, un mensaje asociado a la instancia de un objeto, en el diagrama de secuencia representa un nuevo método en el diagrama de clases o un método definido en el diagrama de clases se convierte en un puerto ya existente para definir la responsabilidad del objeto en el diagrama de secuencia.

2.3 Taxonomía de componentes en AR_2CA

Los componentes en AR_2CA son representados por medio de clases en donde se establece una clasificación de acuerdo a tres criterios (figura 4):

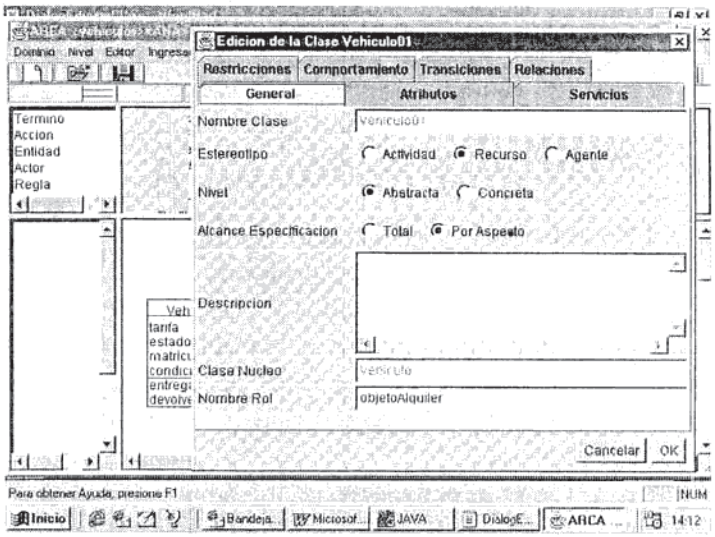


Figura 4. Vista general de edición de una clase

- La *semántica en el dominio* describe el papel que la clase desempeña según la semántica que el término de donde se deriva representa en el dominio; surgen tres tipos de clases: clase actividad, clase recurso y clase agente; este criterio de clasificación se define como el *estereotipo* de la clase.

- El *nivel de abstracción* determina si se trata de una *clase abstracta*, a la cual se le puede aplicar primitivas de refinamiento o se trata de una *clase concreta* que ha sido derivada de una clase abstracta o es una clase elemental a la cual no se le aplica refinamiento.

- El *alcance de la especificación*, válido para clases tipo recurso y determina si se desea introducir el criterio de aspecto de la clase para soportar especificaciones parciales de la clase. Como puede observarse en esta figura, la clase *Vehículo01* es una especificación parcial definida sobre una clase base *Vehículo* que desempeña el papel de *objetoAlquiler*.

Los diálogos asociados a una clase permiten capturar su especificación con toda la riqueza semántica soportada por el lenguaje Oasis. El diálogo *General* permite consultar o modificar las propiedades básicas de la clase; el diálogo *Atributos* permite definir nuevas propiedades de la clase o modificar las características de los atributos que ya existen; el diálogo *Servicios* crea o edita los servicios de la clase, definiendo el tipo de servicio, de acuerdo a la clasificación definida en el lenguaje Oasis; el diálogo *Restricciones*, permite definir los reglas

de integridad asociadas a los atributos, la conformación de la identificación externa de la clase y las fórmulas de derivación de los atributos declarados como derivados; el diálogo *Comportamiento* describe las precondiciones, valuaciones o disparos asociados a los servicios de la clase; el diálogo *Transiciones* crea los estados relevantes del objeto o del proceso complejo (protocolo/operación) y define los pasos válidos de transición de un estado a otro.

2.4 Tratamiento de las Reglas de Negocio

Es muy importante que el componente que esta siendo construido no sólo pueda satisfacer una necesidad actual sino que además, facilite su evolución a requerimientos futuros. Esto significa que el analista del dominio debe decidir cuáles de las propiedades del componente son estables y cuáles de estas propiedades pueden ser variables [DIP96]. En *AR$_2$CA* se definen mecanismos que permiten diferenciar las propiedades estables y variables del componente: la declaración de un término tipo regla en el modelo ontológico, corresponde a una política del negocio que tiene que ver con la ejecución de esta acción y que ha sido catalogada por los expertos del dominio como una propiedad variable.

Las políticas del negocio definidas en el nivel más abstracto, son convertidas en reglas del negocio cuyo control queda bajo responsabilidad del objeto coordinador. De esta manera, se establece una clara separación entre las propiedades estables y variables de la especificación: las primeras se representan en la especificación de los objetos participantes y las segundas son definidas en la clase coordinadora de la acción. Ambos tipos de propiedades se describen utilizando la riqueza semántica del lenguaje Oasis a través de atributos, precondiciones, reglas de integridad o disparos.

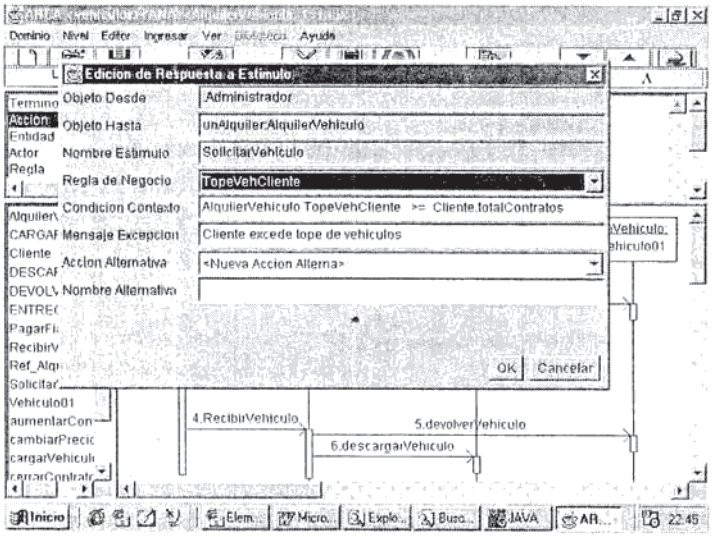


Figura 5. Especificación de una regla del negocio

El incumplimiento de una regla del negocio generará un mensaje dirigido al actor de la acción y opcionalmente se puede declarar un servicio que representa la acción alternativa que debe ser tomada cuando se presenta el incumplimiento de dicha regla. Para dibujar una regla se selecciona el icono que representa la excepción y luego el estímulo al cual se asocia este control. El sistema dibuja entonces un mensaje que irá en sentido contrario al estímulo, es

decir, desde la clase actividad hasta el agente de la acción. Para detallar la excepción se selecciona la línea que la representa y el sistema mostrará entonces un diálogo como el presentado en la figura 5 en el que se detallará la excepción: nombre de la política del negocio asociada que se toma de un área de selección, especificación de la condición, especificación del mensaje que verá el usuario cuando ocurre incumplimiento de la condición y opcionalmente el nombre de la acción alternativa que tomará el componente cuando ocurre la excepción. La figura no.5 presenta la apariencia externa que tendrá la especificación de la regla del negocio TopeVehCliente, asociada al estímulo SdicitarVehículo.

3. El Nivel de Diseño

En este nivel se utilizan los mismos formalismos gráficos del nivel del análisis. Tanto el modelo de colaboración como el modelo de roles de este nivel se generan de manera semiautomática apartir de los modelos del nivel anterior y de las relaciones de refinamiento de tipo funcional y estructural establecidas; dichas relaciones se definen a partir de las primitivas de refinamiento, las cuales determinan el tipo de transformación que se aplica sobre el modelo de análisis para generar el modelo de diseño. Este proceso de transformación controlado permiten establecer la trazabilidad entre los diversos elementos que describen el proceso de construcción del componente.

3.1 Primitivas de Refinamiento Estructural de Clases

Son las primitivas que se aplican sobre las clases en el nivel de análisis con el objetivo de derivar las clases del nivel de diseño; estas primitivas se apoyan en los principios de refinamiento en el área de modelado semántico de datos que tuvieron su origen en el modelo Entidad – Relación; en [BCN92] se proponen primitivas de refinamiento para la construcción de estos modelos de datos bajo un enfoque descendente. Las primitivas de refinamiento son las siguientes:

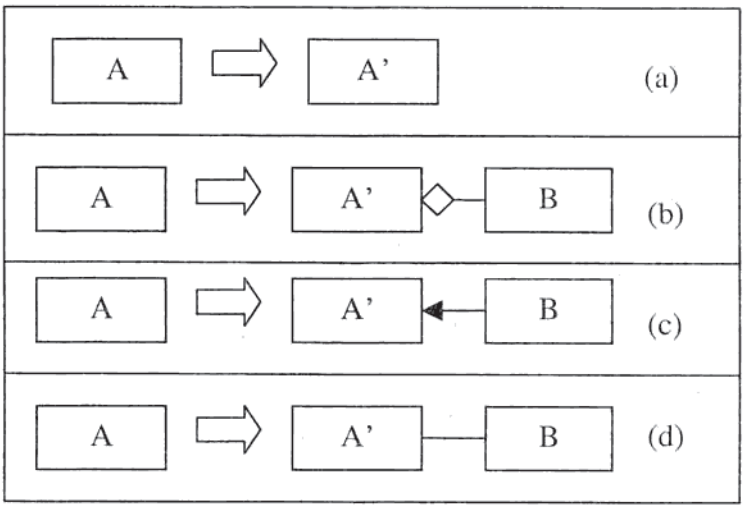


Figura 6 Primitivas de refinamiento estructural

Correspondencia. Primitiva PE1. La clase abstracta A se convierte en una clase concreta A'. Figura 6 (a)

Agregación. Primitiva PE2. La clase abstracta A se convierte en una clase concreta A' más un agregado B. Los atributos definidos en A puede ser atributos derivados a partir de funciones en B. Figura 6 (b)

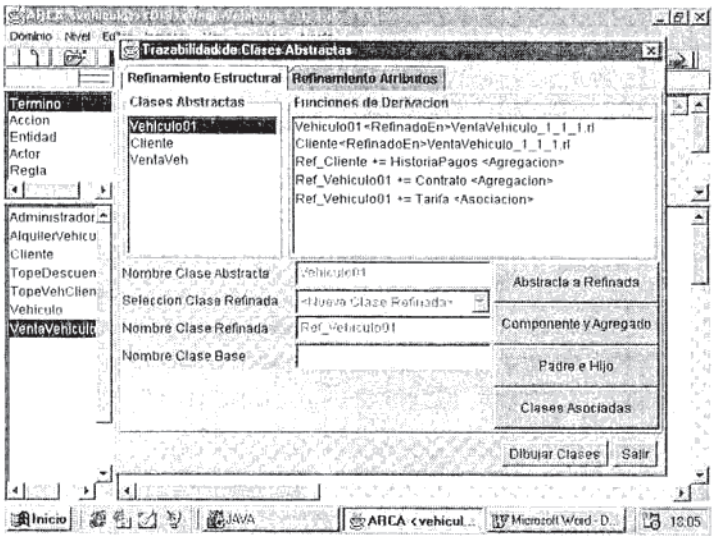


Figura no.7 Refinamiento estructural de clases abstractas

Especilización. Primitiva PE3. La clase abstracta A se convierte en una clase concreta A' más una clase especializada B. Figura 6 (c)

Relación. Primitiva PE4: La clase abstracta A se convierte en una clase concreta A' más una clase relacionada B. Los atributos definidos en A puede ser atributos derivados a partir de funciones en B. Figura 6 (d).

El diálogo *Refinamiento Estructural* presentado en la figura no. 7, permite derivar una clase abstracta (objeto de negocio) en clases concretas (objetos de diseño). Al seleccionar la clase abstracta en la ventana superior izquierda, aparecen en la ventana superior derecha las funciones ya definidas sobre esta clase. En la segunda parte del diálogo se describe la información asociada al refinamiento. El conjunto de botones verticales de las segunda parte del diálogo, implementan las primitivas de refinamiento ya definidas. La opción <Dibujar Clases> sirve para que el sistema haga un dibujo preliminar del modelo de roles refinado que contendrá las clases concretas que el diseñador procederá a organizar.

La figura no. 8 presenta el modelo de roles refinado para nuestro ejemplo. La clase abstracta *Vehiculo01* fue refinada en *Ref_Vehiculo01* (Primitiva PE1) más *Contrato* como clase concreta componente (primitiva PE2) y *Tarifa* como clase concreta relacionada (primitiva PE4). De igual manera la clase abstracta *Cliente* fue refinada en *Ref_Cliente* (primitiva PE1) más *HistoriaPagos* como clase concreta componente (primitva PE2).

3.2 Primitivas de Refinamiento de Atributos

De acuerdo al refinamiento que se aplique sobre las clases abstractas, sus respectivos atributos también pueden ser refinados. Existen dos tipos de refinamiento de atributos:

- Refinamiento de atributos por *derivación*: Se aplica cuando el atributo de la clase abstracta es definido como una función derivada a partir de sus clases refinadas con operadores como suma y promedio. En este ejemplo, el atributo *totalPagado* de la clase abstracta *Cliente,* se convierte en un atributo derivado definido como la sumatoria sobre *valorPagado* del componente *HistoriaPagos*; el atributo tarifa de la clase abstracta *Vehiculo01*, se convierte en un atributo obtenido a través de la relación del *Ref_Vehiculo01* con *Tarifa*.

- Refinamiento de atributos por *composición* : Se aplica cuando el atributo de la clase abstracta representa un concepto general que se detalla en la clase refinada ; el atributo *condiciones* de la clase abstracta *Vehiculo01* fue refinado como la composición de dos atributos : *combustible* y *kilometros*.

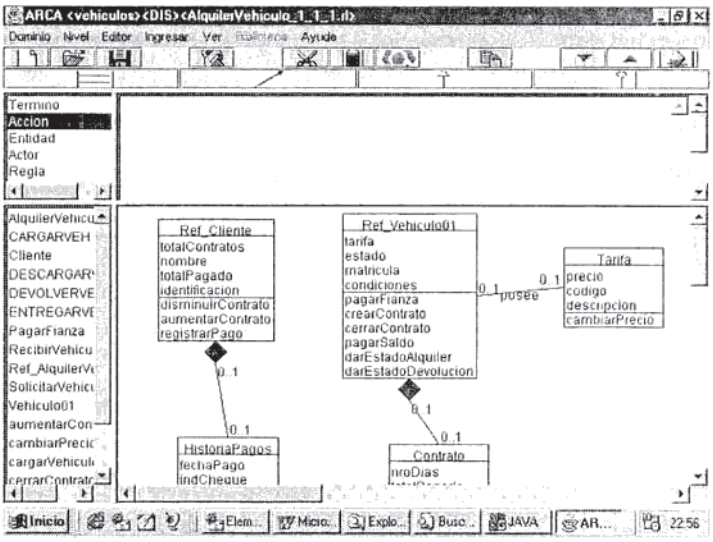


Figura no.8 Modelo de roles refinado

3.3 Primitivas de Refinamiento Funcional

El refinamiento funcional hace referencia al cambio de granularidad de los servicios definidos sobre los objetos abstractos. Un servicio que en un objeto de negocio es visto como un evento de ocurrencia instantánea, se transforma en un conjunto de eventos sobre los objetos de diseño derivados. En la versión 3.0 de Oasis se distingue entre servicios de ejecución instantánea (eventos), servicios que agrupan un conjunto de eventos de ejecución obligatoria (operación) o servicios que agrupan un conjunto de eventos con ejecución opcional (protocolo). Estos dos últimos pueden, a su vez, ser realizados sin observabilidad intermedia en cuyo caso se tendría una transacción [LRS+98]. Se definen entonces tres primitivas de refinamiento desde el punto de vista del comportamiento:

Primitiva PC1: evento abstracto fi protocolo concreto
Primitiva PC2: evento abstracto fi operación concreta
Primitiva PC3: evento abstracto fi transacción concreta

Desde una perspectiva formal este trabajo se apoya la propuesta de **reificación** dada por Denker et.al [Den96], y trasladada al lenguaje Oasis con el propósito de garantizar la consistencia y corrección entre el nivel de análisis y el nivel de diseño [AR99]. Esta propuesta formal ha sido convertida en el soporte metodológico que guía el proceso asistido de refinamiento por pasos implementado en AR_2CA.



Figura no. 9 Trazabilidad de eventos abstractos

La figura no. 9 presenta el diálogo que permite establecer la trazabilidad del comportamiento. El campo de selección superior permite escoger la clase abstracta de la cual se desean refinar comportamientos. Una vez seleccionada un clase abstracta, aparece en la ventana izquierda superior los nombre de eventos abstractos; al seleccionar uno de éstos aparecen en la ventana de la derecha las funciones ya definidas sobre este evento. En el área central de la ventana de diálogo se implementan las primitivas de refinamiento PC1, PC2 y PC3.

Cuando ya se ha realizado la transformación de los eventos abstractos en eventos concretos, se puede activar el botón < Dibujar Diagrama>. El sistema entonces dibuja un diagrama de secuencia en el que aparecen como colaboradores los objetos que son instancia de cada una de las clases concretas. Asimismo en el intercambio de mensajes los eventos abstractos se reemplazan por los eventos concretos que lo conforman. Si se trata de una transacción, ésta se representa como un solo mensaje indicando la propiedad de no observabilidad intermedia.

4. El Nivel de Ejecución

En este nivel el componente se hace ejecutable en el entorno de *JavaBeans*. La traducción del componente a una tecnología específica tiene por objetivo realizar una verificación interactiva

y amigable de su especificación y proveer una versión ejecutable que facilite su entendimiento por parte del diseñador que desee reutilizarlo. Tanto la clase actividad que representa el componente, como los componentes tipo recurso participantes, son definidos como componentes visibles en el entorno de *JavaBeans*, siguiendo los patrones de nombramiento para los métodos de lectura y escritura de propiedades (*accesor method*) y la declaración de clases de interfaz, de tal manera que los componentes pueden ser seleccionados desde el *ToolBox*, instanciados en el *BeanBox* y accedidos utilizando los mecanismos de introspección que posee el API de *JavaBeans*.

La figura no. 10 presenta una vista general del componente que representa la acción *AlquilerVehículo*. El componente exhibirá un panel dividido en tres áreas: *Area de estados* en la parte superior del panel en donde aparece activado el estado actual de la actividad. *Area de Instancias* en la parte intermedia que permite consultar en cualquier momento el estado de los objetos participantes. *Area de Eventos* en la parte inferior, donde se encuentran los estímulos a los que responde la actividad; se encontrarán activos aquellos botones que representan estímulos válidos de acuerdo al estado actual de la actividad. El botón <*Participantes*> servirá para asociar las instancias de las clases recurso involucradas en la acción y es el primer evento que debe ejecutarse.; una vez asociados los participantes, la actividad pasa de su estado de inexistencia (*null*) al estado *inicio*.

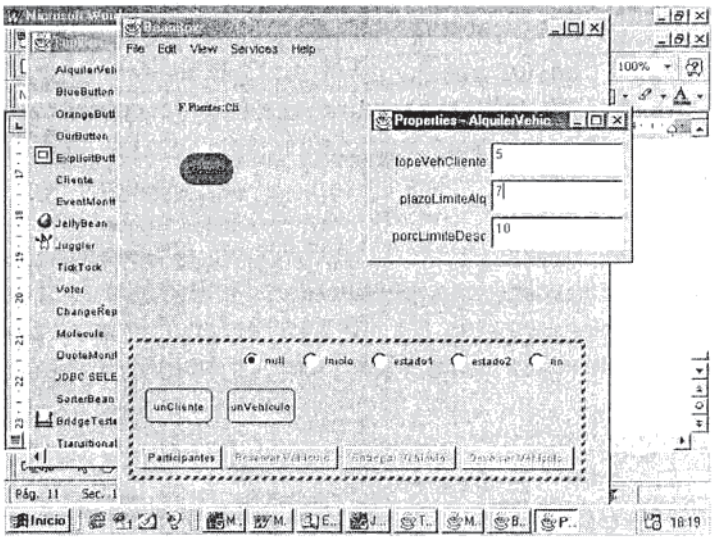


Figura no. 10 Vista general de componentes en el entorno de *JavaBeans*

Los parámetros bajo los cuales va a operar la actividad, que generalmente se encuentran asociados a las reglas del negocio, representan la parte visible del componente que puede ser modificada en tiempo de ejecución de la actividad, utilizando el mecanismo de actualización del entorno de *JavaBeans*.

La ejecución del componente se realiza activando cualquiera de los eventos válidos para el estado actual de la acción. Los botones se activan o desactivan de manera automática y la ejecución del proceso sigue la secuencia definida en la declaración del proceso de la clase actividad. Cuando el evento requiere parámetros de entrada, éstos son capturados antes de la ejecución del evento. Para verificar el efecto que la ejecución de estímulos tiene sobre los objetos participantes, se puede pulsar dos veces sobre el rectángulo que representa el objeto

participante y aparece una ventana que describe el estado del objeto participante y sus objetos agregados.

La implementación de las precondiciones, reglas de integridad y disparos que rigen el comportamiento global entre un objeto de la clase actividad y los objetos participantes y entre éstos y sus objetos agregados, se realiza utilizando el modelo de interacción entre componentes por cambio de estado implementado en el API de *JavaBeans*. Los atributos de los componentes tipo recurso participantes son definidos como propiedades de frontera (*bound properties*) o propiedades de restricción (*constrained properties*); los objetos interesados en controlar el cambio de estado de otros objetos (*VetoableListener*) o de reaccionar ante cambios de estado (*PropertyListener*) se suscriben como objetos receptores y los objetos generadores de estos cambios notifican a los objetos receptores la ocurrencia del evento, los cuales tomarán las acciones correspondientes.

Conclusiones y trabajos futuros

Se ha presentado AR_2CA, la herramienta que guía el proceso de construcción de componentes. En el modelo arquitectónico que describe un componente en AR_2CA se presentan dos tipos de descomposición: la *descomposición vertical* permite un proceso de refinamiento por pasos, asistido y soportado formalmente, que permite ir llevando la especificación de un nivel de abstracción a otro; la *descomposición horizontal* aumenta la capacidad semántica de expresividad en un mismo nivel de abstracción presentando dos vistas complementarias y consistentes en todo momento: la vista de colaboraciones que captura los aspectos dinámico y deóntico del componente y la vista de roles que captura los aspectos estructurales y funcionales [Ana99].

Los niveles de abstracción facilitan el acercamiento progresivo al componente con el propósito de *entenderlo*, estos mismos niveles incrementan las posibilidades de reutilización, puesto que permiten *adaptarlo* en cualquiera de los niveles, definiendo una jerarquía de componentes por nivel/versión. En el nivel de ejecución se pueden modificar las propiedades del componente que reglamentan su comportamiento con el propósito de *parametrizarlo* en tiempo de ejecución.

Esta herramienta ha sido construida en *Java* JDK 1.2, y en esta primera versión se utilizan los servicios de serialización propios de *Java* para almacenar el repositorio de componentes. Una próxima versión de AR2CA debe proveer interfaz a una base de datos para el almacenamiento y consulta de la biblioteca de componentes e debe integrar editores especializados que faciliten la construcción y validación de diferentes tipos de fórmulas complejas.

Bibliografía

[Ana99] Anaya, Raquel. *Un Modelo para el Desarrollo de Componentes Reusables*. Memorias 2^0 Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software. Ed. Centro de Información Tecnológica (CIT), Costa Rica. 1.999

[AR92] Andersen, E.; Reenskaug, T. *System Design by Composing Structures of Interacting Objects*. ECOOP'92. European Conference on Object Oriented Programming. Ed. Springer-Verlag, 1.992

[AR99] Anaya, R.; Ramos, I. *Un Lenguaje Formal para Especificación y Refinamiento de Objetos del Negocio.* Memorias XXV Conferencia Lationamericana de Informática - CLEI99. 1.999

[BCN92] Batini, C.; Ceri, S.; Navathe, S. *Conceptual Database Design : An Entity Relationship Approach.* Ed. Benjamin Cummings, 1.992

[BGK+97] Baumer, D.; Gryczan, G.; Knoll, R.; Lilienthal, C.; Riehle, D.; Zullighoven, H. *Framework Development for Large Systems.* Communications of the ACM. Volumen 40 no. 10, Octubre de 1.997.

[CPT99] Canal, C.; Pimentel E.; Troya, J.M. *Specification of Interacting Software Components: a Case Study.* Memorias 2^0 Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software. Ed. Centro de Información Tecnológica (CIT), Costa Rica. 1.999

[DC98] D'Souza, D., Cameron, A.; *Objects, Components and Framework with UML.* The Catalysis Approach. Addison Wesley, 1.998.

[Den96] Denker, Grit. *Reification - Changing Viewpoint but Preserving Truth.* http://www.cs.tu-bs.de/idb/publications/.

[DIP96] Díaz, O.; Iturrioz, J.; Piattini, M. Stability : *A Criterion for Software Evolution Enhancement.* I Jornadas de Ingeniería del Software, España. 1.998

[DP95] De Antonellis, V.; Pernici, B. *Reusing Specification through Refinement Levels.* Data & Knowledge Engineering . No. 15, 1.995

[FHL+96] Falkenberg, E.; Hesse, W.; Lindgreee, P.; Nilsson, B.; Oei, J.; Rolland, C.;Stamper, R.; Van Assche, F.; Verrijn-Stuart, A.;Voss, K. *FRISCO. A Framework of Information Systems Concepts. The FRISCO Report.* December, 1.996

[FNP92] Fugini, M.; Nierstrasz, O.; Perinici, B. *Application Development through Reuse: The Ithaca Tools Environment.* SIGOIS Bulletin, vol.13, no.2, 1992

[Gru] Gruber, T. http://www-ksl.stanford.edu/kst/what-is-an-ontolofy.html

[Jac93] Jacobson, Ivar, et.al. *Object Oriented Software Engineering : A Use Case Driven Aproach.* Wesley Publishing, 1993

[LRS+98] Letelier, P.; Ramos, I; Sánchez, P.; Pastor, O.. *OASIS version 3.0. Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto.* Universidad Politécnica de Valencia, España. Colección : LIBRO DOCENTE, 1.998

[MT97] Medvidovic, N.; Taylor, R. *A Framework for Classifying and Comparing Architecture Description Languages.* Software Engineering Notes. 6th European Software Engineering Conference, 5th ACM SOGSOFT Symposium. Septiembre, 1997

[NT95]. Nierstrasz, O.; Tsichritzis, D. *Object Oriented Software Composition.* E. Prentice Hall, 1.995

[OH98] Orlafi, R.; Harkey, D. *Cliente/Server Programming with Java and Corba.* John Wiley & Sons, 1998

Reflection in the OASIS object model to solve Schema Evolution Problems

José A. Carsí
Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n, 46071 Valencia
Phone: +34 96 3879721
pcarsi@dsic.upv.es

Isidro Ramos
Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n, 46071 Valencia
Phone: +34 96 3879359
iramos@dsic.upv.es

José H. Canós
Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n, 46071 Valencia
Phone: +34 96 3879359
jhcanos@dsic.upv.es

ABSTRACT

In this paper, we present the extension introduced to the OASIS object model in order to provide solutions for schema evolution problems. The resulting model is able to handle the edition of the schema and the migration of the exiting populations in a simple and elegant way. We present the model and a tool that implement these ideas build on top of a formal system.

Keywords

Evolution, migration, object model, meta model, reflection.

1. INTRODUCTION

Reflective architectures [5] provide a simple and elegant way to cope with some of the classical Software Engineering problems, in particular with software evolution. Reflection consists in constructing systems using several levels, where the dynamics of a level (the base level) is defined as data in the upper one (the meta level); then changes to the behaviour of the base level are implemented as changes to the state in the meta level. There exist different reflection levels [1]: *introspection*, when it is possible to retrieve information about the system; *structural reflection*, if it is possible dynamically to add some components to the system; and *behavioral reflection*, when it is possible to dynamically modify the behaviour of the components.

Our group has been working for years in the definition of the OASIS formal object model [2] and the development of rapid prototyping environments associated to it (e.g. [3]).

In this paper, we describe an OASIS extension consisting of a hierarchy of metaclasses that provide behavioural reflection allowing the definition and evolution of software systems. The OASIS reflective model is outlined in section two. In section three, we describe AFTER, an implementation of the model including mechanisms for object migration through schema versions. To conclude, in section fourth we present the conclusions and some future research work.

2. THE OASIS REFLECTIVE METAMODEL

In the OASIS object model, the universe of discourse is formed by objects. Objects have a state, an independent identifier, offer some services and, in general, they are related with others objects. Systems can be described as objects that interact.

Every object is an instance of a class. The OASIS model allows the specification of the structure and behavior of a class (the *class template*) in detail. The structural aspects are represented, like in other object-oriented models, by attributes (constant, variable and derived attributes); from the behavioral point of view, in addition to the signature of the events, the patterns of *allowed behaviors* of objects are explicitly captured following two dimensions:

- *State dependent specification*: event preconditions and triggering relationships capture the deontic concepts of permission and obligation, respectively.
- *State independent specification*: the allowed patterns of behavior (process definition) are given by using terms of the Basic Process Algebra [4] built over the event alphabet.

The relationship between the structure and the behavior of objects is expressed in the *valuations* section. The valuations specify how the state of an object must be changed when some event occurs.

Complex classes can be defined by means of aggregation and two different kinds of specialization: temporal and permanent. As in other approaches [6, 7], the classes composing an application are viewed as (meta) objects that are instances of some metaclasses.

Figure 1 shows the structure of the OASIS metaclass (OM) hierarchy in UML notation[1]; it is composed of five different metaclasses, one for each different kind of classes defined in the OASIS object model [8]. The *primitive_class* metaclass is used to define Abstract Data Types (ADT), the *elemental_class* metaclass to define simple classes, the *temporal-specialization* and the

[1] The properties of each metaclass are only partially listed in the figure

Carsí, J. A., Ramos, I., Canós, J.H. Reflection in the OASIS object model to solve Schema Evolution Problems. Proc. of the International Symposium on the Principles of Software Evolution (ISPSE2000). 2000.

permanent_specialization metaclasses to define temporal and permanent specializations respectively. Finally, the *aggregation_class* metaclass is used to define aggregations. Characteristics of aggregation relationship are stored in the *characteristics* class.

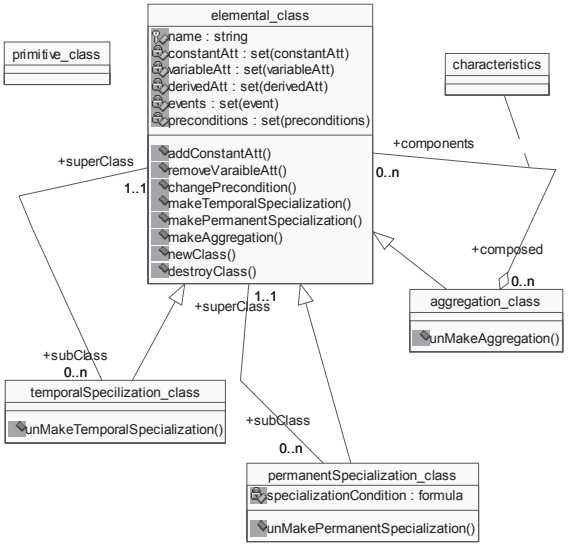


Figure 1: OASIS metaclasses hierarchy

The *elemental_class* metaclass is the root of the OM is-a hierarchy. Other metaclasses (*permanent_specialization*, *temporal_specialization* and *aggregation_class*) are temporal specialization of *elemental_class* because when the user defines a class may change its type and, dynamically, an elemental class can change to be an aggregation or a permanent specialization, etc.

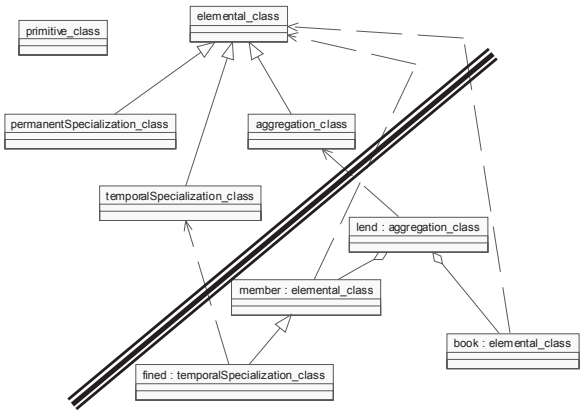


Figure 2: Is-Instance_of relationships (dotted lines) between classes and metaclasses.

Every class in a schema is an instance of some metaclass in our model. Figure 2 depicts the is-instance-of relationships between the classes[2] in a simplified library schema and the metaclasses in OM. On the left side, we can see the metaclasses defined in OM, and, on the right side, we can see the metaobjects defining the library schema. The name of each class in the schema has been suffixed with the name of the metaclass associated to it by the is-instance-of relationship.

A subset T1 of the template defined in the *elemental_class* metaclass includes class properties (namely factory services and the class extent). Another subset T2 defines the template of its instances by means of a set of attributes. Finally, the subset T3 consists of a set of behavioral services implementing the OASIS schema evolution model: change events (some of which are shown in Table 1), evolution invariants (implemented either as event preconditions or class constraints) and an instance migration policy. Note that schema updating is performed via changes in T2 induced by occurrences of events in T3, and object migration as changes in the class extent.

Table 1. Part of the template of an OASIS metaclass

TEMPLATE OF THE METACLASS elemental_class[3]			
Subtemplate	**Property**	**Type**	**MEANING**
T1	*extent*	attribute	extent of the class
	new	event	create an instance of the class
	destroy	event	destroy an instance of the class
T2	*consAtt*	attribute	list of constant attributes
	varAtt	attribute	list of variable attributes
	constraint	attribute	constraints
	event	attribute	list of events
	evPre	attribute	list of event preconditions
	(etc.)		
T3[4]	*addConsAtt*	event	add a constant attribute
	addVarAtt	event	add a variable attribute
	addEvent	event	add an event
	changeConstraint	event	change the class constraint

[2] In this paper we use indistinctly the terms *class* and *metaobject* to refer to OASIS classes

[3] A complete description of the metaclasses' templates can be found in [11]

[4] We only show the signature of the services. Each service has associated a number of preconditions and valuation rules that implement the invariants and the migration policy associated to the service

	remConsAtt	event	remove a constant attribute
	remVarAtt	event	remove a variable attribute
	remEvent	event	remove an event
	(etc.)		

Figure 3 depicts a possible state of the metaobject member; note that, in that state, this class has three instances (member001, member002 and member003). Four attributes have been defined (id_member, name, has_books, and notify), an integrity constraint (has_books < 10), three events (borrow, return and mark), and valuations of every of the events have been defined.

	Member
T1	extent = {member001, member002, member003}
T2	consAtt = { ca(id_member, string), ca(name, string) } varAtt = { va(has_books, integer, 0), va(notify, boolean, false) } constraint={ has_books < 10 } event = { borrow, return, mark } valuations = { val(borrow, has_books=has_books-1), val(return, has_books=has_books+1), val(mark, notify=true) } (etc.)

Figure 3. State of an OASIS metaobject

3. EVOLUTION + MIGRATION IN *AFTER*

AFTER[5] [9] is a TF-logic [10] based implementation of the reflective model presented in section 2. TF-logic is a high-order logic formalism that provides a formal model for object-oriented languages and databases using the traditional concepts of object, logical object identity, inheritance hierarchy and class templates. It is a very interesting formalism to deal with schema evolution since it provides a unique language to declaratively describe and update both data and metadata. Updates to both objects and classes are performed by means of transactions.

3.1 A brief summary of Transaction Frame Logic

TF-logic is the result of the combination of two logics, *Frame Logic* (*F-logic*) and *Transaction Logic* (*T-logic*). The former allows representing the state of object societies, and the latter supports state updates via the execution of *transactions*. Joining them, TF-logic allows defining and evolving object societies by executing transactions that ca be undone through backtracking. For instance, the action:

```
( person[ age => integer ] ).ins
```

adds to the template of the class person the attribute age with type integer. Similarly, the transition:

```
( student :: person ).del
```

removes the class student from the set of subclasses of person.

TF-logic transactions have an atomic formula in the head, a body constructed using F-logic formula and actions combined using the so-called *serial conjunction* connective sc. For instance, the following transaction changes the marital status of an instance of the class person:

```
P1[ marry_to(P2) ] ←
        P1:person sc
        (P1[status->single; spouse->nil]).del sc
        (P1[status->married;  spouse->P2] ).ins .
```

Only persons are allowed to serve this transactions (P1:person). In order to change the marital status of the person P1 the attribute status change from single to married and the attribute spouse is set to P2.

There is no syntactic difference between the invocation of methods and transactions; that provides uniformity on the representation of queries and updates. For instance:

```
?- member[name->Name] % query
?- john[marry_to(anna)]   % execution of a transaction
```

3.2 OASIS translation to TF-logic

AFTER implements OASIS objects as TF-logic object, and OASIS classes as TF-logic classes. As the metaobjects have the two views, they must be represented as objects and classes too.

In order to define the type of the attributes, needed to store all the characteristics of an OASIS specification, several ADT have been defined. In order to describe identifiers: oid and class. To define constant, variable and derived attributes: consAtt, varAtt, and derAtt. To define formulas about the state like preconditions, triggering conditions, constraints: formula. To define valuations and express how the state of the object changes when an event occurs: serialActions. To define events, its preconditions and its valuations: event, evPre, evVal, and so on.

In figure 4 the definition of the ADT serialAction can be seen in TF-logic notation. The ADT serialAction is a subset of the formulas that can be written in TF-logic, restricted to express changes in an object. As can be seen, the ADT serialAction is an instance of the *primitive_class* metaclass (first line). The join of two differents serialActions is also a serialAction (second line). The replacement, insertion and deletion of an attribute are serialActions (3, 4 and 5 lines). Finally, every formula is also a serialAction. In order to apply valuations defined with this ADT over the objects where an event happens, an evaluator has been defined.

```
serialAction : p_c.
sc( A, B ) : serialAction ← A:serialAction sc B:serialAction.
rep( atr(Name,NewValue) ) : serialAction ← atom(Name).
ins( atrs( Name, Value) ) : serialAction ← atom(Name).
del( atrs( Name, Value ) ) : serialAction ← atom(Name).
F : serialAction ← F : formula.
```

Figure 4. ADT serialAction

[5] A Formal Tool for Evolving softwaRe

The metaclasses define the template that metaobjects must have. For instance, in figure 5 the template of the *elemental_class* metaclass is shown.

```
e_c[ name => string ].
e_c[ consAtt =>>[6] consAtt ].
e_c[ varAtt =>> varAtt ].
e_c[ derAtt =>> derAtt ].
e_c[ alias =>> alias ].
e_c[ constraint => formula ].
e_c[ new =>> event ].
e_c[ destroy => event ].
e_c[ event =>> event ].
e_c[ evType =>> evType ].
e_c[ evPre =>> evPre ].
e_c[ evVal =>> evVal ].
e_c[ shared =>> shared ].
e_c[ transaction =>> transaction ].
e_c[ trType =>> trType ].
e_c[ trVal =>> trVal ].
e_c[ trigger =>> trigger ].
e_c[ process =>> process ].
```

Figure 5. Template defined in the *elemental_class* metaclass.

All the instances of this class, like member in figure 6, has a state formed by the values that stores in attributes defined in the template of its class.

```
member : e_c.
member[   name -> "member" ;
          consAtt ->> { ca(id_member, string),
                        ca(name, string) };
          varAtt ->> {  va(has_books, integer, 0),
                        va(notify, boolean, false) };
          constraint -> has_books < 10 ;
          event ->> { borrow, return, mark };
          ... ].
```

Figure 6. Part of the state of the metaobject member.

A view defines the relationship between the template of a class and the state of the metaobject that contains it. The next figure shows the view expressed with deductive rules in TF-logic notation.

```
CLASS[ Att => Type ] ← CLASS : class sc
                       CLASS[ consAtt ->> ca( Att, Type ) ].
CLASS[ Att => Type ] ← CLASS : class sc
                       CLASS[ varAtt ->> va( Att, Type, _ ) ].
CLASS[ Att => Type ] ← CLASS : class sc
                       CLASS[ derAtt ->> da( Att, Type, _ ) ].
```

Figure 7. Deductive rules that implement the relationship between the object part and the class part of a metaobject.

3.3 AFTER evolution services

The AFTER evolution services have been defined as TF-logic transactions using the following pattern:

ClassID[service(Arguments)] ← *precondition sc*
valuations sc
communication.

In the above formula, *sc* is the *serial conjunction* connective. The meaning of the different components of the above transaction pattern is the following:

1) *ClassID* is the identifier of the class that the user wants to change.

2) *Precondition* is a (possibly complex) formula over the service arguments and the metaobject state. The satisfaction of this formulae ensures that the schema invariants (adapted from the ones defined in [12]) will be preserved by the change; it is also used to detect possible inconsistencies that the change could induce in the schema.

3) *Valuations*, on the one hand, modify the template of the class (changing the state of the metaobject); on the other hand, they migrate its instances in order to be compliant with the new template.

4) *Communications* represents the updates that can be triggered to other classes related to the updated one (for instance, via inheritance relationships) in order to maintain the schema consistence.

Above 200 services have been defined in order to cope with all the possible changes that could happen in an OASIS schema. They collect the add/remove/modification of every of the characteristics of an OASIS class (attributes, constraints, events, preconditions, valuations, triggering relationships, transactions, process description, and inheritance and aggregation relationships with other classes in a schema).

When a user wants to define a schema, he/she must obtain a copy of the OM. Executing the creation event of the metaclasses he/she will obtain some metaobjects in which he/she will define the template of some classes using the evolution services. Figure 8 shows a copy of the OM ready to define the schema of a library.

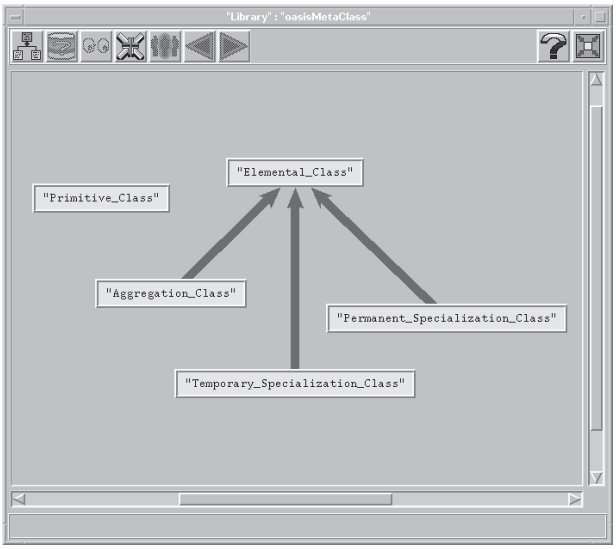


Figure 8. Defining a schema.

[6] The symbol =>> denotes a set-valued attribute.

Figure 9 shows a snapshot of the addition of a variable attribute to a class template. The left window shows the state of the metaobject "Member" and its services. The right window is the dialog box by means of which the user provides the arguments needed to add a variable attribute to this class. For this case, the arguments are: the name of the new attribute, its type, its default value when a new object is created, and a migration action. The migration action will be executed in all the instances of this class. Its intention is to express how to repair the extent of a class in order to be compliant with the new definition of the class.

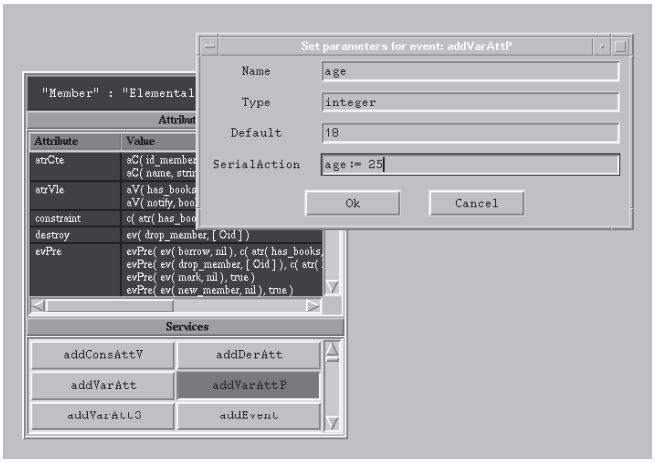


Figure 9. Adding a variable attribute to the class "Member"

As can be seen in Figure 9, the user adds to the class member, an attribute named age with type integer, the default value for this attribute is 18 and, in the extent of the class the new attribute is set to 25[7].

4. CONCLUSION AND FUTURE WORK

In this paper, we have shown the reflective extension to the OASIS model in order to cope with the software evolution problem. A formal implementation of the model in TF-logic (AFTER) supports software definition and maintenance, including object migration, and is completely described in [11]. At this moment, the templates of the metaclasses in OM are fixed.

Research work in progress is to exploit other distinguished features of our model not presented in this paper: first, the process descriptions of the OM metaclasses capture the method to be followed by system designers to build an Information System; this would allow the formal description (and hence the enactment) of software processes. Second, the extent of the classes in OM is the application repository; this fact allows defining a formal framework for software configuration management and reuse. And third, the expressive richness of OASIS would allow the use of other models simultaneously if several metaclasses are defined and included in an integrated software production environment having OASIS as its canonical model.

[7] More complex migration functions can be defined using the ADT serialAction previously defined.

5. REFERENCES

[1] Manola, F., *MetaObject Protocol Concepts for a RISC Object Model*. Technical Report No. TR-0244-12-93-165, GTE Laboratories Inc.

[2] Pastor O. and Ramos, I., *OASIS version 2 (2.2): A Class-Definition Language to Model Information Systems using an Object-Oriented Approach*, SPUPV-95.788, Valencia University of Technology, 1995.

[3] Canós, J.H. et al., *Functional and Object-Oriented Specification of Information Systems*, Proc. of the DEXA-91, Springer-Verlag, August 1991.

[4] Wieringa, R., et al., *Actors, Actions and Initiative in Normative System Specification*. Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

[5] Buschmann F., Meunier R., Rohnert H., Sommerland P., Stal M., *Reflection architectural pattern* in "A System of Patterns", Ed. John Wiley and Sons Ltd, ISBN: 0-471-95869-7, 1996.

[6] Tresch, M. and Scholl, M.H., *Meta Object Management and its Application to Database Evolution*. Proc. Of the 11th International Conference of the ER Approach, October 1992.

[7] Lemke, T., *DDL=DML? An Exercise in Reflective Schema Management for Chimera*. Technical Report IDEA.WP.22.O.003, University of Bonn, March 1995.

[8] Carsí, J. et al., *The OASIS Metaclass*. Proc. Of the Iberoamerican Workshop on Requirements Engineering and Software Environments (IDEAS'98), April 1998 (in Spanish).

[9] Carsí J.A., *A DOOD System for Treating the Schema Evolution Problem*, EDBT'98 Demo Session, VI Intl. Conference on Extending Database Technology, Valencia, March 1998.

[10] Kifer M., *Deductive and Object Data Languages: A Quest for Integration*, Keynote address at the 4-th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, December 1995.

[11] Carsí, J. A., *OASIS as the Conceptual Framework for Software Evolution*. Ph.D. Thesis, Valencia University of Technology, October 1999 (in Spanish).

[12] Banerjee J., Kim W., Kim H.K., Korth H.F., *Semantics and implementation of schema evolution in object-oriented databases*, Proc. of ACM-SIGMOD International Conference on Management of Data, San Francisco CA, May 1987.

Validation of Conceptual Models by Animation in a Scenario-based Approach

Pedro Sánchez **Patricio Letelier** **Isidro Ramos**

Departament of Information Systems and Computation

Valencia University of Technology

Camino de Vera s/n, 46071 Valencia

email {ppalma | letelier | iramos}@dsic.upv.es

Keywords: Conceptual Modeling, Scenarios, Validation, Animation

Abstract

The quality of conceptual models heavily depends on the successfull stakeholder involvement in the requirements engineering process. In this way, the validation of system specifications is a very important task when developing software because it allows developers to be sure they are building the right product. Scenarios are widely used in industry to support requirements elicitation and validation. Tool support is a key factor for effective use and managment of scenarios. With these ideas we have developed OCA, a quality software animation module of OASIS specifications which considers Petri Nets as implementation formalism. OCA is oriented to validation taking into account the concurrency of real systems.

1 Introduction

The feasibility of obtaining quality software specifications is, in a high degree, determined by the involvement of users in the software construction process. This participation can be improved by allowing users to participate in validation tasks. On the one hand, validation process allows the development of quality specifications of software systems [13]. In this sense, the animation of specifications involves identifying scenarios [9] and afterwards validates them. Many methods use scenarios with different names, syntax and semantic and are used for different purposes. Research works shows an important number of scnerario-related methods, models and notations which emphasize the consideration of concrete example during requirements engineering.

OASIS 3.0 (Open and Active Specification of Information Systems) [6] is a formal approach to specify conceptual models following an object oriented approach. Other works with much alike motivation are TROLL [3, 4] and Oblog [12]. Some differences with our work are the formal approach used and the expressiveness and functionality offered by the animation environment.

Our aim has been to produce an environment for developing quality software specifications showing the following characteristics:

1. The environment deals with the primary phases of the development cycle: the most expensive errors are due to weakness and shortcomings in early phases of development process.

2. It will use formalisms in order to suppress or mitigate classical problems of software development and also help in the demonstration of properties as: consistency, correctness, etc.

3. It will support an iterative and incremental development process in order to allow the step by step development of specifications of increasing complexity: elicitation based on real-world examples aims to identify goals underlying the current system [2].

4. The environment is validation oriented and it captures the inherent concurrency of real systems: validation is performed by cooperatively animating a model against instance-level scenarios .

Sánchez, P., Letelier, P., Ramos, I. Validation of Conceptual Models by Animation in an Scenario based approach. Workshop 'Scenario-based round-trip engineering' at OOPSLA 2000. 2000.

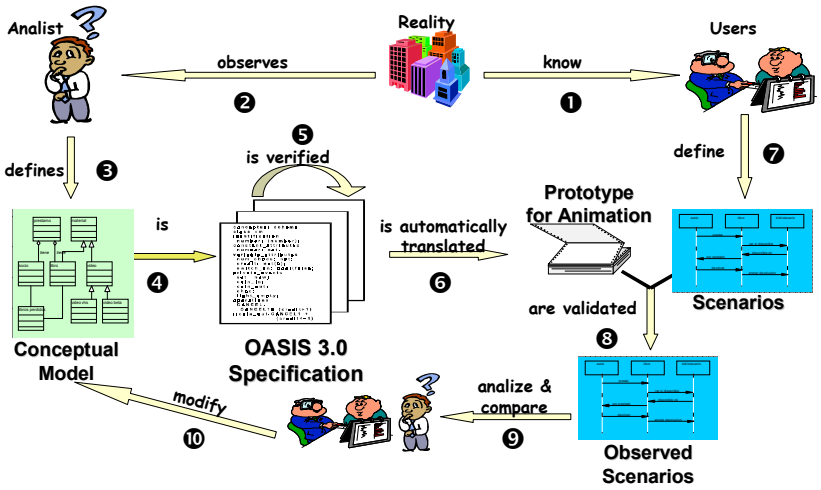


Figure 1: A Validation Process for OASIS.

With this purpose we have designed and developed OCA (**O**ASIS, **C**oncurrency and **A**nimation), a module for validating OASIS specifications oriented to animation. OCA uses Petri Nets as the formalism into which we have implemented the OASIS concepts. In order to achieve these purposes we have needed: (1) to define the OASIS semantics including the concept of step as a snapshot of consistent actions performed in a given instant; (2) to define an execution model [7] implementing the operational semantics of OASIS; and (3) to build a translator from OASIS to Petri Nets.

In this paper we present an integration of two scenario-based techniques. One considers requirements elicitation based on captured real world examples, the other focuses on requirements validation by animating formal specifications. In section 2, we summarize the validation process for OASIS. Afterwards, in section 3 we summarize the state of the art by comparing our approach with others and by pointing out the differences observed. In section 4 we introduce OASIS and its semantics. In section 5 we give a summary of the developed animation environment and, finally, section 6 contains a discussion and the conclusions.

2 The validation process

In Figure 1 we establish the software process followed for the validation of OASIS specifications. At the starting point we find the users who know the reality (system) they want to model. The task of the analysts is to capture the requirements in order to obtain the conceptual model of the system. Conceptual models are expressed in the graphical notation UML, such as class diagrams, state transition diagrams and activity diagrams. A well defined translation process produces OASIS specifications from these graphical representations. When a given OASIS specification is free of syntactic and semantic errors, a prototype with animation purposes can be obtained. Users define a set of representative scenarios describing the expected behavior of the system (using UML sequence diagrams). The execution of the prototype (using the given scenarios as input) gives a result expressed in the same notation. Users and analysts can then compare the behavior differences that will imply modifications in the original conceptual model. This cycle continues until the agreement of users is reached, obtaining in this way the conceptual model *validated* and *verified*. To evaluate if the set of scenarios covers the system functionality is an issue beyond our technique.

3 Related work

Comparing several current CASE environments we can observe deficiencies related with the validation by animation process. Next table summarizes the characteristics of seven CASE tools (plus OCA) which

include a validation module, the first three ones with an academic origin. We only show aspects where validation by animation is concerned. We can establish the following differences with our approach:

- The implementation environment of the prototypes is always sequential been the modeled reality concurrent. Also, the inter-objectual concurrency is simulated and the intra-objectual one is not considered.
- The animation result is not showed in an integrated way for all objects which have participated in the execution.
- There is not formal base nor an execution model is provided.

	Inter & Intra Concurrency	Formal Base	Formal Implementation	Based on Scenarios	Integrated Output
Troll	N	Y	N	N	N
OBLOG	Y	Y	N	N	Y
Albert	N	Y	N	N	N
Aonix	Y	N	N	Y	N
Objectime	N	N	N	Y	Y
Rhapsody	Y	N	N	Y	Y
Bridgepoint	Y	N	N	Y	Y
OCA	Y	Y	Y	Y	Y

4 OASIS

In OASIS [6] a system is seen as a society of concurrent and autonomous objects that can communicate between them by means of actions occurrence. OASIS has been formalized in a variant of Dynamic Logic including deontic aspects (obligation, prohibition and permission) [8]. An OASIS specification is a presentation of a theory in the used formal system and it is expressed as a structured set of class definitions. Classes can be simple or complex. A complex class is defined in terms of other classes (simple or complex) by establishing relationships among classes. These relationships provide aggregation or specialization mechanisms. A class has a name, one or more identification mechanisms for its instances (objects) and a type or template that is shared by every instance belonging to the class. Each object has an unique identifier (*oid*) set by the system, however, objects are referred by their identification mechanisms belonging to the problem space. A function establishes a mapping between the identification mechanisms and the *oid*. The type or template describes the structure and behavior of every object.

As usual in object oriented environments, objects can be seen from two points of view: static and dynamic. From the static perspective, the attributes are properties describing the object structure. The object state in a given instant is the set of structural properties values. From the dynamic perspective, the evolution of objects is characterized by the "change of state" notion. The occurrence of actions implies changes (by means of valuations and derivations) in the values of the attributes. Object activity is determined by a set of rules: preconditions (as forbidden actions in certain states), triggers (as obligations to be fulfilled in certain states), protocols (as allowed sequences of actions in the object life) and operations (as obligated sequences of actions). A step is the consistent set of actions executed at the same instant by the object. In OASIS to specialize a class means to create one or more partitions (statics or dynamics) for it. Each partition is a set of new classes that divides the original one taking into account some criteria. Here is an OASIS specification of a bank account:

Class account
identification
by_number: (number);
constant attributes
number :nat;
variable attributes
balance :nat(0); times :nat(0);
category :string(save);
derived attributes
good_balance :bool(false);
derivations
good_balance:={balance>=5000}
constraints
{sometimes balance>10000}
events
open new; close destroy;
deposit (Qty :nat); reintegro (Qty :nat);
set_category(Categ :string); pay_commision;
valuations
[deposit(Qty)]balance:=balance+Qty, times:=times+1;
[withdraw(Qty)]balance:=balance-Qty,
times:=times+1;

```
 [set_category(Categ)] category := Categ;
 [self:pay_commision]balance:=balance-1000/balance;
 [::pay_commision]times:=0;
preconditions
 withdraw(Qty) if {balance>=Qty}
 close if {balance=0}
triggers
 ::pagar_comision when {times=2 and
            good_balance=false};
protocols
 movements: MOV = open.MOV1;
 MOV1 = deposit.MOV2;
 MOV2 = withdraw.MOV2 + deposit.MOV2 + close;
operations
 forced_close: CLOSE = {balance=0}::close +
      {balance>0}::withdraw(balance).CLOSE1;
 CLOSE1 = ::close;
end class
```

4.1 OASIS Semantics

Dynamic Logic facilitates the incorporation of many concepts of the object-oriented model because it considers state and bevaviour in a declarative way. We have used the variation of Dynamic Logic proposed by Meyer [8], which includes deontic aspects (obligation, prohibition and permission). The declarative semantics of OASIS is given by Kripke structures of possible worlds. An OASIS specification is given by a signature plus a set of Dynamic Logic formulae. A step represents a finite consistent set (it is, any permutation over the actions of any step gives the same final state for the object) of elementary actions that occurs in an atomic way. Properties of an OASIS class can be given using four kinds of Dynamic Logic formulae: permissions, obligations, state changes and assertions[1]. Next table shows a summary of this aspect.

OASIS Concept	Dynamic Logic Formulae
Preconditions	Permissions
Triggers	Obligations
Valuations	State change
Protocols	Permissions + State change
Operations	Obligations + State change
Derivations	Assertions (derivation sem.)
Static Integrity Constraints	Assertions (restriction sem.)
Dynamic Integrity Constraints	All of them

In [10] a procedure to translate protocols, operations and dynamic integrity constraints of OASIS into Dynamic Logic is given. In [10, 11] we have introduced a way to translate OASIS specialization into dynamic logic. The interpretation associated to the occurrence of steps is a calculus which computes a new state starting from another one. This calculus (execution model [7]) represents the operational semantics of OASIS and gives the chance to animate OASIS specifications including Dynamic Logic formulae (permissions, obligations, state changes and assertions). This calculus is correct regarding to the semantics by deduction of Dynamic Logic semantics.

5 OCA: a module for animation

Next we introduce general properties of the developed module with some considerations about the automatically obtained prototype using Petri Nets. We have used the Codesign tool [1] in order to animate the Petri Nets. Some reasons that justify this selection are: the simulation environment is graphical, it is possible to have a good modular decomposition of the system (OASIS) and it has inheritance. OCA has the following features:

- The translator takes as input the set of Dynamic Logic formulae obtained from the graphical specifications. In this way, other OASIS graphical and textual notations could be accepted.
- The output of animation is persistent, so it is possible to validate in one or more animation sessions. Also, it is possible to animate incomplete specifications. This feature allows an incremental software development process.
- Reuse is possible because it is easy to integrate the obtained Petri Nets within other systems using Codesign.

[1]There are two possible semantics for assertions. The first semantics is the classical one in logic and represents a derivation semantics that allows calculating derived properties from others. The second one is the usual one in databases and has associated a restriction semantics: a state will be reachable only if the assertions are accomplished.

The tool Sced [5] represents as sequence diagrams both the scenarios to be validated and the scenarios generated by the animation process. The starting point is the set of Dynamic Logic formulae for an OASIS specifications (The precise mapping between Dynamic Logic formulae and Petri Nets is given in [10]). A translator produces Object Oriented Petri Nets (with Codesign [1] syntax) from Dynamic Logic formulae. The purpose of this translator is to obtain the Petri Nets (as class definitions) needed to animate the considered conceptual model (viewed as an OASIS specification) and taking into account the execution model implementing the operational semantics of Dynamic Logic. Analysts then use Codesign to animate the prototype. From an ideal point of view, analysts have to obtain scenarios semantically equivalent to the input scenarios. The simulation output is mixed in a global interaction diagram (with Sced syntax) including the traces of all objects created during the animation session (including for each object the reached states, the acquired obligations, and the non-satisfied permissions) with comparing purposes. To establish the semantical equivalence between the scenarios used as input and the scenarios obtained by executing the prototype is not a trivial issue; at the moment the comparation process is done manually. However, the more interesting results of this comparison are usually associated to unexpected observed behavior as a consequence of having concurrent objects communicating between them.

6 Conclusions

We have presented the main aspects of an environment focused on the animation of conceptual model specifications written in OASIS. This work fits into the framework of CASE tools although here the emphasis is on the early requirements validation by means on animation using an automatically generated prototype. Our work combines both formal models with scenarios and prototyping techniques. The use of target concurrent formalisms reduce the gap between the specification semantics and the executable representation. The process of assuring the equivalence between the animation and the conceptual model in OASIS presents the same problems as assuring the correction between the design specification and the final code. However, this problem is sorted out thanks to the following features: a) the conceptual model is based on a formal approach which includes an execution model as part of the OASIS semantics and b) the animator is implemented using the Petri Nets formal approach, which would give the chance to formally justify the equivalences between OASIS and the generated prototype.

References

[1] Esser R. An Object Oriented Petri Net Approach to Embedded System Design. Phd. Thesis, Swiss Federal Institute of Technology Zurich, 1996.

[2] Haumer P., Heymans P., Pohl K. *An Integration of Scenario-Based Requirements Elicitation and Validation Techniques*. Technical Report CREWS 98-28, Cooperative Requirements Engineering with Scenarios, 1998.

[3] Herzig R., Gogolla M. An animator for the object specification language TROLL light. Proc. Colloq.on Object-Orientation in Databases and SE, Montreal 1994.

[4] Jungclaus R., Saake G., Hartmann T., Sernadas C.TROLL - A Language for Object-Oriented Specification of Information Systems. ACM Transactions on Information Systems, vol. 14, Number 2, pages 175-211, April 1995.

[5] Koskimies K., Systä T., Tuomi J., Männistö T. Automated Support for Modelling OO Software. IEEE Software, pp. 87-94, 1998.

[6] Letelier P., Ramos I., Sánchez P., Pastor O. OASIS 3.0: Object oriented Conceptual Modeling using a Formal Approach. Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV-98.4011, 1998. (in Spanish)

[7] Letelier P., Sánchez P., Ramos I. Prototyping a requirements specification through an automatically generated concurrent logic program. First International Workshop on Practical Aspects of Declarative Languages (PADL'99), New Mexico, USA, LNCS 1551, pages 31-45, 1999.

[8] Meyer J.-J.Ch. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. In Notre Dame Journal of Formal Logic, vol. 29, pp. 109-136, 1988.

[9] Rolland C., Ben Achour C., Cauvet C., Ralyté J., Sutcliffe A., Maiden N.A.M., Jarke M., Haumer P., Pohl K., Dubois E., Heymans P. A Proposal for a Scenario Classification Framework, Technical Report CREWS 96-01, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm

[10] Sánchez P., Animation of OASIS Specifications using Object Oriented Petri Nets. Doctoral Thesis, Departament of Information Systems and Computation, Valencia University of Technology, April 2000. (in Spanish) http://www.dsic.upv.es/users/oom/docs/tesis-ppalma.pdf

[11] Sánchez P, Letelier P., Ramos I. Animating Formal Specifications with Inheritance in a DL-based Framework, Journal of Requirements Engineering, vol. 4(4), pp. 198-209, Springer-Verlag London Limited, 1999.

[12] Sernadas C., Gouveia P., Sernadas A. Oblog: Object-oriented, logic-based conceptual modelling. Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, 1096 Lisboa, Portugal, 1992.

[13] Siddiqi J., Morrey I.C., Roast C.R., Ozcan M.B. Towards quality requirements via animated formal specifications. Annals of Software Engineering, n.3, 1997.

Full software agents as OASIS 3.0 objects

Adriana Giret, Isidro Ramos

Universidad Politécnica de Valencia
Camino de Vera s/n
Valencia-Spain, 46022
{agiret, iramos}@dsic.upv.es

Abstract. Agent community has proposed and used many formalism to model agents. Many of the formalisms considered so far have focused on just one aspect of agency. However, it is to be expected that any realistic *agent theory* will be represented in a much richer logical framework. Having this in main, the present work aims to analyze OASIS 3.0 as a formal framework for modeling full software agents.

1 Introduction

In the last few years agent technology has received a great deal of attention. As a result, there has been a steadily increasing interest from computer science and other industries in adopting this technology to develop products. In spite of the different agent theories, languages, architectures and successful agent-based applications developed, little effort has been spent on defining formal frameworks to model agents.

In this work we take OASIS 3.0 [8] an OO formal specification language based in Dynamic Logic [5] as formal framework to model agents. In Section 2, we present the basis of agent theory. In Section 3 OASIS 3.0 elements, like concurrent objects, views and observations are presented . Then we made (in Section 4) a first approach to the definition of an agent model, using OASIS as the formal framework. Finally some conclusions are presented.

2 Agent Theory

Agent theory is concerned with the question of what an agent is, and the use of mathematical formalism for representing and reasoning about the properties of agents.

The term “agent” has been so widely used that, according to Finin *et al* [3], anything might qualify as an agent these days. Specialized literature presents a wide range of very different definitions of agents [4]. We have adopted the definition proposed by Wooldridge and Jennings because we believe it includes all the relevant

Giret, A., Ramos, I.Full software agents as OASIS 3.0 objects. Proc. of the 2nd Constraint) Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2001. 2001.

aspects of agenthood [12]. In effect, the authors present two complementary views of agents.

> *A weak definition of agency is that an agent is i) autonomous - agents operate without direct intervention, ii) social - agents interact with other agents, iii) reactive - agents perceive their environment and respond to changes that occur and iv) pro- active - agents affect their environment rather than passively allowing their environment to affect them.*
>
> *A strong definition of agency is that an agent, in addition to the above, has one or more of the following: v) mentalistic notions - an agent has beliefs (knowledge), desires, and intentions (pro-active behavior), vi) rationality - an agent performs actions which further its goals, vii) veracity, and viii) adaptability or learning.*

Agent community has proposed and used many formalism to model agents, some examples are: Possible World Semantics [7], Knowledge and Action [9], BDI [10], etc. Many of the formalisms considered so far have focused on just one aspect of agency. However, it is to be expected that any realistic agent theory will be represented in a much richer logical framework. Having this in main, the present work aims to analyze OASIS 3.0 [8] as a formal framework for modeling full software agents.

3 OASIS 3.0

OASIS is a formal approach for conceptual model specification following the object-oriented approach [8]. The OO formal and declarative framework represented by OASIS and its methodological extension OO-Method, are the result of research done by the Object Oriented Modeling group of the DSIC-UPV[1] during the last few years.

3. 1 The OASIS Model

With OASIS, an information system is understood to be like a society of autonomous and concurrent objects that interact among them by means of actions (see Fig. 1).

An OASIS specification is a presentation of a theory in the used formal system and is expressed as a structured set of class definitions. Classes can be simple or complex. A complex class is defined in terms of other classes (simple or complex) and defined by establishing relationships among these classes. These relationships express aggregation or inheritance. An object is seen as an observable process. Thus, an

[1] DSIC-UPV, Department of Computer Science of the Polytechnic University of Valencia

object has a life characterized by the occurrence of actions, which may either be requested or provided.

Services provided in an atomic level by an object are referred to as events. Each object has a creation event (beginning its life) and optionally a destruction event (finishing its life). Events can be organized at a molecular level as processes. Besides the semantics used when specifying processes, it was added another one in order to make a distinction between prohibition (protocol) and obligation process (operation). An obligation process is a service offered by an object at a higher level. Particularly, a transaction is an obligation process having the implicit all or nothing policy. A prohibition process does not allow the execution of some sequences of actions in an object life. In OASIS, an information system is regarded as a society of autonomous and concurrent objects interacting with each other by the occurrence of actions. An action is a tuple including the client, the server and the requested service.

```
conceptual schema credit_cards
---------------------------------
class owner
identification
        dni:(dni);
constant attributes
        dni:string;
        name:string towards(1,1);
        signature:blob towards(1,1);
        picture:blob towards(1,1);
variable atributes
        email:string;
            telephone_numbers:string
        towards(0,*) list[1..*]
events
        create new;
        delete destroy;
        establish e-mail(text:string);
        add_telephone(text:string) alias for
  insert(telephone_numbers[1],text);
valuations
 [establish_email(text)] email:=text;
end class
---------------------------------
class credit_account
    identification
            number:(number)
    constant attributes
            number:nat
            open_date:date towards(1,1)
    variable attributes
            month_limit:int towards(1,1);
            acumulated_pays:int(0) towards(1,1);
events
            create new;
            delete destroy;
            pay(amount:int);
            establish_limit(amount:int);
valuations
        [pay(amount)]
 total_pays:=acumulated_pays + amount;
 [establish_limit(amount)]
 month_limit:=amount;
end class
---------------------------------
class card_seller
identification
        code:(code);
constant attributes
        code:string;
        name:string towards(1,1);
events
        create new;
        delete destroy;
end class
---------------------------------
credit_account_X_owner aggregation of
      dynamic relational credit_account
 towards(1,1) from(1,1);
 static relational owner towards(1,1)
 from(1,*);
---------------------------------
interface card_seller(someone)
        with credit_account(someone)
        attributes(all)
        services(create)
end interface
```

Fig. 1. An information system in OASIS 3.0

The visibility between objects is determined by an interfacing mechanism. Every object encapsulates its own state and behavior rules. As usual, in object-oriented environments objects can be seen from two points of view; static and dynamic. From the static perspective the attributes are the set of properties describing the object structure. The object state in a definite instant is the set of structural property values. From the dynamic perspective the evolution of objects is characterized by the "change of state" notion. The occurrence of actions implies changes (by means of valuations and derivations) in the values of the attributes. A set of rules; preconditions, integrity constraints, triggers, protocols and operations determine object activity. A step is the set of actions executed at the same instant by the object. Every object has a unique identifier (oid) set by the system. Objects are referred to by one or more identification mechanism belonging to the problem space. An identification function sets up a mapping between the identification mechanisms and the oid. The template describes the structure and behavior of every object.

3. 2 Concurrent Objects

As mentioned previously, an OASIS specification models the Information System like a society of autonomous and concurrent objects that communicate to each other through actions. We are interested in modeling objects as processes according to the OASIS execution model. Modeling objects as processes, in which an object is implemented as a tail-recursive process that passes the update state of the object as arguments in the recursive call [11].

In Concurrent Logic Programming a society of objects can be seen (at run time) as a goal to solve, where each object is a sub goal. Each object is evaluated using AND parallelism (inter-object concurrence).

$\leftarrow object_1(In_1, Out_1, State_1),..., object_k(In_k, Out_k, State_k) \quad ; \; k>0$

In_i is the input channel for receiving actions. It is a merge of the messages received from it and from other objects, Out_i is the output channel for sending actions. $State_i$ is the list of terms (attribute, value) for each attribute of the object. Each object attempts to reduce itself using the clauses that represent its specification. Each clause recognizes one action and is able to reduce the object to a set of subgoals. Here are an object goal and clauses with wich this object goal could be reduced. Considering p,q,s,t ≥ 0 and r>0:

$\leftarrow object_1(In_1, Out_1, State_1), \ldots$

$object_1([action_{11}|In_1, Out_1, State_1) \leftarrow$
$G_{111},\ldots G_{11p}:$
$B_{111},..B_{11t}$
....
$object_r([action_{r1}|In_r, Out_r, State_r) \leftarrow$
$G_{r11},\ldots G_{r1q}:$

B_{r11},..B_{r1s}

3. 3 Schema Evolution

Software evolution is currently one of the most relevant research areas in the Software Engineering field. By evolution we mean not only the modification of the conceptual schema that defines an information system (IS); but also the migration of populations to be compliant with the new schema.

In order to cope with the evolution problem the OASIS object model has been extended in a reflective manner, introducing the meta-class concept [2]. Thereby obtaining, a model with behavioral reflection that allows designers to define, validate and modify software systems. The *Transaction-Frame Logic* (TFL) language [6] was used to define in a formal way the OASIS meta-class.

An OASIS schema can be reified as a meta-object state. This meta-object is composed of other meta-objects, one for each class of the schema. The state of a meta-object defines the structure and the behavior of all the possible units of the associated class. This state can be changed through the services of the meta-object.

Each meta-object has a double view. On the one hand it is an object with a local state, and offers services to define the assembly of properties that constitute the template that will share all the objects that it creates (right side of Fig. 2). On the other hand it is a class, and can create and destroy its instances (left side of Fig. 2).

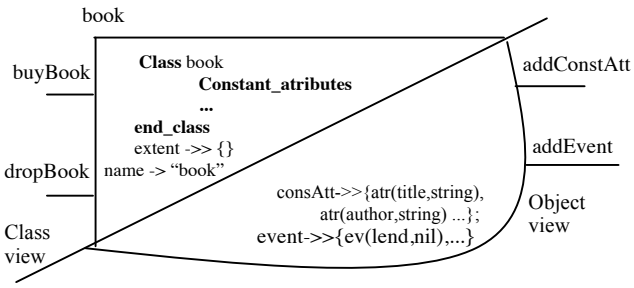


Fig. 2. The metaobject "book"

The meta-class concept in OASIS is defined as a compound class that gathers all the characteristics of the OO model. Schematically, the meta-class will be an abstract editor of OASIS classes, which are created by means of its services. OASIS provides three units of design (Fig. 3) that are: the primitive classes, the elemental classes and the complex classes (aggregation_class, permanentSpecialization_class and temporalSpecialization_class). The meta-class allows the user to construct classes of any of these three categories.

Meta-class services allow us to define and change the characteristics of the classes (through the modification of the state of the meta-objects). The method associated with each service defines the semantics associated with the change it represents, including object evolution in order to be compliant with the new definition. Invariants that must hold are checked as service preconditions or class constraints.

Due to the double view, another important characteristic of the meta-class is that the aspects of schema evolution are totally integrated in the meta-class. For example, this occurs because the firing of the event that adds an attribute to a class can cause the adjustment of all the objects involved (instances of the modified class) to be consistent with the new state (migration of objects).

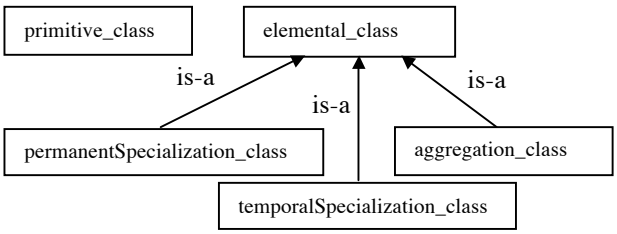


Fig. 3. The OASIS meta-class hierarchy

The OASIS meta-class is the complex meta-object reifying the templates of meta-classes, which allow for the creation of meta-objects. The state of these meta-objects defines the characteristics of primitive, elementary and complex classes. The OASIS meta-class offers as unique services the creation and destruction events of instances. Its population is composed of the set of metaobjects having as state the set of defined schemas, and its template is the composition of the groups of meta-classes.

In order to close the cycle and to avoid having a potentially infinite hierarchy of meta-classes, the reflective closure of meta-class is constructed so that the OASIS meta-class is said to be an instance of itself. At the moment, and because it has so been designated, the structure of the OASIS meta-class is invariable and defines the OASIS model.

3. 4 Views

The definition of views is approached in a very simple way in OASIS. In OASIS, there are the following types of views:

1. Projection Interface: consists of an operation or function that allows the restriction of the set of attributes and/or visible events of a class to a set of users.
2. Specialization Interface: determines the type or population of a class observable by a user.
3. Redefined Interface: allows the incorporation of new attributes and/or events to the class C that is going to be observed.

4. Interface with Derived Properties: new attributes incorporated to the interface can be derived from other attributes.
5. Dynamic Interface: allows the incorporation of the definition of new conditions of firing to the interface. These conditions will be added to the already existing ones and will be activated only when the user, for whom they were defined, acts.

The following is an example of interface definition in OASIS:

interface client **with** atm
import
constant_attributes(atm_type; max_available)
variable_attributes(online)
events(open_door; to_inform)
end_import
end_interface

The previous block in OASIS is an example of projection interface, where it has been defined that the user client has a view of the object atm and that this view contains the attributes and events defined in the **import** section.

3.5 Observations

The objective of any observation is to obtain the value of an attribute pertaining to the signature of the object. When an observation is made the state ot the object does not change.

An observation on an object is an atom of the form: *A ?? B;* where *A* is the oid of an object an *B* is a term whose functor is the symbol of the attribute that is desired to observe.

4 Agent Model in Oasis 3.0

Taking into consideration the characteristics outlined in the previous section, in this section we present a first approach to the definition of an agent model, using OASIS as the formal framework.

In this first approach we try to determine which OASIS elements could be useful to represent each characteristic in the agent model.

The following sections of this chapter on the Agent Model will cover the specification of the entire agent characteristics introduced in the adopted definition.

4.1 Autonomy

An autonomous agent is one that is able and is authorized to work (to reason, to act and to react) in an independent fashion, instead of being directed by other agents. Therefore, an agent is an active software entity, owner of its own thread of control and constantly in execution.

It is possible to view the agent as a perpetual and concurrent process that acts on its environment through communication with all the other software agents and tools. We need a tool to model this fact.

In the specialized literature about concurrency CLP (Concurrent Logic Programming) has been used successfully to model several kinds of systems. Autonomy is a crucial aspect about agents, and CLP seems to be well suited to model it, because of its inherent concurrent nature.

In OASIS [11], by means of CLP (Concurrent Logic Programming) as model of concurrency, the objects are modeled as concurrent and perpetual processes. Adopting this strategy we are able to define the behavior of the agents as follows:

o_agent$_1$*([action*$_1$*|Input], Output, State) :- Guard*$_1$ *|*
    Body$_1$*,*
    o_agent$_1$*(Input, NewOutput, NewState).*
...
o_agent$_i$*([action*$_1$*|Input], Output, State) :- Guard*$_1$ *|*
    Body$_i$*,*
    o_agent$_i$*(Input, NewOutput, NewSate).*

Where *o_agent* is a concrete agent which operates on a list of input events, *Input* , produces a list of output events, *Output*, and has a *State* associated (list of terms **att**(*name, value*)). It has an execution body, *Body*, and recursively invokes to itself (tail recursion).

4.2 Reactivity

A reactive agent simply responds to orders of services. This is the same concept used in the objects oriented paradigms, and it causes a response to their clients' orders of services (other objects).

Therefore, by the process definition of the previous section, it is known that the execution body must contain a block to respond to the invoked methods of the Input list. This execution will be able to produce output events, which will be communicated to the outside through the Output list. Moreover, this execution will have an effect on the change of the agent's state.

We will replace the *Body* section of the agent's block of definition, as follows:

o_agent_i([action1|Input], Output, State) :- Guard1 |
...,
SERVE action1, o_agent_i(Input, NewOutput, NewState).

Where *SERVE accion1* corresponds to the OASIS definition of the execution body.

However, it will be necessary to determine "..." before giving service to the orders of clients. This problem will be addressed in the following section.

4.3 Sociability

The sociability characteristic specifies that an agent must be able to communicate with the other agents and objects of a system. It is the traditional messages passing between objects that are specified in the concurrent model through the *Output* and *Input* lists of the process that models the behavior of the object.

The communication is implemented in the concurrent model through the meta-class (acting as a router), since it knows all the population of the different classes. This idea is represented in Fig. 4.

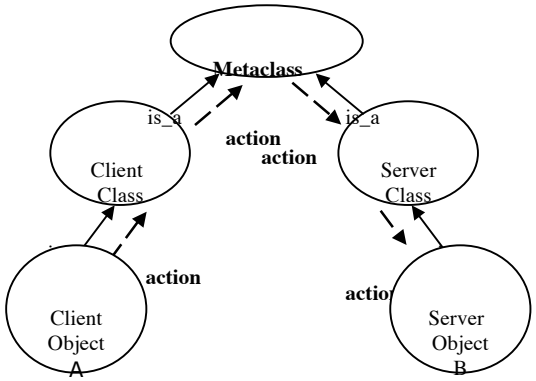


Fig. 4. Inter Object Communication

4.4 Pro-activity

With the pro-activity characteristic the agent, in its own environment, constantly looks for something to do. A very interesting difference from objects is thus introduced. In effect, on object can only change its state as a result of serving the requests of other objects; while an agent, in addition to this, permanently consults its surroundings in order to catch facts or situations that can make it change its state.

It may be useful to define what are the surroundings of an agent. In the section on mentalistic notions it will be clearer, however, we can informally say that the surroundings that the agent perceives, and therefore is able to consult, is a subgroup of the environment in which it moves.

It is therefore clear that we must complete the body of execution of the process, that in the previous section we left incomplete, with a mechanism to query the agent surroundings. Also, we must somehow define these surroundings.

To define the surroundings that the agent perceives we must look for a way to define explicitly what part of that environment will be visible to the agent. This idea may be associated with the notion of data base views. A View is a perspective that allows users to access system objects. The views system provides an automatic security mechanism to hide data in a data base system.

We can consider any agent to be a user of the system (the set of objects of the environment in which the agent moves). So, while we define the views for the different agents, we also specify which objects from its surroundings the agent has permission to query.

It is quite clear that if we denominate *View(O)*, the set of interfaces defined for the object *O* with respect to the different objects *O'*, we have that *View(O)* represents the visible surroundings for the object *O*.

For the query process of the agent's surroundings we can adopt OASIS such as OQL (Object Query Language) as proposed in [1]. The formalism allows an object to query the attributes of other objects, as well as invokes for the execution of events of another object in a very simple way. The syntax is the following:

Observation (query) $\Rightarrow$ *obj ?? atr(V)*, observation of the *V* attribute of the *obj* object.

Actualization (order of execution) $\Rightarrow$ *obj :: ev(arg1, ... , argn)*, invocation of the *ev(arg1, ... , argn)* event of the *obj* object.

Thus the query process of the agent's surroundings, would be modeled as follows:

$\forall O' \in View(O), O' ?? Att$

That is, for all object *O'* that belongs to the *O's* view, query the value of the *Att* attributes.

So, we can enhance the definition of the body of execution of the concurrent process that models the behavior of the agent:

o_agent_i([$\underline{action}_1$|Input],Output,State) :- $Guard_1$ |
$\forall O' \in View(O)$ O' ?? Att, Triggers
Process validation, SERVE $\underline{accion}_1$, o_agent_i(Input, NewOutput, NewSate).

In order to complete this definition, we need to introduce the process validation that will be explained in the learning section.

4.5 Mentalistic Notions

The fact that an agent has mentalistic notions means that it includes, explicitly, a logical model of the world that surrounds it; some authors associate it with the agent ontology. An agreement does not exist about what combination of elements compose the mentalistic notions of an agent; but many authors agree that Beliefs, Desires and Intentions are representative characteristics, so we adopt this approach in the present work.

4.5.1 Beliefs

The beliefs are the facts about the environment that the agent believes. In OASIS an object has an observation state that represents the real values of the different attributes from the objects of the system. *O(A, A')*, is the observation state of the object *O* which includes the values of its own attributes as well as the attributes of the other objects *O'* (where *A* and *A'* are the attribute set of *O* and *O'*).

We defined the beliefs of an agent *O* as a subset of the observation of *O*, that is:

$$Beliefs(O) \subseteq O(A,A')$$

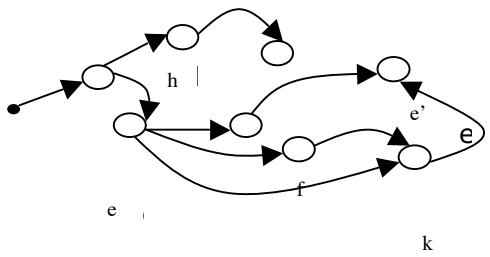
These beliefs will be validated during the continous querying process of the agent (as a pro-active entity).

4.5.2 Desires and Intentions

Desires represent the goals of the agent. Within each goal exists a set of associated plans that determine the steps to follow to reach the goal. The plan that is chosen to be implemented is the intention.

The goals can be seen as states that an agent wants to reach. At every moment the agent is in a given state determined by the value of its attributes. Therefore, at each state we can associate the set of goal states, which the agent wishes to reach from the present one. Consequently, like the objects, an agent has a set of possible lives. Thus we can model it in the same way that it is modeled for objects. This can be done using different formalisms such as Finite State Automata, Petri Nets, etc.. Associated with each change of state there is a transition function that determines the firing pre-conditions and the events that fire when passing from one state to another.

In a simple and intuitive way we can associate with each state the list of goal states in order to model which states we wish to reach from a given state (see Fig. 5).



Desires(e)={e', f, h, k}

Fig. 5. States Graph

In the previous example we associated the *e* state with the goal states *e', f, h* and *k,* for example. Let us suppose that it is desired to reach the goal *e'* from the state *e*; it can be seen that there are several possible paths to get this. Each path constitutes a different plan. To make a decision on a path we can associate with each path a cost in order to measure which of them is the best or, at least, the most suitable one. Thus we have the following:

Plan(e,e') = { (<e,i,e'>,c1),(<e,j,k,e'>,c2),(<e,k,e'>,c3) }

So, for example, if *c1* is better than *c2* and *c3*, then the path < *e, j, é* > is selected as intention.

4.6 Rationality

The rationality characteristic determines that an agent acts to fulfill its goals and will never deviate from its agenda. This characteristic is included in the OASIS dynamic model in which the desires were modeled, since when defining the possible paths or plans of the agent the agent cannot behave in a different way.

4.7 Veracity

The veracity is assured since the querying or observation's semantics are functions, so that at any moment it will return the real value of an attribute and will never be able to deceive with false values.

4.8 Learning

Learning is the characteristic that allows the agent to adapt to its environment, and in addition, to capture knowledge of its surroundings. By learning the agent can modify and reach an evolution of its mental structure as well as its behavior. In OASIS the schema evolution is approached through the meta-class, in this way classes of one schema can be modified in their structure as well as in their behavior using the methods as services.

The agent will be the client of these services of schema modification, allowing it in this way to modify the structure and behavior of the objects with which the agent interacts.

It would seem that we could simply include one Agent meta-class whose instances are Agents classes and which can be instantiated in Agents instances. This, however, is not so easy to do. For example, what happens in a system of seeker agents of the Internet in which each seeker learns the preferences of its particular user? This implies that, if there are several seekers agents, what one of them learns, is not learned by others. Nevertheless, if the concept of class between the seeker agents exists, such as a set of organizations that share a structure and behavior, the others will inherit what is learned by one of them.

This leads us to suppose that agents are templates rather than classes. In effect, when being created agents share the characteristics defined in the template, but after learning things they evolve in an independent way.

In order to cope with this problem, in OASIS it is possible to model the learning capability through the schema evolution mechanism, when the views of the agent are queried. For example, an agent *A* which believes that the object *O* has an attribute valued *N*. During the interaction of all of the system entities the value *N* of the object *O* is changed; so when agent *A* is querying its surroundings, it finds the new value. Depending on its intentions, it may invoke the methods that modify the schema. In Figure 6 there is an example of an agent in TFL [6], in it, we can see that an agent is a client of the methods that modify the schema, so it could modify it.

```
    c(om,e_a)[name => string]; c(om,e_a)[consAtt =>> consAtt];
    c(om,e_a)[varAtt =>> varAtt ]; c(om,e_a)[derAtt =>> derAtt];
    c(om,e_a)[constraint => formula]; c(om,e_a)[new =>> event];c(om,e_a)[destroy  => event];
c(om,e_a)[event  =>> event];
    c(om,e_a)[evType =>> evType]; c(om,e_a)[evPre =>> evPre];
    c(om,e_a)[evVal =>> evVal]; c(om,e_a)[trVal =>> trVal];
    c(om,e_a)[ transaction =>> transaction ];
    c(om,e_a)[trType=>>trType]; c(om,e_a)[trigger=>>trigger];c(om,e_a)[process=>>process];
c(om,e_a)[agent=>>agent];
    agent(Service, ClassList): agent ← Service: sevice & ClassList : list(string)
    lista_agent ->> agent(Service, ClassList); A : e_a
    lista_agent ->> {agent(addConstAtt,A), agent(addVarAtt,A),
                     agent(addDerAtt,A), agent(addEvent,A),
                     agent(addTransaction,A), agent(addTrigger,A),
                     agent(addProcess,A), agent(changeName, A),
                     agent(changeDefValVarAtt,A), agent(changeVarAttType,A),
                     agent(changeDerAtt,A), agent(changeAttName,A),
                     agent(changeConstraint,A), agent(changeEvent,A),
                     agent(changePre,A), agent(changeVal,A),
                     agent(changeNew,A), agent(changeDestroy,A),
                     agent(changeTransaction,A), agent(changeTranBody,A),
                     agent(changeTrigger,A), agent(changeProcess,A),
                     agent(delVarAtt,A), agent(delDerAtt,A),
                     agent(delEvent,A), agent(delNew,A),
                     agent(delDestroy,A),.agent(delTransaction,A),
                     agent(delTrigger,A), agent(delProcess,A)}
```

Fig. 6. Agent Template.

5 Concluding Remarks

Agent design is a complex human activity involving active and proactive modules of software that may include knowledge representation and learning aspects. Agent community has proposed and used many formalism to model agents, but many of the formalisms considered so far have focussed on just one aspect of agency.

In this paper we present a formal specification language OASIS 3.0 [8], that represents agents in a rich logical framework, that captures all the characteristics introduced in the adopted agent definition [12]. The OASIS concurrency model, by means of CLP, was crucial in the specification of the concurrent and perpetual process that defines the behaviour of the agent, without it it could be practically imposible or almost very dificult to express it in a simpler and complete manner. In addition the very clear body structure of the process definition made the other agent's properties implementation very ease.

We can conclude this work saying that an agent model can be defined taking advantage of the different specification formalisms that offers OASIS 3.0.

There are issues remaining for future work. Probably, one of the most interesting one is the inclusion of agents as artifacts in OASIS. Nevertheless it is necessary to study with greater level of detail the inclusion of agents within an OASIS metaclass like structure.

6 References

[1] Canós, H., OASIS: un lenguaje único para Bases de Datos Orientadas a Objetos; Ph.D. Thesis, Supervisor: Dr. Isidro Ramos Salavert, 1996.

[2] Carsí J., Canós J., A Unified Framework for Software Development and Evolution, ASPIRE'99, Advanced modeling and SPecification of distributed InfoRmation systEms Fall Meeting 1999, Linz - Austria, Septiembre 1999.

[3] Finin, T., Labrou, Y., Mayfield, J., KQML As An agent Communication Language, in Jeff Bradshaw (Ed.) "software Agents", MIT Press, Cambridge. Also available at: http://www.cs.umbc.edu/kqml.papers.

[4] Graesser, A. and Franklin S., Is it an Agent, or just a Program?: A taxonomy for Autonomous Agents. Proceedings of the 3º International Workshop on Agent Theories, Architectures and Lenguages, Springer-Verlag, 1996.

[5] Harel D., Dynamic Logic. In Handbook of Philosophical Logic II, editors D. M. Gabbay, F. Guenthner; pages 497-694, Reidel 1984.

[6] Kifer M., Deductive and Object Data Languages: A Quest for Integration, Keynote address at the 4-th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, December 1995.

[7] Kripke, S., Semantical analysis of modal logic. Zeitschrift fur Mathematische Logik und Grudlagen de Mathematik, 9, 1963.

[8] Letelier, P., Ramos, I., Pastor, O., Sánchez, P., Oasis versión 3.0 : un enfoque formal para el modelado conceptual orientado a objeto. –Universidad Politécnica de Valencia,, D.L. 1998. - VII, 184 p. - (SPUPV ; 98.4011).

[9] Moore, R., A formal theory of knowledge and action. In J. R. Hobbs and R.C. Moore, editors, Formal Theories of the Commonsense World. Ablex Publishing Corporation, 1985.

[10] Rao, A. and Georgeff, M., A model-theoretic approach to the verification of situated reasoning systems. In Proceedings of the Thirteenth International Joint conference on Artificial Intelligence (IJCAI-93), pages 318-324, Chambéry, France, 1993.

[11] Sánchez, P., Letelier, P., Ramos, I., Animating Formal Specifications with Inheritance in a DL-based Framework, Requirements Engineering Journal, Springer Verlag London Limited, Vol. 4(4), Pages 198-209, 1999.

[12] Wooldridge, M. and Jennings, N., Agent Theories, Architectures, and Languages: a Survey. In Wooldridge and Jennings Eds., Intelligence Agents, Berlin: Springer-Verlag,Vol. 1,Nº 22, 1995.

Trazabilidad de Requisitos y Calidad del Software: Una Aproximación basada en UML*

Patricio Letelier e Isidro Ramos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{letelier, iramos}@dsic.upv.es

Palabras Clave: Ingeniería de requisitos, trazabilidad de requisitos, gestión de requisitos, UML.

Resumen

La trazabilidad de requisitos permite asegurar la continua concordancia entre los requisitos de los *stakeholders* y los artefactos producidos durante el proceso de desarrollo de software, en particular respecto del código resultante. La trazabilidad de requisitos es considerada un factor clave respecto de la calidad del software y está presente en todos los estándares asociados a la producción de software. Aunque el importante papel de la trazabilidad de requisitos es ampliamente reconocido, el grado de aplicación y consenso en cuanto a prácticas varía considerablemente entre equipos de desarrollo de software. Este trabajo aborda las motivaciones de la trazabilidad de requisitos en el contexto de un proceso de software. Se presenta un metamodelo de referencia para trazabilidad de requisitos que permite integrar tanto especificaciones textuales como elementos de modelado UML, consiguiendo así una representación uniforme para los artefactos y las dependencias de trazabilidad entre ellos. Además, se incluye un pequeño ejemplo para mostrar la aplicación de nuestro enfoque, usando como proceso de software *Rational Unified Process* (RUP).

1. Introducción

El éxito en el desarrollo de software de calidad se basa en cuatro pilares fundamentales: el factor humano (los participantes en el proyecto), la notación y herramientas utilizadas para especificar y construir el sistema, y finalmente el proceso de desarrollo. Respecto del factor humano, ojalá siempre fuera el mejor, y precisamente gracias a ello muchos proyecto llegan a ser exitosos teniendo carencias en los otros aspectos. Además del factor humano, lo técnicamente relevante se concentra en la combinación notación-herramientas-proceso. Mucho se ha avanzado en estos aspectos, sin embargo, por la inmadurez de la ingeniería del software como disciplina, ha sido difícil alcanzar un equilibrio entre ellos. Históricamente las notaciones (por ejemplo las estructuradas y las orientadas a objeto) después de ganar aceptación han incidido en las herramientas CASE que facilitan su aplicación. Posteriormente las propuestas metodológicas han intentado guiar en la explotación de dichas notaciones y herramientas. Pero, la variedad de propuestas y los cambios de enfoque han hecho imposible alcanzar tan deseable consolidación. Actualmente se ha generado una nueva oportunidad, las notaciones

* Este trabajo ha sido financiado por el proyecto *DOLMEN-SIGLO* de la Comisión Interministerial de Ciencia y Tecnología, TIC2000-1673-C06-01.

Letelier, P., Ramos, I. Trazabilidad de Requisitos y Calidad del Software: Una Aproximación basada en UML. En Piattini y García (Eds.), Calidad en el Desarrollo y Mantenimiento del Software. Ra-MA, 2002.

orientadas a objeto más populares se han aglutinado en UML [15] y el soporte para esta notación se ha extendido a la casi totalidad de herramientas CASE. Este fenómeno hace evidente que el gran desafío actual está en el proceso de software. El interés al respecto es claro y se refleja en una efervescente actividad investigadora y el interés por estándares de calidad en procesos desarrollo de software y de su mejora (ISO 9000-3, CMM y SPICE entre otros). Es así reconocido que el proceso de software es factor crítico para la calidad del software.

Un proceso de desarrollo de software determina quién debe hacer qué, cuándo y cómo. El proceso de desarrollo recibe como entrada requisitos nuevos o modificados y genera un sistema nuevo o modificado. Esencialmente el proceso está definido por un modelo de proceso junto con la definición de artefactos, actividades y roles. El modelo de proceso se basa en uno o más de los siguientes enfoques: codificar y reparar (*code-and-fix*), modelo en cascada, desarrollo evolutivo, desarrollo formal (paradigma de programación automática) y desarrollo basado en componentes. Además, existe consenso respecto a que el proceso de desarrollo debe ser iterativo. En este sentido normalmente se combinan una estrategia de desarrollo incremental con una de desarrollo en espiral. Una importante característica de un proceso iterativo es que la especificación del software es desarrollada a lo largo del proceso de desarrollo de software, es decir, no se establece de forma completa e inmutable al principio del desarrollo.

Un artefacto es cualquier información que es usada o producida por el proceso de desarrollo de software [15]. Ejemplos de artefactos son: documentos, modelos, ficheros fuente y ejecutables. Un rol es la definición del comportamiento y responsabilidades de un individuo o conjunto de individuos trabajando juntos como un equipo, dentro del contexto de una organización de ingeniería de software. Una actividad es una unidad de trabajo que puede realizar un determinado rol (de acuerdo con la terminología de RUP[1]).

No existe un proceso de desarrollo de software universal que sea adecuado para cualquier proyecto. Debido a esto, un proceso de desarrollo debe ser entendido como un marco de trabajo configurable, capaz de ser adaptado y escalado según las características del proyecto. En la actualidad quizás los dos exponentes más representativos de procesos, en cuanto a su interés industrial, son RUP y XP (*Extreme Programming*[2]) [2]. Ambos representan la pugna entre lo que se ha clasificado como procesos *peso pesado* (*heavyweigth*) y procesos *pesos ligero* (*ligthweight*, o también llamados metodologías ágiles[3]) [19]. En esta clasificación el término peso pesado se utiliza de manera peyorativa para referirse a RUP y otros similares. La diferencia más clara entre procesos pesados y ligeros está en la envergadura de los proyectos para los cuales están orientados y el grado de "ceremonia" o formalidad que el proceso establece. XP se orienta fuertemente a la producción de código y sus pruebas, restando protagonismo al modelado y enfatizando aspectos tales como: la satisfacción del cliente, el potenciar la capacidad individual y el promover una estrecha colaboración del equipo de desarrollo. En RUP se hace mayor hincapié en el modelado y hay una precisa definición de cada uno de los roles, actividades y artefactos que deben formar el proceso de desarrollo. RUP es un marco de trabajo que puede ser adaptado a las necesidades del

[1] www.rational.com
[2] www.extremeprogramming.org
[3] www.agilealliance.com

proyecto, con lo cual puede configurarse para proyectos de distintas envergaduras. XP debido a sus principios, presenta inconvenientes para ser escalado a grandes proyectos (en condiciones en las cuales, por ejemplo, como se analiza en [18], hay más de 10 participantes, el equipo está distribuido geográficamente, el tiempo de desarrollo es de años, etc.).

Independientemente de las consideraciones de envergadura y otras características particulares del proyecto, el proceso de desarrollo de software siempre debería tener como hilo conductor la trazabilidad de requisitos, la cual permite la conexión y seguimiento de los requisitos hasta su implementación en el software construido. La definición de un proceso de software está orientada esencialmente desde la perspectiva de trazabilidad de requisitos hacia el código que implementa el sistema [3]. La trazabilidad hace que el proceso de software sea observable, es decir, que no sea una caja negra o túnel del cual sólo se conocen sus entradas (requisitos) y salidas (sistema software). Con ello, la trazabilidad de requisitos se define como la habilidad para describir y seguir la vida de un requisito en ambos sentidos, hacia sus orígenes o hacia su implementación, a través de todas las especificaciones[4] generadas durante el proceso de desarrollo de software. La trazabilidad de requisitos ha sido definida en la literatura como un factor de calidad, una característica que un sistema debería incluir como si se tratase de un requisito no-funcional [10]. En el Estándar IEEE 830-1998 "Prácticas recomendadas para Especificación de Requisitos de Software", trazabilidad es una de las ocho características que debe tener una buena especificación de requisitos.



Figura 1: Contexto de la Trazabilidad de Requisitos

Los requisitos del software son susceptibles de cambios, no sólo después de la entrega del producto sino que también durante el proceso de desarrollo iterativo. La gestión de requisitos es el proceso que administra los cambios en los requisitos del software. Gestión de requisitos es un Área Clave de Proceso (*Key Process Area*) necesaria para alcanzar el nivel 2 (nivel repetible) del CMM. Lo anterior refleja la importancia de la gestión de requisitos como subproceso del proceso de desarrollo de software. Para que este subproceso sea efectivo es imprescindible que las relaciones entre los requisitos y otras especificaciones elaboradas durante el proyecto de desarrollo de software estén adecuadamente definidas. Por ejemplo, es importante conocer quién ha propuesto el requisito, qué elementos de modelado (modelos, subsistemas, clases, atributos, asociaciones, etc.) son originados por dicho requisito y qué pruebas lo han verificado y/o validado. La relación entre proceso de desarrollo de software, gestión de requisitos y trazabilidad de requisitos se ilustra en la Figura 1.

[4] De momento, usaremos "especificación" en un sentido general para referirnos a los artefactos del proceso de desarrollo en los diferentes niveles de granularidad y naturaleza, tales como: modelos, diagramas, elementos de modelado, todo tipo de especificaciones textuales con o sin formato, etc.

La gestión de requisitos puede ser un proceso muy costoso, con lo cual, debe planificarse el nivel de detalle que se desea, dependiendo del proyecto. Es primordial que la recolección de información de trazabilidad y su uso sea acorde con las necesidades específicas del proyecto para así conseguir un resultado positivo respecto del costo-beneficio de esta tarea.

Algunos problemas actuales relativos a la trazabilidad de requisitos son [14]:

- La literatura y los estándares no proveen guías detalladas respecto de los tipos de información que debe ser capturada ni del contexto en que debe ser usada. Aunque las herramientas proveen varios tipos de enlaces entre especificaciones, la interpretación de tales enlaces depende del usuario de la herramienta.

- Los diferentes participantes en el proyecto contribuyen a la captura y uso de la información de trazabilidad pero a menudo desde diferentes perspectivas y dando una semántica distinta a los enlaces entre especificaciones.

Pese a estos inconvenientes, la gestión de requisitos debería estar integrada como subproceso dentro del proceso de desarrollo. Pero muy pocas herramientas actuales de desarrollo de software (en el ámbito de herramientas CASE) están orientadas a algún proceso en particular y más difícil aún es encontrar alguna que dentro de un proceso dado provea facilidades para la trazabilidad de requisitos.

Tradicionalmente la especificación de requisitos se ha realizado usando sobre todo especificaciones textuales en lenguaje natural. Consecuentemente, las herramientas de apoyo a la gestión de requisitos se han enfocado a la manipulación de trozos de texto. Estos requisitos expresados textualmente se enlazan formando un grafo de trazabilidad el cual se usa para gestionar los requisitos y su trazabilidad. En este enfoque las especificaciones generadas en las otras actividades del desarrollo de software pueden también ser añadidas al grafo de trazabilidad representándolas como textos (normalmente el nombre de la especificación, por ejemplo el nombre de una clase, atributo u operación). Por otra parte, las especificaciones de pruebas son principalmente textuales, con lo cual también pueden ser tratadas de manera similar. Aunque varios proveedores de herramientas CASE anuncian que sus productos ofrecen una buena integración entre los módulos para requisitos, desarrollo y pruebas, normalmente la solución implementada se basa en mecanismos de importación/exportación entre dichos módulos. Esta estrategia es poco natural de cara a ser promulgarla como parte del proceso de software y debe enfrentar los problemas obvios de sincronización. Lo más adecuado sería ofrecer la posibilidad de trabajar con información de trazabilidad directamente dentro de un entorno integrado para requisitos, desarrollo y pruebas. Por último, no se proveen mecanismos suficientes para la adaptación de las soluciones disponibles a las necesidades específicas de un proyecto, respecto de la granularidad y tipos de información de trazabilidad.

UML ha llegado rápidamente a convertirse en la notación más popular para modelado orientado a objeto. Aunque la mayoría de los conceptos de UML provienen de las propuestas notacionales previamente existentes, el esfuerzo para su establecimiento como un estándar y el consenso asociado a esta iniciativa son remarcables. Gracias a la definición de su metamodelo y a los mecanismos de extensión incluidos, UML ofrece

una excelente oportunidad para establecer un marco común para la representación de especificaciones de requisitos, de desarrollo y de pruebas. La mayoría de las herramientas CASE incluyen la notación UML[5]. Debido a lo anterior, es posible y recomendable integrar la información de trazabilidad en el contexto de UML. Así, aprovechando las facilidades ofrecidas por una herramienta CASE, ésta se podría adaptar para que soporte un proceso de desarrollo de software específico y de paso también un subproceso para trazabilidad de requisitos.

Una vez puesta en contexto la trazabilidad de requisitos en el marco del proceso de software, procederemos a describir un marco integrado para la trazabilidad de requisitos que reconcilia tanto especificaciones textuales como elementos de modelado UML. Este trabajo está organizado en seis secciones. Después de esta introducción, la siguiente sección describe un metamodelo para trazabilidad. La sección tercera explica cómo las especificaciones textuales y lo enlaces de trazabilidad se pueden definir en UML consiguiendo un marco homogéneo para toda la información de trazabilidad. En la sección cuarta se ilustra la aplicación del enfoque propuesto utilizando como ejemplo un proyecto pequeño con RUP. La sección quinta describe algunos trabajos relevantes en el área de trazabilidad de requisitos y algunas herramientas representativas, específicas para la gestión de requisitos. Finalmente la sección sexta presenta las conclusiones.

2. Un Metamodelo para Trazabilidad de Requisitos

Antes de presentar nuestro metamodelo de trazabilidad resumiremos las necesidades de información para la gestión de requisitos. A continuación se indican los tipos de información asociada a la trazabilidad de requisitos y sus posibles usos (adaptado desde [6]):

a) Los enlaces de trazabilidad entre diferentes tipos de especificaciones permiten:
   - Validar que la funcionalidad del sistema reúne las expectativas del cliente y que no se ha implementado funcionalidad superflua.
   - Realizar análisis de impacto de cambios en los requisitos. Los enlaces permiten determinar las especificaciones que se verían afectadas.

b) Las estructuras de contribución [7], es decir, los enlaces entre participantes en el proyecto (*stakeholders*) y las especificaciones permiten:
   - Mejorar la comunicación y cooperación entre los participantes del proyecto. En caso de una solicitud de cambio, los participantes que deberían estar involucrados y/o informados pueden ser determinados fácilmente.
   - Asegurar que la contribución de cada individuo es considerada y registrada.

c) Los fundamentos asociados a las especificaciones, incluyendo alternativas, decisiones, suposiciones, etc. contribuyen a:
   - Mejorar la comprensión del sistema por parte de los *stakeholders* y con ello su aceptación

[5] Aunque ninguna herramienta CASE actual provee UML de manera completa y exacta de acuerdo con el documento oficial de UML y su específica versión.

- Mejorar la gestión de los cambios evitando reconsiderar cuestiones ya antes descartadas, pues las soluciones y sus fundamentos, así como las alternativas descartadas son accesibles

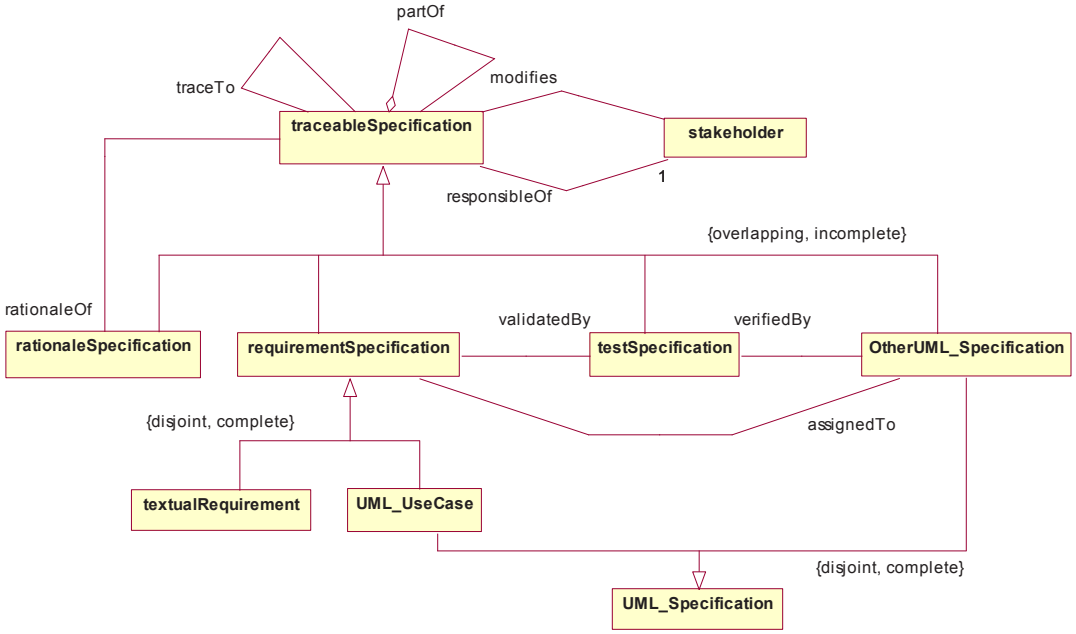


Figura 2: Un metamodelo para trazabilidad de requisitos

Un metamodelo para trazabilidad es una red semántica con nodos representando tipos de entidades[6] y enlaces entre tipos de entidades representando dependencias de trazabilidad[7]. El metamodelo que proponemos se muestra en la Figura 2 mediante un diagrama de clases. Las clases representan los tipos de entidades y las asociaciones los tipos de enlaces de trazabilidad. Se utilizan nombres de roles (*rolnames*) para distinguir los distintos tipos de enlaces de trazabilidad.

En términos generales interesan dos tipos de entidades *TraceableSpecification* y *Stakeholders*. Entendemos por *Stakeholder* a un participante del proyecto de desarrollo de software (sea éste de la parte cliente o usuaria, o del equipo de desarrollo). Los *stakeholders* son responsables de crear y modificar especificaciones. Una *TraceableSpecification* es una especificación de software con un determinado nivel de granuralidad, es decir, puede ser un documento, un modelo, un diagrama, un apartado de un documento, un texto especificando un requisito no-funcional, un caso de uso, una clase, un atributo, etc.). Esta granularidad para una *TraceableSpecification* se define mediante la asociación con nombre de rol *partOf*.

[6] En esta sección hemos preferido utilizar el término "entidad" para referirnos a los nodos. En el trabajo de Ramesh y Jarke [14] se usa "objeto", pero esto podría confundir cuando estamos hablando en un contexto orientado a objeto. En RUP el término sería "artefacto", pero este concepto no coincide exactamente con el homónimo usado en UML 1.4.

[7] Usamos "dependencias de trazabilidad" como tipos de dependencias específicas para trazabilidad de requisitos. Una definición de dependencia más general (extraída de la especificación de UML 1.4), es: "una relación entre dos elementos de modelado, en los cuales un cambio en un elemento de modelado afectará al otro elemento de modelado".

El tipo de entidad *TraceableSpecification* es una generalización de *RationaleSpecification*, *RequirementSpecification*, *TestSpecification*, y *OtherUML_Specification*. Una *TraceableSpecification* puede clasificarse en más de uno de estos subtipos, por ejemplo, cuando se trata de un documento que incluye distintos tipos de especificaciones (mediante *partOf*). Una *RequirementSpecification* es un requisito o grupo de requisitos (entendiendo por requisito la definición de una condición o capacidad que el software debe satisfacer o poseer). Los requisitos en este sentido y según cómo se expresan puede clasificarse en *TextualRequirements* (requisitos expresados mediante un texto) o *UML_UseCase* (elemento de modelado usado en UML para representar un requisito funcional). Con esto se está haciendo una separación entre este elemento de modelado de UML y el resto, reflejados en *OtherUML_Specification* (principalmente son especificaciones de análisis, diseño e implementación, tales como clases o componentes). Por otra parte, una *RationaleSpecification* establece, por ejemplo: fundamentos, alternativas o suposiciones asociadas a una *TraceableSpecification*. Finalmente, una *TestSpecification* define una prueba, ya sea para validar un requisito o para verificar un elemento de modelado UML (por ejemplo: para verificar un fichero con código fuente que es la implementación de una clase o un componente). Establecer que la generalización cuya clase padre es *TraceableSpecification* es incompleta tiene como propósito dejar la posibilidad de tratar con otras formas de información de trazabilidad como por ejemplo: vídeos, imágenes, voz, etc., que pueden resultar útil durante la revisión y evaluación de modelos de análisis y diseño [8].

El enlace más genérico es la asociación con nombre de rol *traceTo* el cual establece relaciones de trazabilidad entre *TraceableSpecifications*. El resto de los enlaces (*modifies*, *responsibleOf*, *rationaleOf*, *validatedBy*, *verifiedBy* y *assignedTo*) constituyen enlaces de trazabilidad más específicos. El enlace *modifies* establece la relación entre los *Stakeholders* y las *TraceableSpecifications* que éstos modifican. Del mismo modo, *responsibleOf* determina el *Stakeholder* que es responsable de la definición y mantenimiento de una *TraceableSpecification*. El enlace *validatedBy* relaciona a las *RequimementsSpecifications* con las correspondientes *TestSpecification* que las validan. Correspondientemente, el enlace *verifiedBy* determina las *TestSpecifications* que verifican una especificación UML. Por último, el enlace *assignedTo* determina a cuáles elementos de modelado UML se les asigna la realización de un determinado requisito, por ejemplo, qué clases realizan un Caso de Uso.

El metamodelo de la Figura 2 cubre las cuatro perspectivas de información de trazabilidad incluidas en el trabajo de Ramesh y Jarke [14]: requisitos, fundamentos (*rationale*), asignación de requisitos a elementos de modelado o implementación, y finalmente, pruebas. En la perspectiva de requisitos, una *RequirementSpecification* es una *TraceableSpecification* que puede tener enlaces hacia o desde otras *TraceableSpecifications*. Normalmente las relaciones de trazabilidad entre *RequirementSpecifications* se dan entre especificaciones en diferentes niveles de abstracción, por ejemplo: necesidades del usuario, características del software y casos de uso UML (de mayor a menor nivel de abstracción). Desde la perspectiva de fundamentos, una *TraceableSpecification* puede estar basada en *RationaleSpecifications*. Respecto de la asignación de requisitos, una *OtherUML_Specification* está asignada a una *RequirementSpecification*. El aspecto pruebas se refleja cuando una *TestSpecification* valida una *RequirementSpecification* o una *OtherUML_Specification* es verificada por una *TestSpecification*.

Por otra parte, nuestro metamodelo incorpora los aspectos de pre-trazabilidad y post-trazabilidad [10,17]. La pre-trazabilidad permite ir desde los orígenes de los requisitos *Stakeholders* hasta su especificación explícita en una especificación de requisitos del software (SRS: *Software Requirement Specification*) o viceversa. La pre-trazabilidad en nuestro metamodelos está soportada por los enlaces *responsibleOf* y *modifies* para determinar los *Stakeholders* involucrados, *traceTo* entre requisitos en diferentes niveles de abstracción y *rationaleOf* asociados a dichas especificaciones de requisitos. La post-trazabilidad permite ir desde la SRS a las posteriores especificaciones de software y las pruebas que las validan/verifican, y viceversa. La post-trazabilidad es soportada en nuestro metamodelo por los enlaces *traceTo*, *validatedBy*, *verifiedBy*, *assignedTo* y *rationaleOf*.

Finalmente, debe destacarse que el metamodelo propuesto integra dos tipos de especificaciones: especificaciones UML estándares (aquellas definidas explícitamente en UML 1.4) y especificaciones que no pertenecen a UML estándar, tales como especificaciones de pruebas, especificaciones de fundamentos y todo tipo de requisitos textuales. En la siguiente sección se explica cómo todas estas especificaciones pueden ser definidas en el marco de UML aprovechando sus mecanismos de extensión.

Trazabilidad Implícita y Explícita

El metamodelo propuesto da soporte para las amplias necesidades de información en trazabilidad de requisitos. Sin embargo, no toda dicha información debe ser explícitamente recolectada y almacenada. Una gran parte de la información de trazabilidad queda implícitamente definida por el proceso de desarrollo, sobre todo en lo que respecta a los enlaces. Cuando en el proceso de desarrollo se establecen los artefactos de entrada y de salida para cada una de las actividades, se están definiendo implícitamente relaciones históricas entre dichos artefactos de entrada y salida, las cuales constituyen en sí enlaces de trazabilidad genéricos, es decir, no referidos a instancias concretas de requisitos o elementos de modelado específicos del sistema. Debido a esto no basta con esta información implícita para realizar una gestión efectiva de requisitos. Además, otra deficiencia que tiene esta información de trazabilidad implícita es que en su mayoría corresponde a un nivel de granularidad grueso.

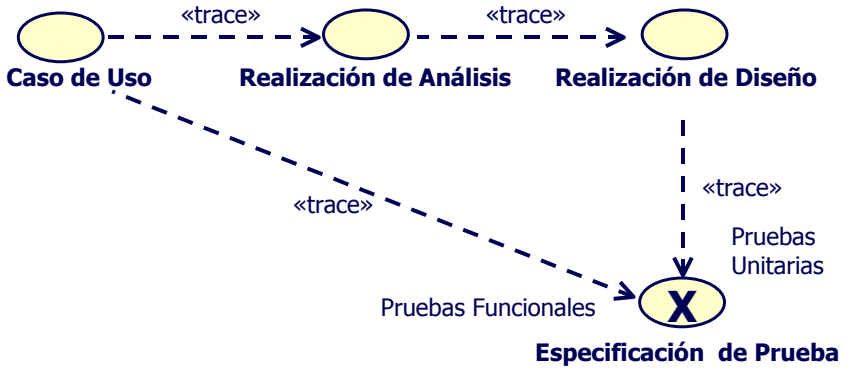


Figura 3: Trazabilidad dirigida por los Casos de Uso en RUP

Ejemplos de información de trazabilidad implícita son (adaptado desde [20]):

- Convenciones de nombre, por ejemplo, una clase en el modelo de diseño recibe el mismo nombre que la clase correspondiente en el modelo de análisis.

- Las correspondencias establecidas entre modelos, por ejemplo, como se muestra en la Figura 3 (adaptada desde [9]), de acuerdo con el enfoque dirigido por los Casos de Uso propuesto en [9] e incorporado en RUP, la unidad de trabajo que integra y dirige el proceso de desarrollo es el Caso de Uso. Mediante Casos de Uso se divide la funcionalidad del sistema, posteriormente para cada Caso de Uso se tiene una realización de análisis (que es opcional), una realización de diseño y especificaciones de prueba tanto unitarias, asociadas a los elementos de diseño e implementación, como funcionales para los Casos de Uso.

- Relaciones entre los elementos de modelado, por ejemplo, las clases que constituyen la realización (de análisis o diseño) de un caso de uso tienen un enlace de trazabilidad al Caso de Uso correspondiente a dicha realización.

- La creación de diferentes perspectivas de un modelo, por ejemplo, una realización (de análisis o diseño) de un Caso de Uso puede estar especificada por clases y asociaciones (en un diagrama de clases) y colaboraciones entre objetos de dichas clases (en diagramas de interacción) ilustrando posibles escenarios. Así, los enlaces de trazabilidad desde el Caso de Uso a las clases que lo realizan, se hacen extensibles a los objetos correspondientes modelados en los diagramas de interacción.

3. El Metamodelo en el marco de UML

Para que la aplicación del metamodelo sea sencilla y práctica es conveniente integrar todos los tipos de entidades y enlaces en un contexto común. Considerando que: (a) las especificaciones UML están definidas con mayor precisión y aceptación que los otros tipos de especificaciones incluidas en el metamodelo, (b) que UML provee mecanismos de extensión (*stereotypes*, *tagged values* y *constraints*) para incorporar nuevos tipos de especificaciones y (c) que las especificaciones UML tienen un amplio soporte en las herramientas CASE, resulta evidente que sería conveniente integrar todos los tipos de especificaciones del metamodelo dentro del contexto de UML. Así, para cada tipo de entidad y tipo de enlace presentes en el metamodelo de trazabilidad se establecerá una correspondencia con un elemento de modelado en UML. En cada caso, se seleccionará una metaclase del metamodelo de UML que se utilizará como clase base para establecer un estereotipo. Para aquellas entidades y relaciones del metamodelo que coincidan semánticamente con una metaclase de UML se utilizará directamente dicha metaclase para representarlos, sin definir un nuevo estereotipo. El resultado de este estudio es un *profile* UML para nuestro metamodelo de trazabilidad. A continuación se detalla cómo se realiza dicha integración.

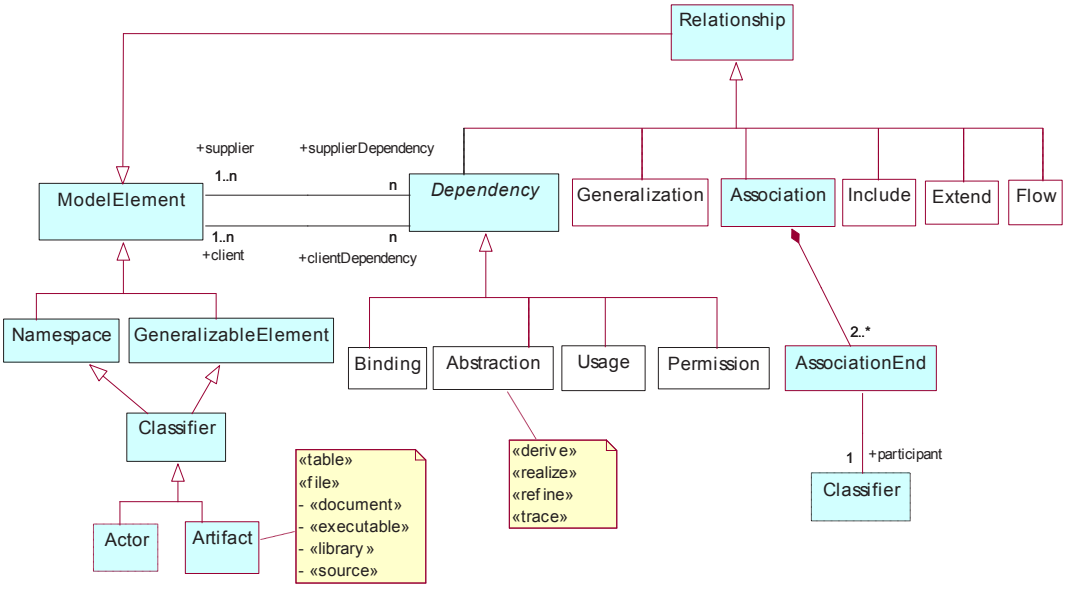


Figura 4: Contexto UML para la integración del metamodelo de trazabilidad

Entidades en UML

Para las entidades del metamodelo el elemento de modelado UML que las represente debería permitir asociaciones para poder establecer relaciones de agregación y composición entre especificaciones. De acuerdo con esto, la elección debería estar entre los elementos de modelado UML que son especialización de *Classifier*[8]. Para la entidad *Stakeholder* la elección es directa; el elemento de modelado *Actor* se utilizó como clase base para el estereotipo. Para las entidades correspondientes a especificaciones no-UML estándar se seleccionó el *Classifier* llamado *Artifact* [9](añadido en la versión 1.4 de UML). Nótese que *Artifact* ya tiene definidos algunos estereotipos, entre ellos «*document*» que es el que utilizaremos para representar documentos y secciones de documentos. Para las entidades que se corresponden directamente con elementos de modelado de UML (*UML_UseCase* y *OtherUML_Specification*) se utilizará el correspondiente elemento de modelado UML para representarlas. Por otra parte, para la agrupación y organización de artefactos UML y *Stakeholders* utilizaremos el elemento de modelado UML provisto para este fin, denominado *Package* y opcionalmente añadiremos los estereotipos predefinidos «*model*» o «*subsystem*» , según estemos definiendo un modelo de un sistema/subsistema o estaremos dividiendo un sistema en subsistemas, respectivamente. La Figura 4 muestra el contexto UML para los elementos de modelado *Actor* y *Artifact*.

[8] En UML 1.4 *Classifier* es una generalización de: *Class*, *UseCase*, *Node*, *Interface*, *Component*, *Artifact*, *Subsystem*, *ClassifierRol*, *ClassifierInState*, *Signal*, *DataType* y *Actor*.
[9] Los elementos de modelado *Comment* o *Constraints* parecían ser los más naturales para representar especificaciones textuales, sin embargo, en UML 1.4 no permiten asociaciones.

Enlaces en UML

Los tipos de enlaces están indicados como asociaciones en nuestro metamodelo de trazabilidad y serán representados como elementos de modelado UML del tipo *Abstraction*, excepto para el caso del enlace *partOf*, el cual se repesentará mediante la agregación o composición entre especificaciones usando como clase base la metaclase *Association*. UML incluye varia dependencias predefinidas mediante estereotipos, éstas pueden verse en la Figura 4 en el comentario conectado con el subtipo de dependencia *Abstraction*. Aunque los distintos tipos de enlaces son modelados por diferentes asociaciones en el metamodelo de trazabilidad, éstas no son independientes, de hecho, el tipo de enlace *traceTo* es la generalización de todos los otros tipos de enlace. Así, *traceTo* se hará coincidir con el estereotipo «*trace*», ya definido de UML. Una dependencia «*trace*» es usada para establecer dependencias de trazabilidad entre elementos de modelado especificados en diferentes modelos[10]. Por lo tanto, exceptuando el tipo de enlace *partOf*, todos los otros enlaces de nuestro metamodelo de trazabilidad serán especializaciones del estereotipo predefinido «*trace*»[11]. En el siguiente apartado se establecerá con mayor detalle la semántica para cada una de dichas dependencias

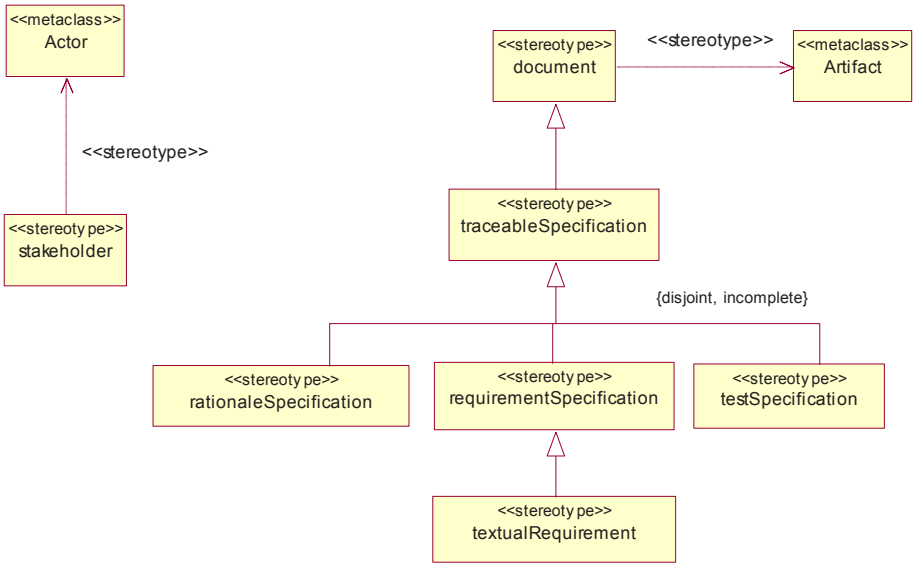


Figura 5: Estereotipos para *stakeholder* y para especificaciones textuales básicas

[10] De acuerdo con UML 1.4, un modelo captura una vista de un sistema (o subsistema) físico, es una abstracción de un sistema físico, para un cierto propósito. Dicho propósito determina qué es incluido en el modelo y qué es irrelevante. Así, el modelo describe completamente todos los aspectos del sistema físico que son relevantes para el propósito del modelo y con un apropiado nivel de detalle.

[11] Según UML 1.4, «*trace*» es una dependencia que indica una relación histórica o de proceso entre dos elementos que representan el mismo concepto sin especificar reglas de derivación entre ellos.

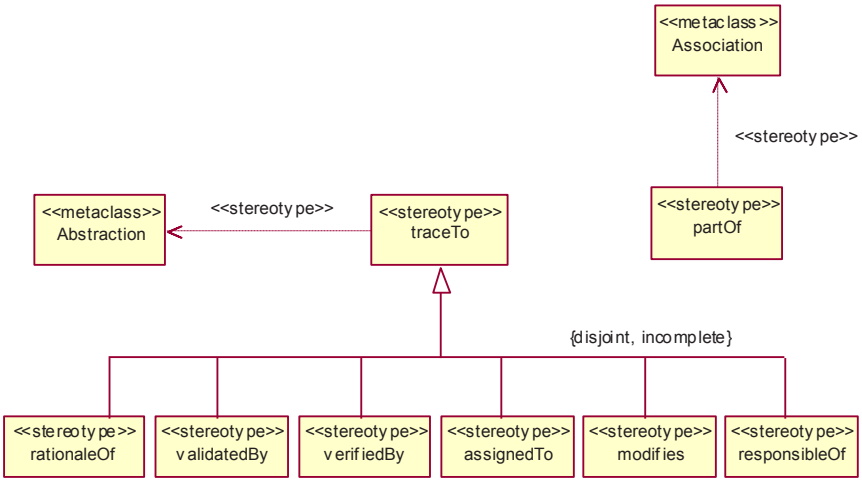


Figura 6: Estereotipos para tipos de enlaces de trazabilidad

Semántica para Entidades y Enlaces

En la Figura 5 y Figura 6 se muestra la representación UML de los tipos de entidades y los tipos de enlaces del metamodelo de trazabilidad, respectivamente. Para el caso de las entidades, y de su representación como artefactos o actores, la semántica es esencialmente la misma que la utilizada en UML para las correspondientes clases base. En cuanto a las dependencias, éstas necesitan ser definidas para evitar ambigüedades respecto de su clasificación.

En cada uno de los tipos de dependencia de trazabilidad de nuestro metamodelo se ha supuesto una dirección desde un tipo de entidad hacia el extremo donde se ha puesto el nombre de rol para la asociación. Esta dirección será la que tenga la correspondiente relación de dependencia en UML (aunque para efectos de recorrido del grafo de trazabilidad, esto no impedirá la navegación en ambos sentidos). Así se evita tener que definir explícitamente cada dependencia y su inversa. Podemos asumir que una herramienta que implemente el metamodelo permitirá establecer un enlace en uno u otro sentido, teniendo en cuenta que al definirlo en un sentido automáticamente se definirá la dependencia inversa. A continuación se describen cada una de las dependencias.

Estereotipo	Descripción
«traceTo»	Establece una dependencia de trazabilidad general entre dos especificaciones. Se utilizaría como dependencia por defecto cuando no se desee detallar más al respecto o no se haya definido una dependencia más específica. La dependencia se establece desde la especificación previa desde el punto de vista histórico o de proceso, hacia la especificación posterior.
«modifies»	Es una dependencia de trazabilidad desde un *Stakeholder* hacia una especificación que ha sido creada o modificada por dicho *Stakeholder*
«responsibleOf»	Establece una dependencia desde un *Stakeholder* hacia una especificación que está bajo la responsabilidad de dicho *Stakeholder*

«validatedBy»	Determina una dependencia de trazabilidad desde un requisito hacia la especificación de prueba que lo está validando
«verifiedBy»	Define la dependencia de trazabilidad desde una especificación UML hacia la especificación de prueba que la está verificando.
«assignedTo»	Es la dependencia de trazabilidad desde un requisito hacia una especificación UML que tiene asignada la realización de dicho requisito
«rationaleOf»	Define una dependencia de trazabilidad desde un fundamento (*rationale*) hacia la especificación que apoya.

Respecto de la agregación de especificaciones para formar artefactos compuestos, como se comentó antes, se utiliza la metaclase *Association* como clase base para el estereotipo *partOf*, tal como se ilustra en la Figura 6.

Puesto que un proceso de desarrollo de software debe ser configurable para adaptarse a las características particulares de cada proyecto, el metamodelo de trazabilidad también debería permitir una adaptación a las prácticas de trazabilidad que se decida adoptar en un determinado proyecto. Este aspecto se ve favorecido por la integración presentada en el marco de UML. Así, una adaptación o extensión de nuestro metamodelo de trazabilidad es simplemente una modificación del *profile* definido antes, incluyendo, por ejemplo, estereotipos adicionales para enlaces (en el lugar adecuado de la jerarquía propuesta), o bien creando estereotipos especializados de «stakeholder» o en la jerarquía del estereotipo «document». Después de adaptar el *profile* y aprovechando el soporte ofrecido para UML por una herramienta CASE, se dispondría de una versión ajustada del metamodelo. Obviamente, para facilitar la aplicación del metamodelo, sobre todo en lo que respecta a especificaciones textuales lo recomendable sería ofrecer interfaces específicas para su edición. En la siguiente sección se ilustra de manera simplificada cómo el metamodelo podría ser aplicado a un proyecto pequeño y en el marco de RUP.

4. Aplicando el Metamodelo con un proyecto RUP

El metamodelo propuesto es independiente del proceso de desarrollo. Sin embargo, para ilustrar su aplicación hemos elegido RUP como proceso, principalmente por disponer en él de un gran detalle respecto de los artefactos y sus descripciones. RUP es un producto de Rational Software basado en el Proceso Unificado de Desarrollo de Software [9]. En RUP las actividades están organizadas en nueve disciplinas: Modelado del Negocio, Requisitos, Análisis y Diseño, Implementación, Pruebas, Despliegue, Configuración y Gestión de Cambios, Gestión del Proyecto y, finalmente, Entorno. Las seis primeras son consideradas disciplinas básicas y las otras tres de apoyo. Cada una de estas disciplinas tiene un conjunto de artefactos. Dichos artefactos no son de uso exclusivo de una disciplina sino que en general constituyen la entrada o salida (cuando se crean o modifican) para actividades en más de una disciplina. Muchos artefactos de RUP son documentos (especificaciones textuales) a los cuales no se les asocia una representación UML, por ejemplo: Glosario, Visión, Documento Arquitectura del Software, Plan de Desarrollo del Software, etc. Para varios de estos documentos se proporcionan plantillas (Word y HTML). La estrategia planteada por RUP para la gestión de requisitos es utilizar la herramienta RequisitePro para gestionar trozos de texto almacenados en los artefactos que son documentos, permitiendo su marcado y manteniendo el control de

modificaciones sincronizado con su base de datos. El único enlace de trazabilidad que se proporciona de manera automatizada con los modelos almacenados en la herramienta Rational Rose es la conexión con los casos de uso. Cualquier otro enlace de trazabilidad debe hacerse introduciendo textualmente otros elementos de modelado UML en RequisitePro, con las implicaciones de redundancia y problemas de sincronización típicos. Otra alternativa posible sería modelar en diagramas UML las dependencias entre artefactos, dibujándolas una a una, lo cual es inviable, incluso para sistemas pequeños.

En trazabilidad de requisitos podemos identificar dos actividades: (a) configurar la trazabilidad a las necesidades del proyecto y (b) definir y explotar la información de trazabilidad durante el desarrollo y mantenimiento del software. Nos centraremos en la actividad de configuración haciendo uso de nuestro *profile* de trazabilidad (definido por el metamodelo de trazabilidad). El *profile* de trazabilidad actuará como marco para establecer los artefactos que tendrán trazabilidad y las dependencias de trazabilidad entre ellos. Además, al introducir trazabilidad explícita, mediante el *profile* se verificaría que la información de trazabilidad sea consistente con lo establecido (por ejemplo, ciertas dependencias son válidas sólo entre determinados artefactos).

La configuración de trazabilidad para un proyecto incluye las siguientes tareas:

1. Seleccionar el conjunto de artefactos para los cuales se realizará un seguimiento de trazabilidad. En general, se trata de un subconjunto del total de artefactos que se utilizarán en el proyecto. A cada artefacto se le establece un estereotipo del *profile* de trazabilidad (excepto para aquellos que se corresponden directamente con un elemento de modelado UML). Al hacer esto, puede que sea necesario extender el *profile* de trazabilidad definiendo estereotipos más específicos.
2. Definir las relaciones de composición entre artefactos. Esta tarea puede que no sea necesaria si dicha información está predefinida como parte de la descripción de cada artefacto.
3. Establecer las dependencias de trazabilidad de interés para el proyecto. Se determinan las dependencias de trazabilidad que son de interés para pares de artefactos seleccionados en la Tarea 1. En este caso también puede ser necesario extender el *profile* de trazabilidad definiendo dependencias de trazabilidad más específicas.
4. Definir criterios para deducir trazabilidad implícita. Tal como se comentó en la sección 2, definiendo ciertos criterios, es posible capturar implícitamente dependencias de trazabilidad. Por ejemplo, usando una convención de nombres y basándose en una dependencia entre un mensaje de una colaboración y una operación de la clase del objeto receptor del mensaje, si el nombre del mensaje coincide con el nombre de la operación entonces implícitamente se determina dicha dependencia de trazabilidad entre ambos elementos. Otro caso trivial de trazabilidad implícita es la que se puede capturar usando la información de composición de los artefactos. Así, las dependencias de trazabilidad entre elementos de mayor granularidad determinan implícitamente dependencias entre los artefactos que los contienen. Por ejemplo, una dependencia del tipo *Class* «traceTo» *Table* determinará implícitamente una dependencia *Analysis & Design Model* «traceTo» *Data Model* (siendo *Analysis & Design Model* y *Data Model* contenedores para *Class* y *Table*, respectivamente).

A continuación se ilustrará la configuración de trazabilidad para un proyecto pequeño basado en RUP.

Tarea 1: Los artefactos de RUP que utilizaremos en nuestro ejemplo se muestran en la siguiente tabla:

Artefacto RUP	Estereotipo
Vision	«document»
Software Feature	«textualRequirement»
Supplementary Specification	«document»
Non-Functional Requirement	«textualRequirement»
Assumption	«rationaleSpecification»
Use Case Specification	«document»
Flow of Events	«flowOfEvents»
Use Case Model	«model»
Use Case	
Analysis & Design Model	«model»
Class	
Implementation Model	«model»
Component	
Data Model	«model»
Test Case	«testSpecification»

Cuando en la tabla no aparece el estereotipo es porque se utiliza exactamente el mismo elemento de modelado definido en UML. En el caso del estereotipo «model», tal como lo mencionamos anteriormente, se trata de un paquete UML con dicho estereotipo. El estereotipo «flowOfEvents» se introduce como nuevo estereotipo siendo clase hija del estereotipo «traceableSpecification».

Tarea 2: Las relaciones de composición entre los artefactos de nuestro ejemplo son las siguientes:

Vision ◊— *Software Feature*
Vision ◊— *Assumption*
Supplementary Specification ◊— *Non-Functional Requirement*
Use Case Specification ◊— *Flow of Events*
Use Case Model ◊— *Use Case*
Analysis & Design Model ◊— *Class*
Implementation Model ◊— *Component*
Data Model ◊— *Table*

Tarea 3: A continuación se listan las dependencias de trazabilidad de interés en proyecto ejemplo:

Stakeholder «responsibleOf» *RUP Artifact*
Stakeholder «modifies» *RUP Artifact*
Software Feature «traceTo» *Use Case*
Assumption «supports» *Software Feature*

Use Case «traceTo» *Use Case Specification*
Use Case «validatedBy» *Test Case*
Flow of Events «traceTo» *Class*
Class «traceTo» *Component*
Class «traceTo» *Table*
Class «verifiedBy» *Test Case*
Component «verifiedBy» *Test Case*

Hemos utilizado *RUP Artifact* para referirnos a cualquiera de los artefactos RUP seleccionados en la Tarea 1. El estereotipo «supports» se introduce como nuevo estereotipo siendo clase hija del estereotipo «rationaleOf».

Tarea 4: En nuestro ejemplo utilizaremos coincidencia exacta de nombres entre artefactos para determinar dependencias de trazabilidad implícita. Esto se aplicará a los artefactos participantes en las siguientes dependencias:

Use Case «traceTo» *Use Case Specification*
Use Case «validatedBy» *Test Case*
Class «traceTo» *Table*
Class «verifiedBy» *Test Case*
Component «verifiedBy» *Test Case*

Además, se establecerán dependencias de trazabilidad implícitas entre artefactos compuestos cuando existan dependencias de trazabilidad entre sus componentes. En caso de existir más de un tipo de dependencia entre componentes, se establecerá «traceTo» entre los artefactos compuestos.

Nótese que al configurar la información de trazabilidad de acuerdo con las necesidades del proyecto se está configurando en gran medida el proceso de desarrollo (aunque no necesariamente todos los artefactos que se utilicen tienen que ser considerados para el seguimiento de trazabilidad).

Según lo establezca el proceso de gestión de requisitos, en determinados momentos del desarrollo se introducirá la información de trazabilidad explícita, por ejemplo al crear o modificar artefactos, o después establecer una *baseline*[12].

5. Otros Trabajos en Trazabilidad de Requisitos

Metamodelos para Trazabilidad

Ramesh y Jarke en [14] ofrecen una amplia visión de la información necesaria para trazabilidad en proyectos industriales de desarrollo de software. Identifican dos segmentos de usuarios de trazabilidad y sus respectivas necesidades de información de trazabilidad, principalmente caracterizados por el tamaño de los proyectos (proyectos de

[12] Una *baseline* es una instantánea de cada artefacto en el repositorio del proyecto. Esto provee un punto de referencia oficial para volver atrás, para aplicar un protocolo de gestión de cambios, o para establecer la información de trazabilidad.

alrededor de 1000 requisitos en un segmento y proyectos de alrededor de 10000 requisitos en el otro). A partir de esto proponen dos metamodelos de trazabilidad (siendo uno la simplificación del otro). El metamodelo más amplio tiene 31 tipos de entidades (metaclases en el metamodelo de trazabilidad) y alrededor de 50 tipos de enlaces entre entidades. Hay que destacar que todas las especificaciones de análisis y diseño se representan sólo con un tipo de entidad, denominado "*system_subsystem_component*", es decir, no se presenta mayor granularidad ni conexión con alguna notación de modelado específica. Además de la complejidad asociada a la diversidad de elementos, no se provee una definición precisa ni para las entidades ni para los enlaces lo cual hace muy difícil su aplicación. Por otra parte, el metamodelo propuesto no ofrece mecanismos para adaptarlo a necesidades específicas de trazabilidad en un proyecto, y tal como lo proponen los autores dicha adaptación se realizaría rudimentariamente cortando o añadiendo partes del metamodelo.

Toranzo y Castro en [21] proponen un metamodelo para trazabilidad definido mediante múltiples vistas, cada una asociada a un determinado tipo de usuario de la información de trazabilidad (*Project Manager*, *Requirement Engineer* o *Software Engineer*). Sin embargo, tampoco en este trabajo se proveen mecanismos de extensión o adaptación del metamodelo. Además, el nivel de granularidad de los artefactos es muy grueso, trabajando en el ámbito de documentos y diagramas.

Spence y Probasco en [20] presentan varias estrategias de trazabilidad aplicables cuando se trabaja con un proceso dirigido por los Casos de Uso. Cada estrategia es descrita con un metamodelo para trazabilidad, estableciendo artefactos y dependencias de trazabilidad entre ellos. Todas las estrategias propuestas sólo ilustran dependencias de trazabilidad entre artefactos de requisitos, la trazabilidad hacia otros tipos de especificaciones se deja implícita al nivel de granuralidad que por defecto tiene un proceso dirigido por los Casos de Uso, es decir, trazabilidad desde un Caso de Uso hacia su realización en Análisis/Diseño y hacia el Caso de Prueba correspondiente. Similarmente, Leite et al. en [11,12] proponen un marco de trabajo para la captura y organización de requisitos expresados en lenguaje natural. Se establecen enlaces de trazabilidad entre los requisitos registrados pero tampoco se incluye trazabilidad hacia otros tipos de artefactos.

Trazabilidad en Herramientas para la Gestión de Requisitos

En general, las herramientas para gestión de requisitos[13] ofrecen un tratamiento satisfactorio para las especificaciones textuales pero presentan inconvenientes cuando se desea integrar dichas especificaciones con otras de análisis, diseño o implementación. Estas herramientas no suelen estar orientadas a un proceso de software ni a una notación determinada. Respecto de la integración con otros entornos (por ejemplo herramientas de modelado), normalmente esto se provee mediante importación/exportación de datos con diferentes formatos.

Una herramienta del ámbito académico que constituye un punto de referencia obligado en trazabilidad de requisitos es la presentada por Pinheiro y Goguen en [16]. La

[13] Para obtener un análisis detallado del estado del arte de las herramientas para gestión de requisitos recomendamos visitar el sitio web del INCOSE (Internacional Council On System Engineering) `www.incose.org/tools/tooltax.html`

herramienta TOOR (*Traceability of Object-Oriented Requirements*) está basada FOOPS, un lenguaje formal orientado a objeto. El usuario de la herramienta debe tener un entrenamiento previo en especificaciones FOOPS. Curiosamente, no se registra una continuación de TOOR posterior a dicha publicación, sin embargo, el enfoque formal utilizado y la funcionalidad descrita mantienen vigencia.

En Rational RequisitePro[14] se pueden enlazar especificaciones textuales no-UML con especificaciones UML dentro del repositorio de Rational Rose. Sin embargo, esto es posible sólo para Casos de Uso, es decir, el resto de los elementos de modelado dentro de los modelos en Rational Rose no son accesibles directamente desde RequisitePro. Mediante un lenguaje disponible para desarrollar extensiones podría mejorarse este aspecto. La representación textual de los elementos de modelado y su introducción en RequisitePro supone una redundancia que es costosa de controlar. Por otra parte, en RequisitePro sólo existe un tipo de enlace, que corresponde la dependencia genérica *trace*. Otra estrategia de integración entre una herramienta específica para gestión de requisitos y una herramienta CASE es la seguida por Telelogic DOORS[15]. Esta herramienta permite la conexión con diversas herramientas CASE para importar los elementos de modelado de los modelos en el repositorio de la CASE hacia el entorno de gestión de requisitos. En este caso está el inconveniente de tener que cambiar de entorno según se quiera gestionar requisitos o modelar y construir el software.

Recuperación Automática de Información de Trazabilidad

El gran esfuerzo de recolección y mantenimiento de la información de trazabilidad hace atractivas las propuestas para captura automática o semiautomática de dicha información. En esta línea en [4] se propone un enfoque para transformar especificaciones textuales en representaciones formales introduciendo una terminología entre ambas. Los términos pueden ser empleados para enlazar conceptos y fuentes textuales actuando como estructuras de navegación. En el otro extremo de la información de trazabilidad (el de la implementación), en [1] se aborda el problema de recuperar automáticamente enlaces de trazabilidad entre entidades orientadas a objeto de diseño y de código. Este enfoque resulta interesante cuando no se provee un soporte explícito para trazabilidad en el proceso de desarrollo. La recuperación de información de trazabilidad puede conducir la evolución del diseño disponible para que éste se corresponda con el código proporcionando una vista de más alto nivel del sistema. Por otra parte, en [5] se propone un enfoque para detectar información de trazabilidad entre el sistema y sus modelos basado en la observación y análisis de información en tiempo de ejecución del software. La técnica consiste en evaluar la huella que los escenarios (expresados en los casos de prueba) dejan durante la ejecución. Aquellas huellas se comparan para obtener información de trazabilidad entre los elementos de modelado asociados a los correspondientes escenarios. Finalmente, siguiendo la misma dirección hacia la automatización, pero desde una perspectiva más ambiciosa, hacer realidad el paradigma de programación automática de Balzer reduciría el esfuerzo en recolección y mantenimiento de la información de trazabilidad, no sólo por el hecho de automatizar estas tareas sino debido a la drástica eliminación de artefactos necesarios para el desarrollo de software. Una vez conseguido un contexto de compiladores de modelos o generación automática de código a partir de modelos, las actividades de verificación y

[14] www.rational.com
[15] www.telelogic.com

de transformación de modelos de alto nivel hacia otros de implementación y código serían trasparentes para el desarrollador.

6. Conclusiones

La trazabilidad de requisitos es la clave para lograr un proceso de gestión de requisitos exitoso y por consiguiente asegurar en gran medida la calidad del software. Sin embargo, no existe consenso respecto de las estrategias adecuadas para lograr una trazabilidad efectiva de requisitos. Estos y otros inconvenientes hacen que en la práctica la trazabilidad de requisitos en proyectos de software presente diferentes grados de uso y aceptación [13]. Consecuentemente, el soporte para trazabilidad en herramientas CASE no es el más adecuado. Históricamente, la investigación en gestión de requisitos ha ido en paralelo con la investigación en enfoques para desarrollo de software creando dos comunidades de investigación diferentes: la primera enfocada a mejorar el tratamiento de especificaciones textuales de requisitos sin entrar en detalles respecto de los elementos de modelado particulares de alguna propuesta metodológica, la otra enfocada a mejorar los métodos de análisis y diseño con técnicas estructuradas u orientadas a objeto, pero sin prestar demasiada atención a la gestión de requisitos ni a las especificaciones textuales. En la actualidad UML representa una buena oportunidad para aprovechar los resultados de estas dos comunidades y trasladarlos a un marco común para la especificación de trazabilidad y de su explotación.

En este trabajo hemos presentado un metamodelo de trazabilidad que integra especificaciones textuales (para requisitos, fundamentos y pruebas) con especificaciones UML estándares, usando el propio contexto de UML. Así, desde el punto de vista de la información necesaria para trazabilidad de requisitos, el metamodelo ofrece un marco esencial de entidades y enlaces (artefactos y dependencias) que puede ser adaptado a un proyecto específico usando los mecanismos de extensibilidad provistos por UML. Gracias a estar definido en UML, el metamodelo debería poder ser implantado en cualquier herramienta CASE que dé soporte a UML. El metamodelo para trazabilidad ha sido definido como un *profile* UML con lo cual puede ser incorporado y adaptado como una plantilla de proyecto en una herramienta CASE. Adicionalmente, el subproceso de trazabilidad debería estar enmarcado dentro del proceso de software. Pero no existe un proceso universal y como consecuencia las herramientas CASE no suelen estar orientadas a un proceso específico. Aunque en teoría bastaría con los mecanismos de extensibilidad de UML para adaptar la trazabilidad a las características del proyecto, en la práctica es necesario hacer cierta adaptación de la herramienta CASE para ofrecer un acceso expedito a la información de trazabilidad junto con proveer un editor ad hoc para especificaciones textuales. Actualmente estamos trabajando en el desarrollo de un módulo para implementar nuestro metamodelo extendiendo Rational Rose y siguiendo el proceso RUP.

Las nuevas tendencias metodológicas con el calificativo de ágiles (tales como *Extreme Programming*) a primera vista parecen ser inmunes a la problemática de los procesos más ceremoniosos (tales como RUP) y por consecuencia a temas como la trazabilidad de requisitos. Sin embargo, salvo en proyectos muy pequeños, en los cuales se pueda prescindir de todos los artefactos adicionales al código, en cualquier otro proyecto al

menos se tendrán los básicos artefactos de requisitos y de pruebas, y por supuesto el código. Pues bien, aún en este contexto tan minimalista ya es necesario realizar alguna gestión de los requisitos mediante trazabilidad entre requisitos, pruebas y código.

Un aspecto importante en la configuración de la trazabilidad del proyecto es el establecimiento de atributos de trazabilidad (y sus posibles valores) para cada uno de los artefactos seleccionados. Intencionalmente este tema no ha sido abordado porque consideramos que dicha tarea no presenta especial dificultad. Puede suponerse que la herramienta de gestión de requisitos ofrecerá un conjunto de atributos predefinidos para seleccionar y asociar a cada artefacto, y que también permitirá definir nuevos atributos. Por ejemplo, en RUP para una *software feature* algunos de los atributos que se proponen son: estado (propuesta, aprobada o incorporada), beneficio (crítica, importante o útil), esfuerzo estimado, riesgo y estabilidad (para estos últimos atributos se suelen usar valores tales como: alto, medio o bajo).

Referencias

[1] G. Antoniol, B. Caprile, A. Potrich and P. Tonella. Design-code traceability recovery: selecting the basic linkage properties. Science of Computer Programming, Vol. 40 (2-3) pp. 213-234, Elsevier Science, 2001.

[2] K. Beck. Extreme Programming Explained. Addison-Wesley, 2000.

[3] J.-P. Corriveau. Traceability Process for Large OO Projects. IEEE Computer, pp. 63-68, September 1996.

[4] F. Cerbah and J. Euzenat. Traceability between models and text through terminology. Data and Knowledge Engineering, Vol. 38 (1) (2001) pp. 31-43, Elsevier Science, 2001.

[5] A. Egyed. A Scenario-Driven Approach to Traceability. In Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), pp- 123-132, 2001.

[6] R. Dömges and K. Pohl. Adapting Traceability Environments to Project-Specific Needs. Communications of ACM, Vol. 41, No 21, December 1998.

[7] O. Gotel and A. Finkelstein. Extended Requirements Traceability: Results of an Industrial Case Study. I'n Proceedings of 3rd International Symposium on Requirements Engineering (RE'97), IEEE Computer Society Press, pp. 169-178, 1997.

[8] P. Haumer, M. Jarke, K. Pohl and K. Weidenhaupt. Improving reviews of conceptual models by extended traceability to captured system usage. Interacting with Computers, Elsevier Science, 13 (1) pp. 77-95, 2000.
ftp://sunsite.informatik.rwth-aachen.de/pub/CREWS/CREWS-99-16.ps.gz

[9] I. Jacobson, G. Booch and J. Rumbaugh. The Unified Software Development Process. Addison-Wesley, 1999.

[10] M. Jarke. Requirements Tracing. Communications of the ACM, Vol. 41, No. 12, pp. 32-36, December 1998.

[11] J.C. Leite and A.Oliveira. A Client Oriented Requirements Baseline. In Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE'95), pp. 108-115, IEEE Computer Society Press, 1995

[12] J.C. Leite, G. Rossi, F. Balaguer, V. Maiorana, G. Kaplan, G. Hadad, A. Oliveros. Enhancing a Requirements Baseline with Scenarios. Requirements Engineering Vol. 2, No. 4, pp. 184-198, 1997.

[13] B. Ramesh. Factors influencing requirements traceability practice. Communication of the ACM, Vol. 41, No. 12, pp. 37-44, December 1998.

[14] B. Ramesh and M. Jarke. Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering, Vol. 27, No. 1, pp.58-93, January 2001.

[15] OMG Unified Modeling Language Specification. UML 1.4 with Action Semantics, Final Adopted Specification, January 2002. http://www.omg.org

[16] F. Pinheiro and J. Goguen. An Object-Oriented Tool for Tracing Requirements. IEEE Software, pp. 52-64, March 1996.

[17] K. Pohl. PRO-ART: Enabling Requirements Pre-Traceability. In Proceedings of the 2th International Conference on Requirements Engineering (ICRE'96), pp. 76-44, 1996.

[18] J. Smith. A Comparison of RUP and XP. Rational Software White Paper, 2001. www.rational.com/products/whitepapers/423.jsp

[19] R. Smith. Why UML Methodologists Are Throwing Their Weights Around. www.devx.com/uml/articles/Smith03/Smith03-1.asp

[20] I. Spence and L. Probasco. Traceability Studies for Managing Requirements with Use Cases. Rational Software White Paper, 1998. www.rational.com/products/whitepapers/022701.jsp

[21] M. Toranzo and J. Castro. A Comprehensive Traceability Model to Support the Design of Interactive Systems. ECOOP Workshops 1999, pp. 283-284, Lecture Notes in Computer Science 1743, Springer, 1999.

On the implication of application's requirements changes in the persistence layer: an automatic approach

Jennifer Pérez, José Ángel Carsí, Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n
E-46071 Valencia – Spain
{jeperez | pcarsi | iramos}@dsic.upv.es

Abstract. This paper presents a solution to the data evolution problem of information systems. This solution follows an automatic approach that reduces the number of people and the time invested in the software maintenance process. The information captured by the UML-like OASIS OO conceptual schemas, representing system evolution, is used in order to generate automatically a data migration plan. This paper is focused in the process of automatic generation of the structure and the content of this migration plan using patterns. A real migration example is used to explain the designed patterns. The data migration plan execution evolves and migrates the data from the initial conceptual schema database to the evolved conceptual schema database. The solution has been implemented in the migration tool ADAM (Automatic DAta Migration)[1].

Key words: Data migration plan, migration order, correspondences, migration patterns and migration expressions patterns

1. Introduction

Information systems have a dynamic nature and they frequently undergo changes. These changes are due to volatile business rules or to external factors. For this reason, system management tools must be able to deal with them in an efficient and complete way.

Existing CASE tools are able to generate applications following the paradigm of automatic prototyping. These CASE tools are called model compilers. They generate automatically the application code and the database schema from the conceptual schema of an information system. The automatic generation can be complete as Oblog Case ([Ser94]), Sosy Modeler® (OO-Method/CASE ([Pas97]), or partial, as Rational Rose ([Rat02]), System Architect ([Sys02]), Together ([Tog02]) and others. Moreover, they can evolve the application modifying the conceptual schema and regenerating the code and the database schema from the modified conceptual schema (see figure 1).

These model compilers do not consider the data stored in the database during the evolution process. When an information system undergoes an evolution, its conceptual schema is updated and a new schema results. From the new schema, a model compiler generates a new code and a new empty database. The structure and the properties of the new database could be different from the old

[1] ADAM is a CONSOFT technology product.

Pérez, J., Carsí, J. A., Ramos, I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR) at ICSM'02. 2002.

database. Therefore, the data remains compliant to the old database schema and the designer of the application must preserve the data of his company in spite of the evolution of the information system conceptual model. For this reason, the data remaining in the old database must be correctly migrated to the new one in order to satisfy the properties of the new database.

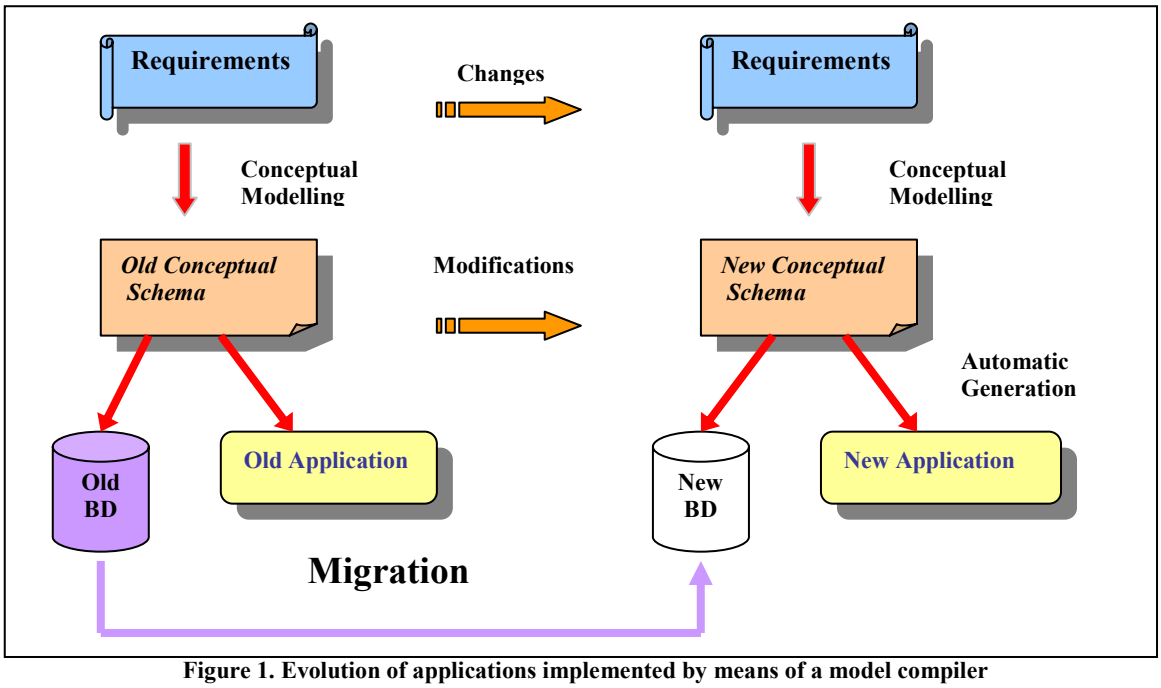


Figure 1. Evolution of applications implemented by means of a model compiler

The migration task is necessary and normally is handmade. This task increases considerably the maintenance cost of a software product. For this reason, our work is focused on data evolution in order to improve this database maintenance process. In this paper, we present a solution for the data evolution problem following an automatic approach and a migration tool (ADAM) has been implemented. Nowadays, ADAM is used together with Sosy Modeler®, a model compiler generating automatically three layer architecture (presentation, business logic and persistence) applications. A formal language OASIS ([Pas95]) based in dynamic logic is used as the specification language of UML-like models allowing its automatic compilation.

ADAM is a migration tool developed in the Department of Information Systems and Computation of the Valencia University of Technology in collaboration with the industrial partner CONSOFT. The data migration process followed by ADAM transfers and updates information system data from the old database to the new one. It reduces the time invested and the number of people involved in data evolution process up to an 80%. This maintenance improvement is due to the automatic tasks that constitute the three steps of the migration process. Despite the fact that these steps are performed automatically, its results can be modified by the analyst in a free way. In this last case, the process will be semi-automatic. The three steps involved are the following:

1.- Old and new conceptual schemas are compared using an algorithm and several comparison criteria ([Sil02]) to automatically obtain the matching (correspondences) between them.

2.- Changes on the data are obtained from these matching and are automatically translated to migration expressions using an object oriented migration language. These expressions constitute a OO data migration plan ([Per01-b]). The migration plan is generated as XML document, used by the third step to extract easily the migration information.

3.- The object oriented data migration plan is translated to a relational one. This translation process generates *Data Transformation Service* packets ([Cha00]) whose execution migrates automatically data from the old database to the new database ([Ana01]). These packets are executed by *SQL Server* ([SQL]).

The structure of the paper is the following: first, we show a real migration example that we use to explain the automatic generation of a data migration plan. Section 3 presents the generation process, the information that it needs, the patterns used to generate a data migration plan and the applied methodology. In section 4, a brief summary of related work is given. Finally, the future work and conclusions are presented in section 5.

2. A Real Example of Migration

In the paper, the well known database example of bills-customers-products presented in Figures 2 and 3 is used. We take a part of the object model of the complete example to understand the reasons why the *Product* class of the example has been evolved. The migration plan generation is exemplified in the paper using the *Product* class evolution. The OASIS schemas capture the structure of the billing management of an information system. The new conceptual schema (figure 3) improves the previous one (figure 2). This improvement consists in:

- Partition of the initial customers. As a result, the *Customer*, the *Person* and the *Company* classes of the new conceptual schema will extract data from the *Customer* class of the old conceptual schema.
- A finer specification of the *Bill* class. As a result, the *Bill* and the *Bill_row* classes of the new conceptual schema will extract their data from the *Bill* class of the old conceptual schema.
- Currency change. As a result, all the prices of the products must be converted into euros.
- Some model errors corrections.

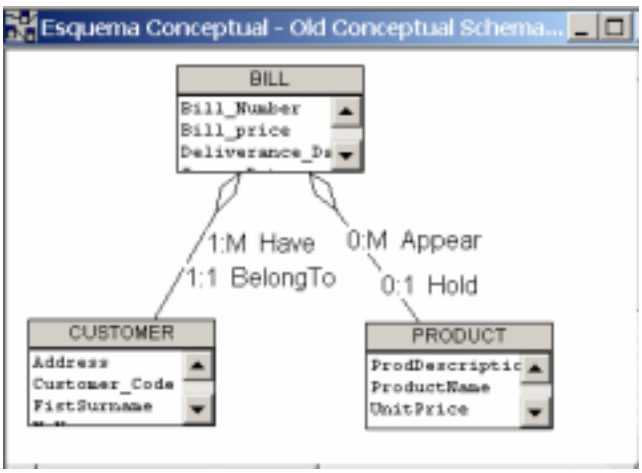


Figure 2. Old conceptual Schema

As a technical detail, it is worth noting that these schemas are modeled using OASIS. However, if we model them using UML, their appearance would be different. This is due to the fact that in OASIS the association is considered as a kind of aggregation and thus, the referential aggregation and the inclusive aggregation are represented by a lozenge. This is the reason why UML associations are modeled like OASIS referential aggregations. For example: The relationship between the classes *Bill_Row* and *Product* is an association represented by an OASIS referential aggregation.

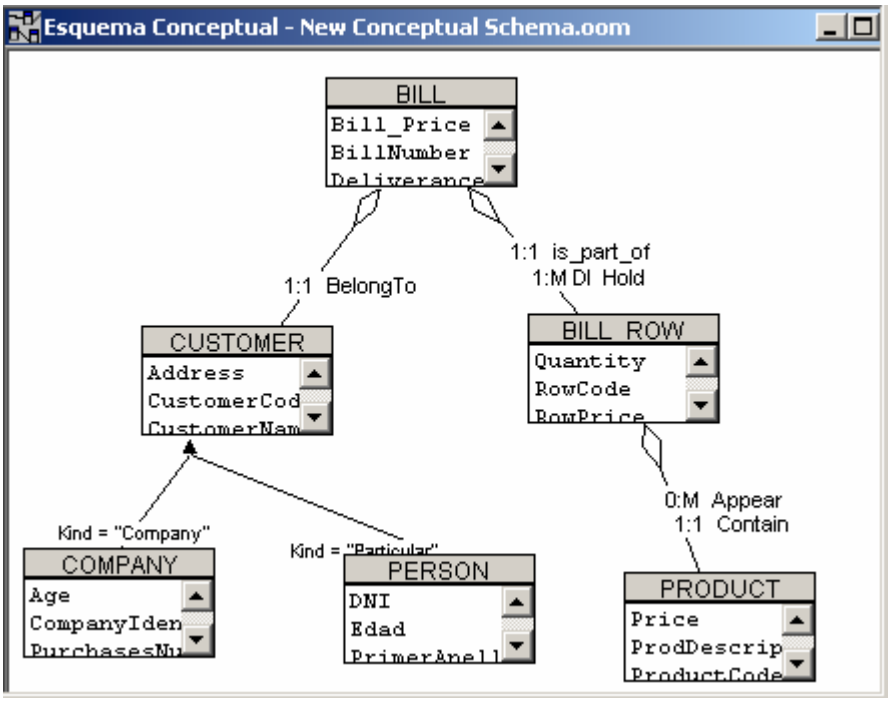


Figure 3. Old conceptual Schema

3. Automatic Generation of a Data Migration Plan

A data migration plan is a set of data changes that are specified using a migration language (ADML). Its execution allows us to transfer data from an old database to a new database. A data migration plan must include all the necessary changes in order to perform a correct migration and in the right order. It is structured as the composition of the following different elements: migration expressions, changes and modules. The highest abstraction level of this composition is constituted by migration modules whereas migration expressions are the lowest level. The definition of these elements is as follows:

- *Migration expressions:* Migration expressions are those expressions that can be specified in a data migration plan. Each type of migration expression has different semantic and follows different syntactic restrictions. Examples of migration expressions are: Data Source, Transformation Function, Filters, etc

- *Changes:* A change is the set of migration expressions that specify the updates undergone and the filters applied on the old database's instances.

- *Migration modules:* A migration module has a transactional behavior when it is executed by the ADAM tool. The composition of modules forms a data migration plan. Finally, it is important to note that there is one migration module for each class, aggregation and specialization of the new conceptual schema.

Each one of the changes that can be undergone in the old database are specified using a declarative language (a language which express properties of WHAT to do and not HOW to do it). This migration language (ADML) follows the object-oriented metaphor. The main advantage of ADML is the independence from any DBMS due to its high abstraction level.

This language makes use of path expressions notation to specify changes that have been undergone in a conceptual schema element[2]. Conceptual schema elements are a set of origins of the data, transformations of the data and filters that will be applied on the data at the execution moment.

3.1. Inputs

The generation process of a migration plan needs several data sources to generate automatically the structure and the contents of the plan. The inputs of this process are the following:

- *Matchings between conceptual schemas:* A matching defines the old element from which must be obtained the data to provide to a determinate new element. This information is available because the previous step of ADAM finds the matches between elements of both conceptual schemas. This step executes an algorithm that uses different comparison criteria in order to find the most suitable matches ([Sil01]). As a result, this process produces the correspondences between the elements of the conceptual schemas that have been compared. For example: The *Product* class of the new conceptual schema obtains its data from the *Product* class of the old conceptual schema (matches at class level) and the matches between their attributes are the following:

OLD CONCEPTUAL SCHEMA	NEW CONCEPTUAL SCHEMA
Cod_Product	ProductCode
ProductName	ProductName
ProdDescription	ProdDescription
UnitPrice	Price

Table.1. Matches between the attributes of the old Product class and the new Product class

- *Properties of the conceptual schema elements:* The properties of the conceptual schema elements are necessary in order to know the changes between the elements of a match instance and to generate the implied transformations. These transformations must be applied on the data of the old element in order to be compliant with the new element. This information is afforded by the conceptual schemas where only the properties of each element are defined. In our example we are going to focus on the following correspondence:

[2] The conceptual schema elements that have been considered for the generation of a data migration plan are: classes, attributes, aggregations, associations and specializations.

- The *Price* attribute of the *Product* class of the new conceptual schema obtains its data from the *UnitPrice* attribute of the *Product* class of the old conceptual schema. The properties of both attributes are included in the specification of the classes (see figures 4 and 5).

Atributos

Nombre	Tipo atributo	Tipo dato	Id	Tamaño	Valor defecto	Pedir al crear	Nulos
Cod_Product	Constante	Autonumerico	0			No	No
ProductName	Variable	String		15		Sí	Sí
ProdDescription	Variable	String		100		Sí	Sí
UnitPrice	Variable	Nat				Sí	Sí

Figure 4. Attributes of the Product class of the old conceptual schema

Atributos

Nombre	Tipo atributo	Tipo dato	Id	Tamaño	Valor defecto	Pedir al crear	Nulos
ProductName	Variable	String		15		Sí	Sí
ProdDescription	Variable	String		80		Sí	Sí
Price	Variable	Real				Sí	No
ProductCode	Constante	Autonumerico	0			No	No

Figure 5. Attributes of the Product class of the new conceptual schema

We can see at the figures 4 and 5 that the prices of the products were in pesetas, and now they must be in Euros. As a result, the data type of the *Price* attribute was *natural* in the old schema and is *real* in the new schema and also the *Price* value must be converted to the equivalent in euros. In addition, the attribute *UnitPrice* allowed null values and now the attribute *Price* does not allow null values.

- *Migration order:* Migration order is the order in which the data should be migrated. This preserves the database not to be in inconsistent states during the migration process. Moreover, this order affords the migration modules composition order and also the migration order between non-related modules. The computation of this order is made by means of an algorithm. The migration order algorithm analyses the structure of the new conceptual schema in order to obtain the relationships between elements. These relationships imply dependences determining the order to be followed in the data migration process ([Aba01]). The migration order obtained by the algorithm for the new conceptual schema of our example is the following: Customer, Bill, Product, Company, Person, Bill_Row.

3.2. Patterns

The automatic generation of a data migration plan implies generating its structure and its contents. First of all, ADAM generates the structure creating an empty migration module for each element of the new conceptual schema and includes the modules in the data migration plan following the computed migration order. Next, the module's content is automatically generated giving the migration expressions of each migration module and including them into their modules. Finally, a complete migration plan results. This automatic and complete generation of the migration plan is performed using two types of patterns: migration patterns and migration expressions patterns. We have specified them using the patterns design criteria proposed by [Ale79] and [Gam94]. They are available

in a patterns catalog ([Per01-a]), where they can be identified by a pattern number (**P-number**) and a title. Each pattern is composed of several sections that explain different qualities of the pattern. In the example patterns (see tables 2 and 3), the solution and the example sections are presented.

- Migration expressions patterns

There is a pattern for each one of the element's properties that can be changed by the schema evolution process and for any of the possible combinations of them. Each pattern produces a migration expression or a set of migration expressions specifying the correct transformation of data.

The generation of the migration expressions for a new element consists of determining which old element is related to it in the mapping and consulting their different properties. Next, apply the specific element pattern that specifies the migration expression code for the updated properties is instantiated and the resulting migration expressions are generated. Finally, these expressions are included in the new element module.

When the data migration plan is executed, the generated migration expressions of an element will be evaluated and the instances migrated to the new database. An example of a migration expression pattern is the one for an *attribute* when the "*name*", the "*data type*" and the "*not null value*" properties change (**P-12).**

P-12: Pattern for an attribute when the "name", the "data type" and the "not null value" properties change.
Solution
The solution presents the generic migration expressions that specify the attribute changes of "name", "data type" and "not null value" properties. In this case, as in the **P-04**[3] and **P-08**[4] patterns, it is necessary to perform a type conversion in the transformation function as follows: old_data_typeTOnew_data_type (old_attribute) This pattern is a composition of the "name", the "data type" and the"not null value" property patterns (**P-03**[5]**, P-04** and **P-06**[6] or **P-08** and **P-10**[7] patterns). The migrations expressions that express these changes are the following: Filter: IDENT_clase '.' IDENT_attr <> NULL Transformation_Function: generic_func '(' IDENT_clase '.' <IDENT_attr> ')'
Example
The prices of the products were in pesetas, and now they must be in euros. As a result, the data type of the *UnitPrice* attribute was natural in the old schema and is real in the new schema and the *Price* value must be converted to €. In addition, *Number* allowed null values and now does not allow null values. 

[3] **P-04:** Pattern for an attribute when the "data type" property change.
[4] **P-08:** Pattern for an attribute when the "name" and the "data type" properties change.
[5] **P-03:** Pattern for an attribute when the "name" property change.
[6] **P-06:** Pattern for an attribute when the "not null value" property change.
[7] **P-10:** Pattern for an attribute when the "name" and the "not null value" properties change.

```
Text Format
   Filter: OldCS.Product.UnitPrice <> NULL
      Transformation_Function: NatToReal(OldCS.Product.UnitPrice)
XML Format
   <Filter> OldCS.Product.UnitPrice <> NULL </Filter>
      <Transformation_Function> NatToReal(OldCS.Product.UnitPrice)
      </Transformation_Function>
```

Table.2 Solution and Example sections of the pattern P-12: Transformation function for the change of name, data type and not null value properties

In our example, ADAM uses the necessary patterns for each one of the established correspondences between the attributes of the new *Product* class and the old *Product* class. Moreover, we need take into account that the transformation function generated by the pattern (NatToReal(OldCS.Product.UnitPrice)) must be modified by the user in order to add the currency conversion function[8]. As a result, the transformation function that will be included in the data migration plan is PtsToEuro(NatToReal(OldCS.Product.UnitPrice)).

- Migration patterns

Migration patterns establish the way of migrating data for each type of conceptual schema element together with the necessary actions to migrate each type of conceptual schema element and the allowed migration expressions for each one. We must take into account, at the design process of a migration pattern, the type of conceptual schema element that will use the pattern, because the transformations that can undergo each conceptual schema element are different.

Moreover, migration patterns specify the expression place inside the module for each type of conceptual schema element, in the internal structure of a module.

A type of a conceptual schema element can have different associate patterns because there are different properties that influence the migration process. For example, the migration of a specialization relationship is different if its condition is more restrictive or less restrictive than the previous one or different because we must apply different types of filters on the data and different migration expression in a different place.

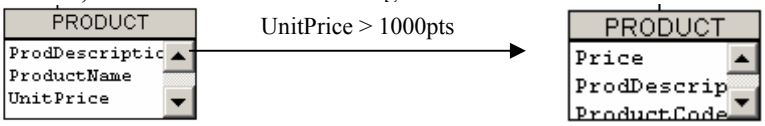
An example of a migration pattern is the pattern of the elemental class (**P-01)**:

P-01.Pattern: Elemental class
Solution
Let S be a set of schemas, C be an alphabet of classes, A be a set of attributes, G be a set of filters that are applied on old class population, GC be a set of conditions that are applied over old attributes, F be a set of transformation functions, SM be a set of matches between conceptual schemas, CM be a set of matches between classes of new and old conceptual schemas, and AM be a set of matches between attributes of conceptual schemas. $S1, S2 \in S \wedge$ $S1.C1, S2.C2 \in C$ $\wedge$ $S1.C1.a1, S2.C2.a2 \in A \wedge f_1, .., f_n \in F \wedge g_1,...,g_n \in G \wedge gc_1,...,gc_n \in GC \wedge SM1 \in SM \wedge CM1 \in CM \wedge AM1 \in CA \wedge SM1.old=S1 \wedge SM1.new=S2 \wedge CM1.old=S1.C1 \wedge CM1.new=S2.C2 \wedge AM1.old=S1.C1.a1 \wedge AM1.new=S2.C2.a2 \rightarrow$ data (S2.C2) = $\{y \mid \exists x \in data(S1.C1) \wedge \forall i \vDash_x g_i \; i=1,..,n \wedge ((y.a2 = f_n of_{n-1}...of_1(x.a1) \vee y.a2 = cte) \wedge \forall i \vDash_{x\downarrow a1} gc_i)\}$

[8] The PtsToEuros conversion function is included in the ADAM's set of built-in transformation functions.

Example

The *Product* class of the new conceptual schema obtains its data from the *Product* class of the old conceptual schema. However, the analyst of this system is only interested in the products that have a price higher than 1000. Moreover, all its attributes must be migrated with its transformations and conditions.



Text Format:

S1, S2 ∈ S ∧ S1.Product, S2.Product ∈ C ∧ S1.Product.Cod_Product, S1.Product.ProductName, S1.Product.ProdDescription, S1.Product.UnitPrice, S2.Product.ProductCode, S2.Product.Price, S2.Product.ProductName, S2.Product.ProdDescription ∈ A ∧ NatTOReal, RightTrunc, PtsToEuro ∈ F ∧ {S1.Product.UnitPrice > 1000pts}, {S1.Product.UnitPrice <> NULL} ∈ G ∧ SM1 ∈ SM ∧ CM1 ∈ CM ∧ AM1, AM2, AM3.AM4 ∈ CA ∧ SM1.old=S1 ∧ SM1.new=S2 ∧ CM1.old=S1.Product ∧ CM1.new=S2.Product ∧ AM1.old=S1.Product.Cod_Product ∧ AM1.new=S2.Product.ProductCode ∧ AM2.old=S1.Product.ProductName ∧ AM2.new=S2.Product.ProductName ∧ AM3.old=S1.Product.ProdDescription ∧ AM3.new=S2.Product.ProdDescription ∧ AM4.old=S1.Product.UnitPrice ∧ AM1.new=S2.Product.Price → data (S2.Product) = { y | ∃x ∈ data(S1.Product) $\models_x$ (S1.Product.UnitPrice > 1000 AND S1.Product.ProductName <> NULL AND S1.Product.ProdDescription <> NULL AND S1.Product.UnitPrice <> NULL ∧ (y.ProductCode = x.Cod_Product) ∧ (y.ProductName = x.ProductName) ∧ (y.ProdDescription = RightTrunc(x.ProdDescription) ∧ y.Price = PtsToEuro(NatToReal(x.UnitPrice)) }

XML Format:

```
<New_Conceptual_Schema>
 <Class>
  <Name> Product </Name>
  <Origin>
   <Name> Product </Name>
   <Filtered>
    <Filter> OldCS.CS1.Product.UnitPrice > 1000 AND
          OldCS.Product.ProductName <> NULL AND
          OldCS.Product.ProdDescription <> NULL AND
          OldCS.Product.UnitPrice <> NULL
    </Filter
    <Attribute>
     <Name> ProductCode </Name>
     <OriginAttribute> OldCS.Product.Cod_Product </OriginAttribute>
     <Transformation_Function> OldCS.Product.Cod_Product
     </Transformation_Function>
    </Attribute>
    <Attribute>
     <Name> ProductName </Name>
     <OriginAttribute> OldCS.Product.ProductName </OriginAttribute>
     <Transformation_Function> OldCS.Product.ProductName
     </Transformation_Function>
    </Attribute>
    <Attribute>
     <Name> ProdDescription </Name>
     <OriginAttribute> OldCS.Product.ProdDescription </OriginAttribute>
     <Transformation_Function> RightTrunc(OldCS.Product.ProdDescription)
     </Transformation_Function>
    </Attribute>
```

```
      <Attribute>
        <Name> Price </Name>
        <OriginAttribute> OldCS.Product.UnitPrice</OriginAttribute>
        <Transformation_Function> PtsToEuro(NatToReal(OldCS.Product.UnitPrice))
        </Transformation_Function>
      </Attribute>
    </Filtered>
  </Origin>
</Class>
</New_Conceptual_Schema>
```

Table.3 Solution and Example sections of the pattern P-01: Elemental Class

4. Related work

Nowadays, data evolution support is given by DBMS: Relational DBMS such as ORACLE ([Ora02]) or SQL Server ([Sql02]) and object-oriented DBMS like O2 ([Ard02]), Poet ([Poe02]) or Versant ([Ver02]).

Existing works in the schema evolution field focuses in the consistence problems between modified schemas and the corresponding databases. A set of invariants were defined by [Kim89] and several DBMS uses the same or similar invariant set. Existing solutions vary from those that use schema evolution (OTGen [Ler90] or Cocoon [Tre92]) to those that use schema versioning or views (Avance [Bjø88], CLOSQL [Mon93] or Multiperspectives [Odb95]). Our work follows the schema evolution solution due to our interest in application dynamics in order to be compliant with new requirements.

Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done by means of SQL statements or user defined scripts which can be executed on the database. However, these tools do not provide automatic support for the generation of these statements and scripts as theADAM tool does. For this reason, DB administrators must write migration code by hand. ETL tools offer a friendly interface to migrate data from an old database to a new database. However, they do not solve two important problems at the software maintenance stage: the high number of people required for the migration process and the high temporal cost.

The closest point of view to our approach is the TESS tool presented by Barbara Staund Lerner in her paper [Sta00]. Both approaches have two things in common: the processes are automatic and the tools are based on schema evolution. TESS uses an intermediate language generated from the relational schema code. This is an important difference with our approach, because we deal with directly the OO conceptual schemas and we do not have to translate them to an intermediate language. The OO conceptual schemas give us a higher abstraction level and save us from the translation process. It also implies that we do not have to take implementation details into account and so our process is more agile and has a higher abstraction level than the TESS process.

5. Further work and conclusions

The paper presents a solution to the data evolution problem of information systems. It explains the importance of this evolution problem and the need to support data changes in information systems.

The presented solution uses conceptual schemas, which represent the system evolution, to generate a data migration plan. This data migration plan is able to evolve and migrate data between the

respective databases of the conceptual schemas. The automatic generation process gives us a first version of a data migration plan that can be modified by the analyst. The contents and the structure of this first version of a data migration plan are generated by means of a set of patterns.

This solution has been implemented in a migration tool whose name is ADAM. Nowadays, ADAM is being used together with Sosy Modeler®, a model compiler. ADAM has achieved to reduce the time invested in the maintenance stage and the number of employees involve in the data evolution task up to 80 % in real experiences.

The high abstraction level of the migration language ADML allows us to be independent from the underlying DBMS. In order to use a given DBMS, it is only necessary to construct its specific "driver". ADAM specifically is a compiler which generates DTS packets of SQL Server from migration plan .Using ADAM we do not care of implementation details at the comparison and plan generation processes. The final "driver" will do automatically the work.

Finally, it is important to note that it is possible to improve the automatic migration process. Therefore, there are some aspects that we must solve in the future: the migration language needs to be able to specify migration expressions that contain elements of the new conceptual schema and the migration process should take into account also the dynamics of the system (States Transition Diagrams (STD)), etc.

6. Bibliography

[Aba01] Abad, S, Carsí, J.A, Ramos, I, *How to obtain a data migration order for the elements of an OO conceptual schema*, Workshop on Evolution in the VI conference on Software Engineering and Databases , Almagro, Ciudad Real, Spain, November 2001 (in Spanish)

[Ana01] Anaya, V. *Generation of transformation modules to migrate data between databases using a data migration plan*, Msc Project, Computer Science Faculty, Valencia University of Technology, September 2001 (in Spanish)

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press. 1979.

[Ard] Ardent Software, *O2*, http://www.ardent.com/

[Bjø88] Bjørnerstedt A., Britts S., *Avance - An object management system*, Proc. of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA), San Diego, California, USA, pag. 206-221, September 1988.

[Cha00] Chaffin M., Knight B., Robinson T., Professional SQL Server 2000 DTS (Data Transformation Services), Wrox, 2000

[Gam94] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addyson-Wesley.1994.

[Kim89] Kim W. and Lochovsky, F.H. (eds.), *Object-Oriented Concepts, Databases and Applications*, ACM Press, Addison-Wesley, 1989.

[Ler90] Lerner B.S., Habermenn A.N., *Beyond Schema Evolution to DatabaseReorganization*, In Proc. Of the Joint Conference on Object-OrientedSystems, Languages and Applications (OOPSLA) and ECOOP, Ottawa, Canada,pag.67-76, Octubre 1990.

[Mon93] Monk, S., *A Model for Schema Evolution in Object-Oriented Database Systems*, Ph. D. Thesis, Computing Department, Lancaster University, Febrero 1993.

[Odb95] Odberg, E., *Multiperspectives: Object Evolution and Schema Modificacion Management for Object-Oriented Databases*, Ph.D. Thesis, Department of Computer Systems and Telematics, Norwegian Institute of Technology, Febrero 1995.

[Ora] Oracle Corporation, *Oracle*, http://www.oracle.com

[Pas95] Pastor O. Et al., *OASIS versión 2 (2.2): A Class-Definition Language to Model Information Systems using an Object-Oriented Approach*, SPUPV-95.788, Universidad Politécnica de Valencia, España, 1995.

[Pas97] Pastor O. Et al, *OO-METHOD: A Software Production Environment Combining Conventional and Formal Methods*, Procc. of 9th International Conference, CaiSE97, Barcelona, 1997.

[Per01-a] Pérez, J. *Generation of a data migration plan for OASIS conceptual schemas*, Msc Project, Computer Science Faculty, Valencia University of Technology, September 2001 (in Spanish)

[Per01-b] Pérez J., Carsí J.A, Ramos I., Anaya V., Silva J.F., D*ata migration of conceptual schemas*, Workshop on Evolution in the VI conference on Software Engineering and Databases , Almagro, Ciudad Real, Spain, November 2001 (in Spanish)

[Poe] POET Software, *POET*, http://www.poet.com

[Rat] Rational Software, *Rational Rose*, http://www.rational.com/products/rose/

[Ser94] Sernadas A., Costa J.F., Sernadas C., "Object Specifications Through Diagrams: OBLOG Approach" INESC Lisbon 1994

[Sil02] Silva, J.F.., Carsí, J.A., Ramos, I., *Theoric analyze of the criteria of OO conceptual schemas comparison,* Ingeniería Informática Magazine, ISSN:0717-4197, Enero, http://www.inf.udec.cl/revista/edicion7/jsilva.htm

[SQL] Microsoft, *SQL Server*, http://www.microsoft.com/sql

[Sys] System Architec, http://www.popkin.com/products/sa2001/systemarchitect.htm

[Sta00] Staund Lerner, B., *A Model for Compound Type Changes Encountered in Schema Evolution*, ACM Transactions on Database Systems (TODS) Marzo 2000, Volumen 25 número 1

[Tre92] Tresch M., Scholl M.H., *Meta object Management and its applications to database evolution*, In Proc. Of the 11th International Conference on the Entity-Relationship Approach, Karlsruhe, Germany, pag.299-321, Octubre 1992

[Tog] TogetherSoft Corporation http://www.togethersoft.com/

[Ver] Versant Object Technology, *Versant*, http://www.versant.com/



ISIDRO RAMOS

PUBLICACIONES 1968-2013

VOLUMEN 3:
ARQUITECTURAS SOFTWARE (2002-2010)

VALENCIA, SEPTIEMBRE DE 2014

ÍNDICE GENERAL

VOLUMEN 1: MADRID, BILBAO, VALENCIA

Prólogo i

Días de hemeroteca v

… y noches de ordenador ix

Estructura del libro xi

El saber y el saber hacer en informática: una perspectiva histórica xiii

Índice general xxi

PARTE I: MADRID (1968-1974)

Prólogo de Roberto Moya Quiles 3

Isidro Ramos y el Centro de Cálculo de la Universidad de Madrid, por Ernesto García Camarero ex-director del CCUM 5

Seminarios del Centro De Cálculo De la Universidad Complutense de Madrid. 1968. 11

Gil Gayarre, M., Ramos, I., et al., Posibilidades de la Automatización en la corrección de exámenes. Revista de Automática. Año III, nº 7. 1970. 109

Fernández Flores, I. Ramos I., Traductor 70/13. Boletín CCUM. Volumen 10. 1970. 123

Moya R., Ramos, I. Aplicación de Hash-Coding al tratamiento automático de Gramáticas". Boletín CCUM. nº 17. 1971 137

Fernández Flórez, I., García Camarero, E., Ramos, I. A Basic Language Oriented to secondary School. Information Processing 71. 1972. 147

García Camarero, E., Ramos, I. Ensayo sobre la definición automática de tratamientos clínicos. Boletín CCUM. 1971. 151

Ramos, I. Lenguajes para la escritura de traductores. Revista de Automática. Enero-Marzo 1973. 163

Alfonseca, M., Ramos I. Un analizador sintáctico para un subconjunto de PL/I. Actas del Congreso Nacional de Automática. 1972. 169

Ramos, I. Un ejemplo de recursividad: HANOI. Boletín CCUM. nº 24. 1974. 205

Ramos, I., Lozano, C. Muñoz, A. Sistema de Creación y Mantenimiento de Bancos de Información. Revista de Automática, 1974 (Parte I), 1975 (Partes II y III). 211

PARTE II: BILBAO (1975-1983)

Prólogo de José Ramón Gariagoitia 255

Ramos, I. Système pour la création et gestion des banques de données. Seminaires Institut de Programmation (Université de Paris VI). 1974. 257

Moya R., Ramos I. Metaprogramación Estructurada. 3er Congreso Nacional de Informática y Automática. 1975. 263

Ramos, I., Moya, R., Martín, P. Evaluador Sintáctico del Lenguaje de Control de Comandos del Sistema Operativo EXEC-8. 3er Congreso Nacional de Informática y Automática. 1975. 289

Ramos, I. Aramberri, J. Cross-Macroassembler for Micro MCS-80 in a HP 2116-B Minicomputer. Euromicro. 1976. 301

Ramos, I. Informátique de Base: These, Antithèse, Synthèse. Convention Informatique. 1976. 309

Garitagoitia, J.R., Ramos, I. Relocating Direct Linking Loader for MCS-8 Microcomputer Series. Proc. of COMPCON 77. 1977. 313

Ramos, I. Sistema Editor / Perforador de textos. Revista de Automática e Informática. Volumen: nº 31. 1977. 319

Moya, R. Ramos, I. LPE: Un lenguaje en español para la programación estructurada. Proceso de Datos, no. 70, 1977. 325

Moya, R., Ramos I. SPL (Structured Programming Language). A language for structured programming. Proc. Of the 5th Annual III Conference. 1977. 331

Ramos, I. Aproximación prospectiva al software de base. Actas de la Convención Informática Latina. 1977. 339

Ramos, I., Aramberri, J. y Mellado, M. Modelo de realización de primitivas de sincronización para sistemas multimicroprocesador. Actas del 75 Aniversario Real Sociedad Española de Física y Química. 1978. 345

Aramberri, J., Ramos, I., Garitagoitia, J. R. Diseño y desarrollo de un sistema microprocesador. Actas I Jornadas sobre microprocesadores. 1978. 347

Moya, R., Ramos, I. SP Gramars (Context Free Gramars for Structured Programming). ACM SIGPLAN. Vol.4 Nº1. 1979. 357

Aramberri, J, Ramos, I., Mellado, M., Garitagoitia, J. R. Modelo de Sistema Microprocesador. Actas del 4º Congreso de Informática y Automática. 1979. 365

Sopeña, L., Ramos, I. Sistema de control en tiempo real de una red viaria. Actas del 4º Congreso de Informática y Automática. 1979. 375

Aguirre, J., Ramos, I. Sistema de programación estructurada con vuelta atrás. Actas del 4º Congreso de Informática y Automática, Madrid, 1979. 385

Ramos, I. Garitagoitia, J.R., Aramberri, J. Algunas soluciones al problema de interbloqueo entre procesos. 4º Congreso de Informática y Automática. 1979. 391

Guaresti, J.M., Garitagoitia, J.R. y Ramos I.A software chronological state controller. Actas del 4º Congreso de Informática y Automática. 1979. 403

Ramos, I. Calculadora de mesa. Un proyecto de clase sobre teoría de Lenguajes y Autómatas. Revista de Informática y Automática. 1979. 415

Zuloaga, J.L., Santamaria, R., Emaldi, R., Ramos, I. Sistema de Control lógico programable sobre microordenador. Revista de Informática y Automática. 1979. 421

Garitagoitia, J.R., Martinez,C., Ramos, I. MIKBUS: un bus para placas de microcomputadores cooperantes. Actas del II Simposio Español de Informática Repartida. 1982. 429

Garitagoitia, J.R., Ramos, I. Algoritmo de prevención para un sistema de gestión de ficheros. Actas de la Convención Informática Latina. 1983. 439

PARTE III: UNIVERSITAT DE VALÈNCIA - UNIVERSIDAD DE CASTILLA-LA MANCHA (1978-1987)

Prólogo de Francisco José Quiles Flor 455

Ramos, I. Semántica para SPL. Questió. Vol. 4. nº 1.1980. 459

Ramos, I. Un lenguaje de Programación con tipos Abstractos interpretado en PROLOG. Otra forma de compilar. Revista de Informática y Automática. 1984. 467

Ramos, I. Especificación de Pantallas. Revista de Informática y Automática, 1984. 471

Marín, J, Ramos, I. Metodología para la adquisición de material informático. Actas de las Jornadas de Informática en la Administración Local. 1984. 475

Ramos, I. The impact of new tecnologies on University organization: the example of Castilla-La Mancha. C.R.E. INFORMATION. Volumen: nº 70. 1985. 491

VOLUMEN 2: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

PARTE IV: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

Prólogo de María Alpuente 527

Celma, M., Moreno, L., Pastor, O., Ramos, I. Implementación de un sistema de ges-

tión de bases de datos deductivas. Revista de Informática y Automática, Vol. XX, Nº 3. 1987. 529

Alpuente, M., Botti, V., Galiano, I., Sanchis, E., Vila, F., Ramos, I. Entorno de programación LALI implementado en PROLOG. Revista de Informática y Automática . Vol. XX, Nº 4. 1987. 537

González del Río, A., Alpuente, M., Casamayor, J.C., Pastor, M.A., Ramírez, M.J., Ramos, I. Protolog : A conceptual Schema Facility for Automated Prototype Generation. 7th. IASTED International Symposium. 1989. 543

Ramos I., Canós, J.H., Forradellas, R., Oliver, J. A conceptual schema specification system for rapid prototyping. 8th IASTED International Symposium. Austria. Feb. 1990. 547

Ramos, I. Logics and OO-Data Bases: a declarative approach. Proc. International Conference on Database and Expert Systems Applications (DEXA 90), 1990. 551

Ramos, I.; Pastor, O.; Casado, V. OO and Active Formal Information System Specification. Proc. International Conference on Database and Expert System Applications (DEXA 91) Vol. II. 1991. 557

Pastor, O. Ramos, I. Canós, J.H. Object Oriented and Relational Specification of Information Systems. Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS. 1991. 563

Ramos, I., Cuevas, J., Devesa, J., Pastor, O., Canós, J.H. On the use of Algebras as Semantic domains of Object Societies. Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS. 1992. 579

Alpuente, M., Ramírez, M. J. An Equational Constraint Logic approach to Conceptual Modelling. In Ramos, I. and Tjoa, A. (eds), Proc. of the International Conference on Database and Expert Systems Applications (DEXA 92), 1992. 589

Ramos, I., Pastor, O., Cuevas J., Devesa , J. Objects as Observable Processes. Proc. of the 4th International Workshop of the Deductive Approach for DB and IS.1993. 595

Pastor, O., Ramos, I., Canós, J.H. From Analysis to Design in an OO and Deductive Environment. Proc of the International Conference on Database and Expert System Applications (DEXA´93). 1993. 611

Toval, A., Ramos, I. , Pastor, O. Prototyping Object Oriented Specifications in an Algebraic Environment. Proc. Of the International Conference on Database and Expert System Applications (DEXA'94), 1994. 617

Devesa, J., Cuevas, J., Ramos, I. Implementation of a Term Rewriting System for solving process queries in an object-oriented environment". Proc. Of the GULP-PRODE 94. 1994. 627

Canós, J. H., Penadés, M.C., Ramos, I., Pastor, O. A Knowledge-Base Architecture for Object Societies. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 643

Pastor, O., Ramos, I., Canós J.H. OASIS v2: A Class Definition Language. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 649

Ramos, I., Toval, A., García-Molina, J., Flores, M., González, M., Fernández, J. L., Nicolás, J., Pastor, O., Canós, J. H., Pelechano, V., Penadés, M.C., Bonet, B. Entorno General de Produccion automática de prototipos de software orientado a objetos: OOAP (Object-Oriented Assistant Prototyper). Actas de las I Jornadas de Informática. Asociación Española de Informática y Automática (AEIA). 1995. 659

Pastor O., Pelechano V., Bonet B., Ramos I. An object Oriented Approach for making Automated Prototyping Feasible. Proc. of the International Conference on Database and Expert System Applications (DEXA 96). 1996. 673

Carsí, J.A., Ramos, I., Penadés, M.C., Pelechano, V., Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic. Actas de las I Jornadas de Trabajo en Ingeniería del Software, 1996. 683

Penadés, M.C., Ramos, I., Canós, J.H., Tolsá, J., Ferrer, J. Análisis, diseño e implementación de un Entorno de Prototipación Automática. Actas de las I Jornadas de Trabajo en Ingeniería del Software,1996. 695

Sánchez, P. Letelier, P., Ramos, I. Representación de Objetos OASIS como Teorias y Traducción a un Modelo de Concurrencia. Computación y Sistemas, Vol. I, Nº. 2. 1997. 707

Lozano, M.D., Ramos, I. Evolución y perspectivas en el desarrollo de software: nuevas tendencias orientadas a objetos. Ensayos: revista de la Escuela Universitaria de Magisterio de Albacete, Nº. 12. 1997. 719

Sánchez, P., Letelier, P., Ramos, I. Constructs for Prototyping Information Systems using Object Petri Nets. Proc. of the IEEE International Conference on System, Man and Cybernetics.1997. 733

Lozano, M.D., Ramos, I., Cuenca, P. Application of an Object-Oriented Methodology for Automatic Software Production : OO-Method. Proc. Of the IEEE PACRIM 97. 1997. 739

Letelier, P., Sánchez, P., Ramos, I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitios. Proc. of the Argentine Symposium on Object Orientation. 1997. 743

Letelier, P. Sánchez, P., Ramos. I. Animation of Systems Specifications Using Concurrent Logic Programming. Proc. of the Symposium in Logical Approaches to Agent and Modelling and Design (ESSLLI´97). 1997. 751

Carsí, J.A., Canós, J.H., Ramos, I. Un intérprete de Transaction Frame Logic. Actas de las II Jornadas de Trabajo en Ingeniería del Software. 1997. 759

Tolsá, J.M., Ferrer, J., Penadés, M.C., Canós, J.H., Ramos, I. Meta-KAOS. Un metaentorno de prototipación automática. Actas de las II Jornadas de Trabajo en

Ingeniería del Software. 1997. 773

Ramos, I.,Canós, J.H., Pastor,O.+ investigadores proyecto MENHIR. MENHIR: Modelos, Entornos y Nuevas Herramientas para la Ingeniería de Requisitos. Actas de las III Jornadas sobre Tecnología de Objetos. 1997. 787

Canós, J.H., Lozano, M.D., Ramos, I. KAOS: An Object-Oriented Software Tool for the Objects Definition, Updating, Querying and Programming in an Object-Oriented Environment. Proc. of the IEEE Canadian Conference on Electrical & Computer Engineering.1998. 799

Letelier, P., Sánchez, P., Ramos, I. Especificaciones de proceso para objetos y su representación en Lógica Dinámica. Actas de las III Jornadas de Ingeniería de Software. 1998. 803

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto. Actas de las III Jornadas de Ingeniería del Software. 1998. 813

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a las pautas de representación de OASIS en un formalismo gráfico. Actas de las III Jornadas de Ingeniería del Software. 1998. 827

Carsí, J. A., Camilleri, S., Canós, J.H., Ramos, I. Propuesta de Interfaz Gráfica de Usuario homogénea para el diseño y la explotación de Sistemas de Información. Actas de las III Jornadas de Ingeniería del Software. 1998. 839

Letelier, P., Sánchez, P., Ramos, I. Un ambiente para especificación incremental y validación de modelos conceptuales. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99).1999. 853

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99). 1999. 867

Letelier, P., Sánchez, P., Ramos, I. Prototyping a requirements specification through an automatically generated concurrent logic program. Proc. of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99). 1999. 879

Pastor,O., Canós,J.H., Ramos,I. From CASE to CARE (Computer-Aided Requirements Engineering). Proc. of the International Conference on Conceptual Modeling (ER'99). 1999. 895

Lozano, M., Ramos, I. Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. Workshop on Integrating Human Factors into Use Cases and OO Methods at ECOOP'99. 1999. 911

Letelier, P., Sánchez, P., Ramos, I. Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales". Actas del VI Congreso Internacional de Investigación en Ciencias Computacionales (CIICC'99). 1999. 917

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. Modelado conceptual con un lenguaje

formal y orientado a objeto. Actas de la XXV Conferencia Latinoamericana de Informática (CLEI'99). 1999. 929

Letelier, P. Sánchez, P., Ramos, I. Animation of Conceptual Models using two Concurrent Environments: An overview. Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers. 1999. 943

Parets, J., Carsí, J., Canós, J.H., Anaya, A., Hurtado, M.V., Penadés, M.C., Paderewski, P., Ramos, I., Rodriguez, M.J., La evolución de modelos en el desarrollo de software: los enfoques MEDES y OASIS. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos. 1999. 951

Sánchez, P., Letelier, P., Ramos, I. Una Interpretación de la Herencia de OASIS en Lógica Dinámica. Computación y Sistemas, vol. 4, núm.4. 2000. 963

Sánchez, P., Letelier, P., Ramos, I. Animating formal specifications with Inheritance in a DL-based framework. Requirements Engineering Journal, vol.4, no.4. 2000. 973

Anaya, R., Ramos, I. Un lenguaje formal para especificación y refinamiento de Objetos de Negocio. Ingeniería del software y bases de datos: tendencias actuales. Universidad de Castilla-La Mancha, 2000. 995

Anaya, R., Ramos, I. AR2CA: Una Herramienta para la Construcción de componentes reutilizables a través de niveles de refinamiento. Actas de 3as Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software (IDEAS2000). 2000. 1011

Carsí, J. A., Ramos, I., Canós, J.H. Reflection in the OASIS object model to solve Schema Evolution Problems. Proc. of the International Symposium on the Principles of Software Evolution (ISPSE2000). 2000. 1025

Sánchez, P., Letelier, P., Ramos, I. Validation of Conceptual Models by Animation in an Scenario based approach. Workshop 'Scenario-based round-trip engineering' at OOPSLA 2000. 2000. 1031

Giret, A., Ramos, I.Full software agents as OASIS 3.0 objects. Proc. of the 2nd Constraint) Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2001. 2001. 1037

Letelier, P., Ramos, I. Trazabilidad de Requisitos y Calidad del Software: Una Aproximación basada en UML. En Piattini y García (Eds.), Calidad en el Desarrollo y Mantenimiento del Software. Ra-MA, 2002. 1053

Pérez, J., Carsí, J. A., Ramos, I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR) at ICSM'02. 2002. 1075

VOLUMEN 3: ARQUITECTURAS SOFTWARE (2002-2010)

PARTE V: ARQUITECTURAS SOFTWARE (2002-2010)

Prólogo de Patricio Letelier 1109

Ferrer, J., Lorenzo, M. A., Ramos, I.,Carsí, J. A., Pérez, J. Modeling Dynamic Aspects in Architectures and Multiagent Systems. Proc. of the Constraint Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2002. 2002. 1111

Pérez, J., Ramos, I., Lorenzo, M. A., Letelier, P., Jaén, J. PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos. Actas de las VII Jornadas de Ingeniería del Software y Bases de Datos. 2002. 1125

Ferrer, J., Lorenzo, M. A., Ramos, I., Carsí, J. A. Dinamismo en arquitecturas y sistemas multiagente. Actas de las V Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2002). 2002. 1137

Jaén, J., Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures. Proc. of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03). 2003. 1147

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. Proc. of the 3rd IEEE International Conference on Quality Software (QSIC 2003). 2003. 1153

Pérez, J., Ali, N., Ramos, I., Carsí, J. A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes. Workshop Desarrollo de Software Orientado a Aspectos (DSOA). Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1161

Pérez, J., Ali, N., Ramos, I., Pastor, J. A., Sánchez, P., Álvarez, B. Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1167

Lorenzo, M. A., Peñarrubia, A., Carsí, J.A., Ramos, I. Modelando con UML Arquitecturas Software Reflexivas desde la Aproximación Orientada a Aspectos: Un Caso de Estudio. Actas de las VI Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2003). 2003. 1177

Navarro, E., Ramos, I. Requirements and Architecture: a marriage for Quality Assurance. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1189

Silva, J., Ali, N., Carsí, J. A.,Ramos, I. El aspecto de distribución de PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1199

Ali, N., Silva, J., Jaén, J., Ramos, I., Carsí, J. A., Pérez, J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proc. of the 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2003). 2003. 1209

Pérez J., Ali, N. Carsí, J. A., Ramos, I. Navarro, E. Designing Software Architectures with an Aspect-Oriented Language. ADVICE: Journal of Aspect Orientation, Vol. 1, No. 1, 2004. 1217

Navarro, E., Letelier, P., Ramos, I. Goals and Quality Characteristics: Separating

Concerns. Proc. of Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004. 2004. 1239

Navarro, E., Letelier, P., Ramos, I. UML Visualization for an aspect and goal oriented approach. Proc. of the 5th Aspect-Oriented Modeling Workshop (AOM'04) at UML 2004 Conference. 2004. 1247

Navarro, E., Ramos, I., Pérez, J. Goals Model Driving Software Architecture. 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004). 2004. 1255

Ali, N., Pérez, J., Ramos, I. High Level Specification of Distributed and Mobile Information Systems. Proc. of the Second International Symposium on Innovation in Information & Communication Technology (ISSICT2004). 2004. 1263

Ali, N., Carsí, J. A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2004. 1277

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004). 2004. 1285

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Orientación a aspectos y Orientación a objetivos: una propuesta para su integración. Taller de Desarrollo de Software Orientado a Aspectos, junto a JISBD'2004. 2004. 1293

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Dynamic Evolution in Aspect-Oriented Architectural Models. European Workshop on Software Architecture. 2005. 1299

Pérez, J., Ali, N., Costa, C., Carsí, J. A., Ramos, I. Executing Aspect-Oriented Component-Based Software Architectures on . NET Technology. Proc. of the 3rd International Conference on .NET Technologies. 2005. 1317

Ali, N., Ramos, I., Carsí, J. A. A Conceptual Model for Distributed Aspect-Oriented Software Architectures. Proc. of the International Conference on Information Technology Coding and Computing, ITCC 2005. 2005. 1329

Limón, R., Torres, J., Ramos, I. Separación de "Crosscutting Concerns" en el Desarrollo Orientado a Aspectos a Nivel de Diseño Arquitectónico. Actas del 6º Simposium Iberoamericano de Computación e Informática (SICI'05). 2005. 1335

Navarro, E., Letelier, P., Ramos, I. Integrating Expressiveness of Modern Requirements Modeling Approaches. Proc. of the 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005). 2005. 1345

Costa, C., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1355

Cabedo, R., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Implementación Orientada a Aspectos en C# de un sistema robótico tele-operado. III Taller de Desarrollo de Software Orientado a Aspectos de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2005. 1363

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1371

Navarro, E., Letelier, P., Mocholí, J. A.,Ramos, I. A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems, XLVI(5), Special Issue on Systems Analysis and Design. 2006. 1379

Cabello, M. E., Costa, C., Ramos, I.,Carsí, J. A. Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System. Journal of the WSEAS Transactions on Information Science and Applications, Issue 10, Volume 3. 2006. 1391

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. Proc. of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006). 2006. 1399

Pérez, J., Navarro, E., Letelier, P., Ramos, I. A Modelling Proposal for Aspect-Oriented Software Architectures. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). 2006. 1415

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. A Goal-Oriented Approach for Safety Requirements Specification. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06). 2006. 1425

Limón, R., Ramos, I., Torres, J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development. Proc. of the International Conference the Computational Science and Its Applications (ICCSA 2006) . 2006. 1433

Limón, R., Ramos, I. Using Styles to Improve the Architectural Views Design. Proc. Of the International Conference on Software Engineering Advances (ICSEA 2006). 2006. 1443

Cabello, M. E., Costa, C., Ramos, I., Carsí, J. A. Generic Architecture of an Expert System for Diagnosis. Proc. of the 10th WSEAS International Conference on Computers. 2006. 1449

Cabello, M. E., Ramos, I., Costa, C. Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico. Actas del XIX Congreso Nacional y V Congreso Internacional de Informática y Computación del ANIEI-CIICC´07. 2006. 1455

Ali, N., Millan, C., Ramos, C. Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model. Proc. of the 8th International Sym-

posium on Distributed Objects and Applications (DOA). 2006. 1463

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes. IEEE Latin America Transactions, Special Edition JISBD'06, vol. 5, issue 4. 2007. 1481

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Mobile Ambients in Aspect-Oriented Software Architectures. Proc. of the IFIP Working Conference on Software Engineering Techniques - SET 2006. 2006. 1489

Navarro, E., Letelier, P., Reolid, D., Ramos, I. Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques. Proc. of the 15th International Conference on Information System Development. 2006. 1501

Navarro, E., Letelier, P., Ramos, I. Requirements and Scenarios: running Aspect Oriented Software Architectures. Proc. of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). 2007. 1515

Limón, R., Ramos, I. Analyzing Styles of the Modular Software Architecture View. Proc. of the First European Conference on Software Architecture (ECSA 2007). 2007. 1521

Limón, R., Ramos, I. Relating Software Architecture Views by Using MDA. Proc. of the ICCSA 2007 International Conference on Computational Science and its Applications. 2007. 1525

Limón, R., Ramos, I., Morales, M. Zárate, J. Analyzing and Designing Software Architecture Views driven by their Relationships. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 1537

Pérez, J., Costa, C., Carsí, J. A.,Ramos, I. Verificación de Modelos Arquitectónicos Orientados a Aspectos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos. 2007. 1547

Costa, C., Ali, N., Pérez, J., Carsí, J. A., Ramos, I. Dynamic Reconfiguration of Software Architectures Through Aspects. Proc. of the First European Conference on Software Architecture (ECSA'07). 2007. 1557

Limón, R., Cabello, M. E., Ramos, I. Establishing Relations among Software Architecture Views through MDA for SPL. Proc. of the 14th International Congress on Computer Science Research (CIICC'07). 2007. 1563

Pérez, J., Ali, N., Carsí, J. A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J. A. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Journal of Information and Software Technology 50 (2008) . 2008. 1575

Ali, N., Solís, C., Ramos, I. Comparing architecture description languages for mobile software systems. Proc. of the First International Workshop on Software Architectures and Mobility at ICSE'08. 2008. 1597

Navarro, E., Gómez, A., Letelier, P., Ramos, I. MORPHEUS: a supporting tool for MDD. Proc. of the18th International Conference on Information Systems

Development (ISD2009). 2009. 1603

Ali, N., Ramos, I., Solís, C. Ambient-PRISMA: Ambients in Mobile Aspect-Oriented Software Architecture. Journal of Systems and Software 83. 2010. 1615

Pérez, J., Ramos, I., Carsí, J.A., Costa, C.: Model-Driven Development of Aspect-Oriented Software Architectures. Journal of Universal Computer Science, vol. 19, no. 10. 2013. 1637

VOLUMEN 4: INTERFACES DE USUARIO (2000-2002) - GESTIÓN DE MODELOS (2002-2006)

PARTE VI: INTERFACES DE USUARIO - IDEAS (2000-2002)

Prólogo de María Dolores Lozano 1701

Lozano, M.D., Ramos, I. An Object Oriented Approach for the Specification of User Interfaces. Proceedings of PACRIM'99 Conference. 2000. 1703

Lozano, M. D., Ramos, I., González, P. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development". Proc. of the IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000). 2000. 1707

Lozano, M. D, González, P., Ramos, I. Models and Ontologies for the Specification of User Interfaces. Proc. of the Workshop on Model Engineering at ECOOP'00, 2000. 1717

Lozano, M. D., Ramos, I. Aproximación Basada en Modelos en la especificación de Interfaces de Usuario. Actas de las I Jornadas de Interacción Persona-Ordenador (INTERACCIÓN-20), 2000. 1725

Lozano, M. D., González, P., Ramos, I. User Interface Generation. Current Trends. INFORMATIK /INFORMATIQUE. Journal of the Swiss informatics societies. Vol. 2. 2001. 1733

Lozano, M. D., González, P., Ramos, I. IDEAS: An Object-Oriented Environment for developing User Interfaces". Proc. of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01). 2001. 1739

Lozano, M. D., González, P., Ramos, I., Montero, F., Molina, J. P. Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.16.2002. 1749

Lozano, M., González, P., Ramos, I. A Graphical User Interface Development Tool. 16th British Computer Society Conference on Human-Computer Interaction (BHCI'02). 1759

Lozano, M., González, P., Montero, F. Molina, J. P., Ramos, I. Integrating Usability within the User Interface Development Process of Web Applications. Proc. of the 2nd International Workshop on Web Oriented Software Technology at ECOOP'02. 2002. 1763

Montero, F. Lozano, M. D., González, P., Ramos, I. Designing Web Sites by Using Design Patterns. Proc. of SugarLoafPLoP 2002. 2002. 1779

PARTE VII: GESTIÓN DE MODELOS

Prólogo de José Ángel Carsí 1805

Carsí J.A, Ramos I., Silva J., Pérez J., Anaya V. Un Generador Automático de Planes de Migración de Datos. I+D Computación, Vol. 1, No. 1. 2002. 1809

Silva J.F., Carsí J.A., Ramos I. Comparación automática de esquemas conceptuales orientados a objeto, Revista Ingeniería Informática - Revista electrónica del Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción - Chile. 2002. 1821

Pérez J., Ramos I., Anaya V., Cubel J.M, Domínguez F., Boronat A., Carsí J.A. Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. Proc. of the Workshop on Software Evolution Through Transformations (SET'02) at ICGT2002. 2002. 1833

Silva, J., Carsí, J. A., Ramos, I. An algorithm to compare OO-Conceptual Schemas. Proc. of the International Conference on Software Maintenance (ICSM'2002). 2002. 1847

Pérez J., Carsí J.A, Ramos I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR'2002) at the ICSM'02. 2002. 1857

Pérez, J., Carsí, J. A., Ramos, I. ADML: A Language for Automatic Generation of Migration Plans. Proc. of the First Eurasian Conference on Advances in Information and Communication Technology. 2002. 1869

Boronat, A., Carsí, J. A., Ramos, I. Una plataforma semántica para la gestión de modelos. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos. 2003. 1879

Boronat, A., Pérez, J., Carsí, J. A.,Ramos, I. Two experiencies in software dynamics. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. 2004. 1889

Boronat, A., Carsí, J. A., Ramos, I. An Algebraic Baseline for Automatic Transformations in MDA. Proc. of the Workshop on Software Evolution Through Transformations at ICGT2004. 2004. 1919

Boronat, A., Pedrós, J., Carsí, J. A., Ramos, I. Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004). 2004. 1935

Boronat, A., Carsí, J. A., Ramos, I., Pedrós, J. An Approach for Cross-Model Semantic Transformation on the . NET Framework. Proc. of the .NET Technologies'2005 Conference. 2005. 1947

Boronat, A., Boronat, A., Carsí, J. A.,Ramos, I. Una Arquitectura para la Definición

de Metáforas Gráficas para Metamodelos. Revista IEEE América Latina, Special Edition - JISBD'2004. 2005. 1957

Silva, J., Carsí, J. A., Ramos, I. Análisis teórico experimental de criterios de comparación de esquemas conceptuales orientados a objeto. Computación y Sistemas, Vol. IX no.1. 2005. 1967

Boronat, A., Carsí, J. A., Ramos, I. Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. Proc. of the 9th European Conference on Software Maintenance and Reengineering. 2005. 1983

Boronat, A., Carsí, J. A., Ramos, I. Automatic Support for Traceability in a Generic Model Management Framework. Proc. of the European Conference on Model Driven Architecture - Foundations and Applications. 2005. 1995

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). 2005. 2011

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos - JISBD'05. 2005. 2019

Queralt, P., Hoyos, L., Boronat, A., Carsí, J. A., Ramos, I. Un motor de transformación de modelos con soporte para el lenguaje QVT Relations. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). 2006. 2025

Boronat, A., Carsí, J. A., Ramos, I. Exogenous Model Merging by means of Model Management Operators. Proc. of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change at ICGT 2006. 2006. 2035

Boronat, A., Carsí, J. A., Ramos, I.,Letelier, P. Formal Model Merging Applied to Class Diagram Integration. Proc. of the International ERCIM Workshop on Software Evolution (2006). 2006. 2055

Boronat, A., Carsí, J. A., Ramos, I. Algebraic Specification of a Model Transformation Engine. Proc. of the Fundamental Approaches to Software Engineering Workshop (FASE'06) at ETAPS'06. 2006. 2061

Boronat, A., Ramos, I., Carsí, J. A. Definition of OCL 2. 0 Operational Semantics by means of a Parameterized Algebraic Specification. Proc. of the 1st International Workshop on Algebraic Foundations for OCL and Applications. 2006. 2077

Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J. A. An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework. Proc. of the Second European Conference on Model Driven Architecture. 2006. 2093

Gómez, A., Boronat, A., Hoyos, L., Carsí, J. A., Ramos, I. Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos.

Actas de las XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006). 2006. 2109

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I., Täubner, C., Eckstein, S. Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2119

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2129

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2135

Lloret, E., Escobar, S., Palomar, M., Ramos, I. Incremental and adaptive Software Systems Development of Natural Language Applications. Proc. of the 22nd International Conference on Information Systems Development (ISD 2013). 2013. 2143

VOLUMEN 5: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

PARTE VIII: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

Prólogo de Emilio Insfrán y Silvia Abrahão 2177

Carsí, J. A., Insfrán, E., Abrahao, S., Genero, M., Piattini, M., Ramos, I. ¿Cómo utilizar experimentos controlados en la selección de transformaciones QVT para obtener diagramas de clases UML más fáciles de entender? Proc. of the 8th Argentine Symposium on Software Engineering (ASSE 2007). 2007. 2179

Limón, R. Ramos, I. Aragón, R. Transformación de Vistas Arquitectónicas Orientada por Modelos. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 2189

Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Marco de Trabajo basado en MDA para la Medición Genérica del Software. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2199

Mora, B., García, F., Francisco Ruíz, Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Software Measurement by Using QVT Transformations in an MDA Context (ICEIS 2008). Proc. of the Tenth International Conference on Enterprise Information Systems, Volume DISI. 2008. 2209

Abrahao, S., Genero, M., Insfrán, E., Carsí, J. A., Ramos, I., Piattini, M. Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In Rech, J. and Bunse, C. (Eds.), Model-Driven Software Development: Integrating Quality Assurance. 2008. 2217

Insfrán, E., Carsí, J. A., Abrahao, S., Genero, M., Piattini, M., Ramos, I. Towards Quality-Driven Model Transformations: A Replication Study. Proc. of the International Workshop on Empirical Studies of Model-Driven Engineering (ESMDE 2008) at MODELS 2008. 2008. 2247

Cabello, M. E., Ramos, I. A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines. Proc. of the International Conference of Health Informatics- HEALTHINF´08. 2008. 2257

Cabello, M. E., Ramos, I. Variability Management in Product Lines for Decision Support Systems. Proc. of the 10th International Conference on Enterprise Information Systems (ICEIS´08). 2008. 2267

Cabello, M. E., Ramos, I. The Baseline: the milestone of Software Product Lines for Expert Systems automatic development. Proc. of the 9th Mexican International Conference on Computer Science. 2008. 2277

Gómez, M., Gómez, A., Cabello, M. E.,Ramos, I. BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. Actas del VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). 2009. 2287

Cabello, M. E., Ramos, I., Gómez, A., Limón, R. Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for Expert Systems development. Proc. of the 1st Asia Conference on Intelligent Information and Database Systems (ACIIDS´09). 2009. 2297

Cabello, M. E., Ramos, I. Expert Systems development through Software Product Lines techniques. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2303

Cabello, M. E., Ramos, I. Model-Driven Development of Decision Support Systems: tackling the variability problem. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2315

Gómez, A., Ramos, I. Cardinality-based feature models and Model-Driven Engineering: fitting them together. Proc. of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. 2010. 2327

Gómez, A., Cabello, M. E., Ramos, I. BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. Proc. of the 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010) at SPLC 2010. 2010. 2335

Gómez, A., Ramos, I. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. Proc. of the 19th International Conference on Information Systems Development (ISD'10) 2010. 2343

Abrahao, S., Javier González, Insfrán, E.,Ramos, I. Software Evolution in Model-Driven Product Line Engineering. ERCIM News 88, Special Theme: Evolving Software. 2012. 2355

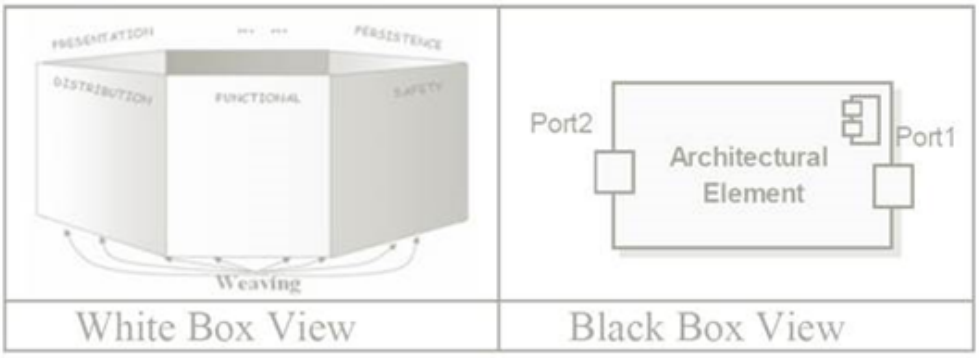
Cabello, M. E., Ramos, I., Gutiérrez, J. R., Gómez, A., Limón, R. SPL Variability Management, Cardinality and Types: an MDA Approach. International Journal of Intelligent Information and Database Systems (IJIIDS), Vol. 6, No. 2. 2012. 2357

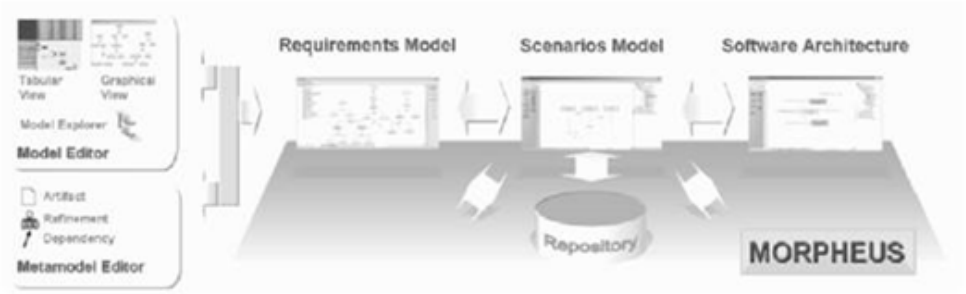
Insfrán, E., Abrahao, S., Javier González, McGregor, J. D., Ramos, I. A Multimodeling Approach for Quality-Driven Architecture Derivation. Proc. of the 21st International Conference on Information Systems Development (ISD2012). 2012. 2383

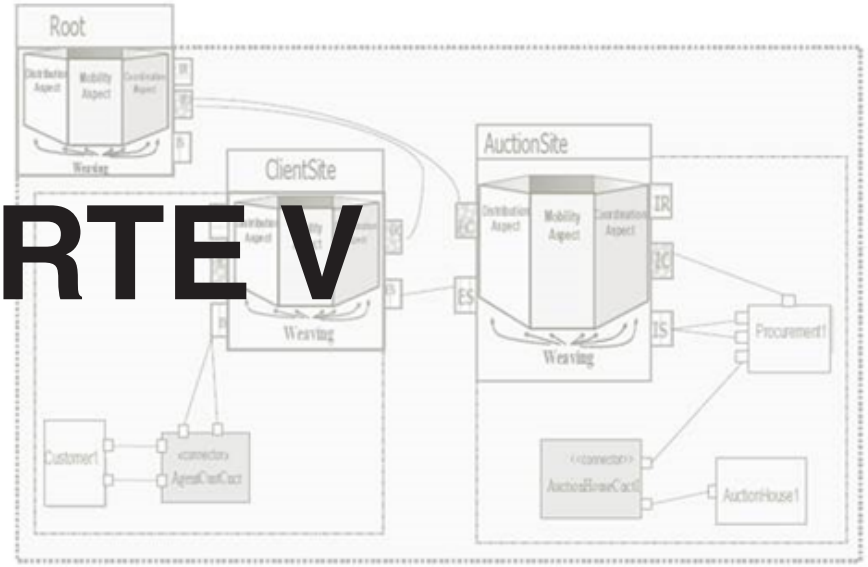
PARTE IX: LOS AMIGOS

Los amigos 2397

Apéndice 2441







PARTE V

ARQUITECTURAS SOFTWARE (2002-2010)

PRÓLOGO DE PATRICIO LETELIER

ARQUITECTURAS SOFTWARE (2002-2010)

PARTE V

Me incorporé como profesor y como alumno del programa de doctorado del DSIC en la UPV el año 1992, integrándome también ese mismo año en el grupo de investigación liderado por Isidro. Recuerdo las reuniones de grupo en las cuales Isidro impulsaba una disciplina de presentación y discusión de artículos. Cada semana se presentaba algún tema para ser comentado. Para un recién llegado como yo, a solo dos años de haber terminado Ingeniería en Informática y sin ninguna experiencia en investigación, fue un gran desafío, el cual asumí con el mismo hábito que nos transmitía Isidro. Pero, ¿qué era la Ingeniería de Software en 1992?, por supuesto algo bastante diferente a lo que hoy conocemos :-). Sin embargo, para un grupo de personas interesadas en trabajar en esta disciplina la necesidad era la misma que podría ser actualmente, esto es: saber hacia dónde dirigir nuestra energía, especialmente cuando el esfuerzo debía conllevar conseguir nuestro doctorado. Respecto de esto, son dos características que destacaría de Isidro como nuestro líder de grupo: su visión respecto de líneas de trabajo en ingeniería de software y su capacidad para transmitírnosla con gran convencimiento y pasión.

En nuestro grupo una de las líneas de investigación más fructíferas en cuanto a producción de artículos y desarrollo de tesis fue la asociada a Arquitecturas Software. Mirando hacia los orígenes de esta temática en nuestro grupo, aún recuerdo los acetatos que presentaba Isidro remarcando que el desarrollo de software, y tal como sucedía en otras ingenierías asociadas a la producción de hardware u otros dispositivos físicos, debía basarse en el ensamblado de componentes y que tanto la estructura como el comportamiento del todo debían estar determinados por operaciones de composición.

Primero trabajamos en el ámbito más detallado de las especificaciones conceptuales basadas en el modelo orientado a objetos, en el cual tenían protagonismo las operaciones de agregación y especialización. La elegancia de poder expresar declarativamente mediante nues-

tro lenguaje OASIS la estructura y el comportamiento de sociedades de objetos marcaba una gran diferencia con respecto de los lenguajes imperativos disponibles en aquellos años. En este marco se desarrollaron varias tesis que profundizaron en la expresividad del lenguaje y en su semántica operacional desde la perspectiva de diferentes formalismos, tales como programación lógica concurrente o redes de Petri. En OASIS 3.0 conseguimos definir un lenguaje muy completo en cuanto a mecanismos de composición y refinamiento, y basado en una variante de lógica dinámica que incorpora operadores deónticos. La automatización de la ingeniería de software y en particular la generación automática de código a partir de modelos OASIS estuvo siempre presente en nuestros trabajos. También la evolución de modelos fue un tema protagonista y se abordó de una forma elegante y efectiva mediante un meta nivel de especificación en el cual metaelementos ofrecen operaciones para gestionar los modelos OASIS. En cada uno de estos trabajos en torno a OASIS, además del trabajo de definición del lenguaje, se desarrollaron interesantes prototipos de herramientas, que pese a las limitaciones tecnológicas de esos años, conseguían ilustrar el potencial de dichos trabajos.

El paso hacia Arquitecturas Software de alto nivel se dio de forma natural. Mientras otras propuestas solo trabajaban a nivel de componentes de gránulo grueso, nuestra propuesta era utilizar OASIS para el detalle de los componentes y usar extensiones de OASIS como lenguaje para el nivel de abstracción mayor. Esta propuesta más global se materializó en PRISMA, un lenguaje para especificación de arquitecturas software que además integra desarrollo basado en aspectos (AOSD) y desarrollo basado en componentes (CBSD). PRISMA abrió nuevas líneas de trabajo, no solo en la definición del lenguaje sino también en desafíos asociados a la distribución y movilidad, coordinación, evolución dinámica y requisitos, y junto con ello se llevó a cabo la construcción de herramientas con un alto nivel de desarrollo y funcionalidad.

Isidro, los homenajes en esta etapa de tu vida probablemente suenen a despedida y obviamente son un buen momento para destacar tu trayectoria profesional. Primero quiero dejar claro que no tengo la intención de despedirme. Luego, respecto de tu trayectoria, además del volumen evidente en este documento, me conmueve cómo puede verse también como la trayectoria de cada uno de nosotros entremezclada con la tuya. Este homenaje a ti despierta una reflexión y sentimiento en todos aquellos que hemos trabajado contigo. Para mí, es un orgullo haberme desarrollado como docente e investigador dentro de nuestro grupo de investigación, siendo parte de tu equipo de trabajo.

Patricio Letelier Torres

Modeling dynamic aspects in architectures and multiagent systems

J. Ferrer, A. Lorenzo, I. Ramos, J.A. Carsí, J. Perez

Departamento de Sistemas Informáticos y Computación

Universidad Politécnica de Valencia

{ jferrer,alorenzo,iramos,pcarsi,jperez }@dsic.upv.es

Abstract

A multiagent system specification language should be able to model the evolution aspect of Information Systems (IS). OASIS, our object-oriented model, allows the evolution of an IS using the reflection concept. This facility is used to manage, in a homogeneous way and with a unique language, the existing dynamism in functional, interaction, distribution and composition aspects of multiagent systems.

1 INTRODUCTION

The domain model that represents the software system evolves along time, so the specification languages should deal with this aspect in order to obtain valid models. Also, the selected system architecture might evolve, and this is an important issue to take into account in order to select an architecture definition language.

Both, statics and behaviour of a system evolve, also business rules might change in future states. In agent-oriented systems, the dynamics are more relevant, and the fact that the same agent is able to play several roles depending on the scene, must be taken into account. However, the same agent will show different state and behaviour facets along its life. The multiagent systems specification languages should be able to deal with the role evolution aspect.

A software agent has the learning property, that is, an agent is able to modify its own properties to reach its goals (or to plan an easier way to obtain them). This feature is also an important part of the evolution aspect.

Moreover the system architecture might be modified by adding or changing its components and connectors.

Ferrer, J., Lorenzo, M. A., Ramos, I.,Carsí, J. A., Pérez, J. Modeling Dynamic Aspects in Architectures and Multiagent Systems. Proc. of the Constraint Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2002. 2002.

This paper presents the evolution model associated to OASIS, which deals with the basic agent model, the coordination model, the distribution model and the component model of a multiagent system, and solves several problems that exist in the specification of its dynamics. So OASIS allows modeling several existing crosscutting concepts of the agents and, in a homogeneous way, it allows to model another (meta) crosscuting concept of these aspects: evolution.

First of all, the description of a system example is shown. Section 3 deals with the dynamism in the functional aspect. The evolution in the interaction model is explained in section 4. Section 5 introduces the evolution in the distribution aspect. The dynamic architectures are outlined in section 6. Finally, conclusions and future works are presented.

2 THE EXAMPLE

We will follow the example of a multiagent system proposed at [Nor97]. It represents a traditional fish auction institution based in the Dutch auction protocol (downward-bidding). The developed activities are:

- Ships arrive stocked with fish, which is distributed in several lots to be registered as goods to be auctioned.
- At the same time, buyers are registered and ask for a credit line to the buyer admitter.
- Goods are shown to bidders before the selling process begins. The auctioneer establishes the starting price, usually higher than the market one.
- The selling process consists of reducing the initial price by a constant amount while no bid is done, while the actual price gives benefits to the seller. If a buyer bids, the process stops and the bidder can buy at the actual price while he has enough credit, the rest of the lot can be obtained by the other buyers until it is totally sold.
- The process continues by decreasing prices until the session closes when all goods of the ship are sold.

3 EVOLUTION IN THE AGENT MODEL: DYNAMICS IN THE FUNCTIONAL ASPECT

3.1 SOFTWARE AGENT MODEL

The object model presented in [Let98] can be easily extended to deal with the functional aspect of an agent [Gir01]:

- ***Autonomy***: An autonomous agent is able and is authorized to work (to reason, to act and react) in an independent fashion, instead of being directed by other agents. Therefore, an agent is an active software entity, owner of its own thread of control and constantly in execution.
- ***Reactivity***: A reactive agent simply responds to orders of services. This is the same concept used in the objects oriented paradigms, and it causes a response to their clients' orders of services.

- ***Sociability***: The sociability characteristic specifies that an agent must be able to communicate with other agents of the system, by using a common ontology.
- ***Pro-activity***: With the pro-activity feature the agent is constantly looking for something to do. An object can only change its state as a result of serving the requests of other objects, while an agent, in addition to this, permanently consults its surroundings in order to obtain facts or situations that can make it change its state in order to purchase its goals.
- ***Mental notions***: The fact that an agent has mental notions means that it includes, explicitly, a logical model of the world that surrounds it:
  - *Beliefs*: They are the facts about the environment that the agent believes.
  - *Desires and Intentions*: Desires represent the goals of the agent. Within each goal exists a set of associated plans, which determine the steps to follow to reach the goal. The intention is the plan that is chosen to be implemented.
  - *Rationality*: The rationality characteristic determines that an agent acts to fulfill its goals and will never deviate from its agenda.
  - *Veracity*: The veracity is assured since the querying or observation's semantics are functions, so that at any moment it will return the real value of an attribute and will never be able to deceive with false values.
  - *Learning*: Learning is the characteristic that allows the agent to adapt to its environment, and in addition, to capture knowledge of its surroundings. The agent, using its learning property, can modify and reach an evolution of its mental structure as well as its behavior.

If we also desire distributed agents, to deal with coordination, composition, evolution, etc, each of these aspects have to appear in the model. By using OASIS, the agents will be represented through two objects of the language: one which implements the complete functionality and the other that reifies into its state the (structural and behavioural) properties of the first one (which can be dynamically modified).

We can visualize in a graphical way the model as a prism (see figure 1) where each face represents one aspect: functionality, coordination, distribution, composition, role evolution and learning evolution. Each aspect is divided into two parts: the higher one (metalevel) and the lower one (basic level). All the properties of the basic level that can be modified have their modification services defined at the metalevel interface.

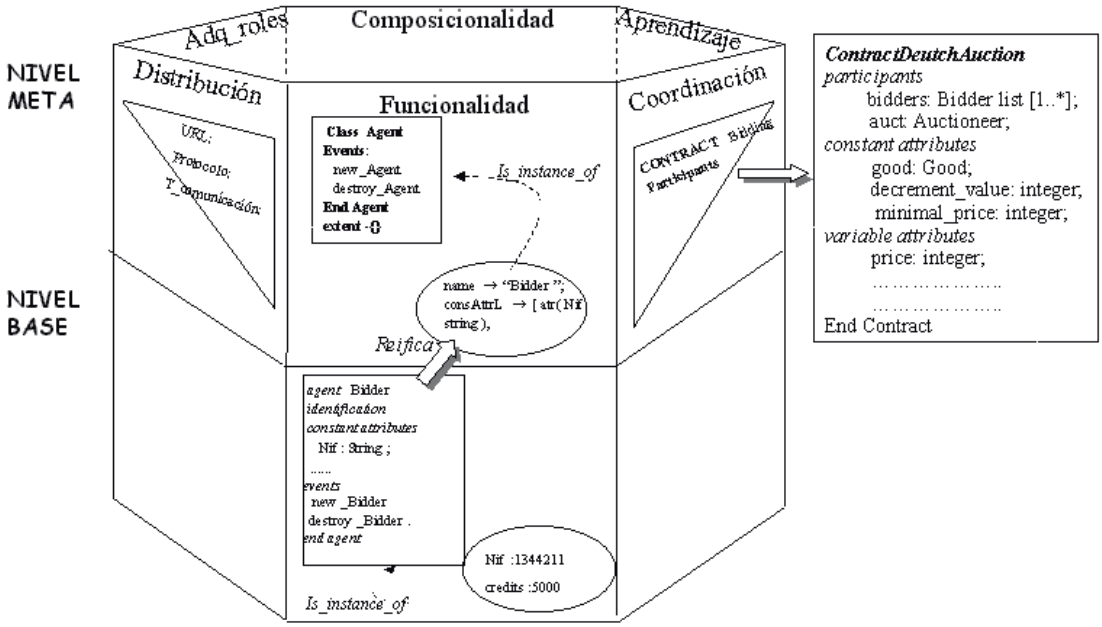


Figure 1: Agent model: Aspects representation

The aspects that conform the model are:

- *Functionality*: Adjusted to the proposal of [Gir01] and attached with the reflection concept used in AFTER [Car99].
- *Coordination*: Aspect attached at the metalevel by following the contract theory exposed at [And99] and extended in OASIS [Fer99].
- *Distribution*: Also expressed at the metalevel under the architecture model exposed at [Cor01].
- *Learning*: It is associated to the versioning evolution concept. That is, the agent gives new properties to its type, so its new definition is captured in the application, without the necessity of old version maintenance.
- *New roles acquisition*: We are talking now about temporal evolution. As in learning, the agent changes its properties in a concrete scene but it is able to return to the specification given by other scenes.

OASIS defines each agent as the aggregation of two object societies:

- *Meta-agents*: Metaobjects that when are aggregated, according to the inclusive aggregation semantics, build an object which offers the aggregated functionality of all its aspects (composition, distribution, interaction, functionality and evolution).
- *Base-agent*: The objects of the base level, which presents all the aspects of the model, that is: has one functionality, are distributed, are able to learn, interact, might be composed, and play roles in several scenes.

This aggregated society of objects is the realization in OASIS 3.0 of the agent model under the aspects shown in the prism. Once the extensions of the language are defined in order to express all the aspects mentioned, a compiler will translate each one, in an automatic way, to the suitable technology. To keep the easy maintenance

and scalability, it will be reasonable to generate a code that follows the aspect oriented programming rules (AOP) [Kic96].

3.2 DYNAMICS IN THE AGENT MODEL

With the OASIS model, the schema evolution does not only deal with the modification in the conceptual schema that the multiagent system defines, but also with the migration of the existing extension to be compatible with new features [Car99].

The evolution problem has been considered in OASIS by using reflection. So a model with behaviour reflection is achieved, and this allows the designer to validate and modify object oriented systems to conform schema evolution.

An agent schema is reified as a metaentity state. This metaentity is composed of other entities, which represent whatever aspect of the agent model (coordination, composition, distribution, new role acquisition...). The metaentities state defines both structure and behaviour of all their possible instances. These properties can be modified through the services offered in their interface.

Metaentities offer a double vision: on one hand they are entities with local state which offer a set of services which allow to modify the defined type. On the other hand they are entities that create and destroy their associated entities (extension).

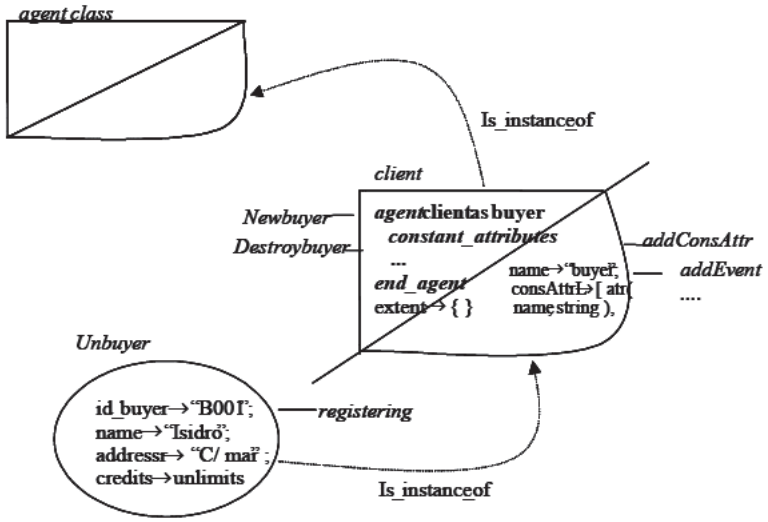


Figure 2: Double vision of the *Buyer* agent

The definition of the semantics of the change (and its extension evolution) is attached to the services which modify the metaentity state (i.a. type properties).

Due to the possible existence of infinite metaentities hierarchy, the transitive closure of metalevel is constructed by making the *Agent_Class* metaentity an instance of itself.

3.3 ROLES

We define the multiagent system as an institution composed by several scenes. In each one of these scenes a concrete functionality is carried out by a set of agents which develop a well defined role. The performative structure (transition scene diagram) of the institution [Nor97] describes the order in which the different scenes could be performed.

The role concept describes a partial view of the agent's behaviour and constitutes the basic abstraction which allows to adapt it to the different scenes where it plays.

The behaviour of an agent changes depending on the scene where it is. The same agent is able to offer several faces during its life. That is, the global properties on an agent will be defined as an aggregation of the individual properties of the roles that it plays. In the entity that represents the agent type, the life cycle that can follow the agents of this type will be defined, according to the performative structure of the institution.

The OASIS evolution model deals with the dynamics associated to the several roles that one agent is able to play. One change in the environment state, or a message received, could make the agent to acquire one responsibility guided by its goals. In this case, the agent will active the event in its metaagent of which it is instance of, to migrate to the role which handles the acquired responsibility.

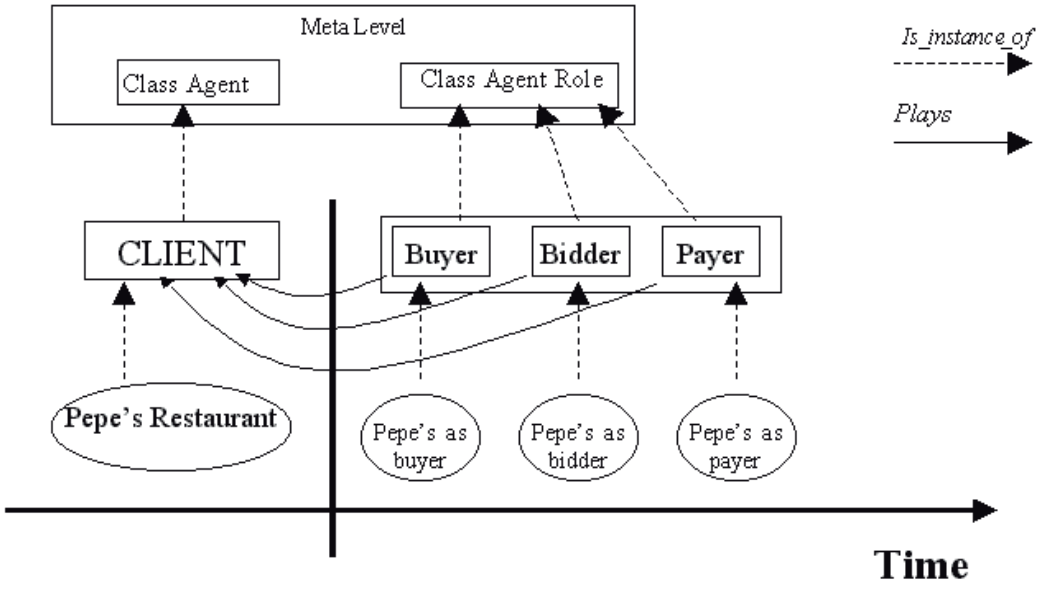


Figure 3: Cross roles evolution

3.4 LEARNING

Learning is one of the mental notions that agents have, that is, they are able to change their properties in a dynamic way. The learning problem is treated by the reflection provided by OASIS. If one agent wants to learn (change its properties), it should modify the entity that represents its type (meta-agent) by the activation of the corresponding services.

When an agent wants to change its meta-agent state, it has to know if it has to change the properties of the agents that share its type, learning all together. If it does not want its agent type extension to learn, the agent will create a new meta-agent, usually a specialization of the first one, with the learned features.

4 EVOLUTION IN THE INTERACTION MODEL: DYNAMICS IN THE COORDINATION MODEL.

4.1 INTERACTION MODEL

[And99] and [Fer99] indicate that the explicit definition of contracts among agents allows to specify scalable and easy to modify systems. Basically, the contract is defined as an entity which links several participants and looks for their correct coordination, in a totally transparent way, by defined business rules. Another task of a contract is the exception handling occurred during interactions. In this way it is possible to separate the domain concepts at the object level from the business rules that govern its behaviour.

The interaction model of a multiagent system is usually more complex than the object one, that is because the goal of an interaction is not only a simple message passing and its exception handling. The goal of an interaction among agents usually is to achieve a goal, and it implies an agreement by negotiation (see figure 4).

```
contract DutchAuction
participants
    bidders: Bidder list [1..*];
    auct: Auctioneer;
constant attributes
    good: Good;
    decrement_value: integer;
    minimal_price: integer;
variable attributes
    price: integer;
events
    new
      open_auction(g:Good,dv:int,mp:integer);
    destroy adjudicate;
    destroy close;
    decrement;
valuations
    [decrement]  price := price –
      decrement_value;
    [bidder:auctioneer::enter] bidders :=
      insert(bidders,bidder);
…
trigger
    close when {price <= minimal_price};
protocols
    open:
    OPEN =
      auctioneer:Buyer::open_auction.(ENTER
      + close);
    ENTER = bidder:auctioneer::enter.(ENTER
      + BID);
    BID: (bidder:auctioneer::bid &
      auctioneer:bidder::adjudicate) +
      DECREMENT;
    DECREMENT: close +
      auctioneer:bidders::offer_decrement & bid;
end_contract
```

Figure 4: Contract between bidder and auctioneer with the Dutch auction protocol

4.2 DYNAMICS IN THE INTERACTION MODEL

Section 3.1 deals with the entities evolution in a volatile domain. The problem of this solution is that the changes of the interaction properties imply the modification of their participants. This problem is easily solved by using contracts in the interaction model. This is reason why the evolution of a contract is transparent to its participants.

Once again, the dynamics in interaction among agents is solved by reflection. When a property of an interaction changes, the analyst will active the corresponding services in the metalevel, but only in the coordination and distribution aspects.

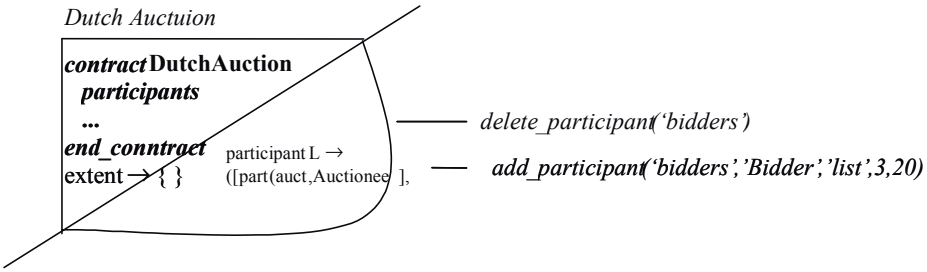


Figure 5: Modification on the cardinality of contract participants

Figure 5 shows the sequence of events which modify the cardinality of the possible bidders in an auction. It states that to start the auction, there should be at least three bidders and no more than twenty. The events : *delete_participant* and *add_participant* are defined in the signature of the metaclass *Contract*. Its activation modifies the state of a contract type, that is, adds or eliminates participants.

5 AGENTS MOBILITY: DYNAMICS IN THE DISTRIBUTION ASPECT

5.1 DISTRIBUTION MODEL

[Cor01] indicates that the creation of agents and the manipulation of their physical location are responsibilities of the metalevel. Moreover in the distribution modeling, the design proposal has to be independent from the final system implementation.

The same application has to work under different structural topologies. By modeling independently the agent localization from other aspects, it is able to assure a framework in which the system design is not determined by the physical location of its parts. Even more we will have the possibility to execute our own prototype over different topologies in order to evaluate the one which accomplishes the established requirements.

[Car99] presents the metalevel instances as instance factories. So, those entities will manage the information about distribution of their extensions, by this, new semantics is added to the *is_instance_of* relationship. This information will be queried by the entities defined in the interaction model: *contracts*.

In the example showed in figure 6, the *Buyer* agent, will store information about the distribution of the set of buyers that participate in the auction. In the case that the *auctioneer* adjudicates a lot to a *buyer*:

(1)(2) The *auctioneer s*hould communicate to its meta-agent (*Auctioneer*) the adjudication. This entity will send the message to the buyer meta-agent (*Buyer*) through the *agent_class* (which is the entity that knows the agent that serves the *adjudicate* service.

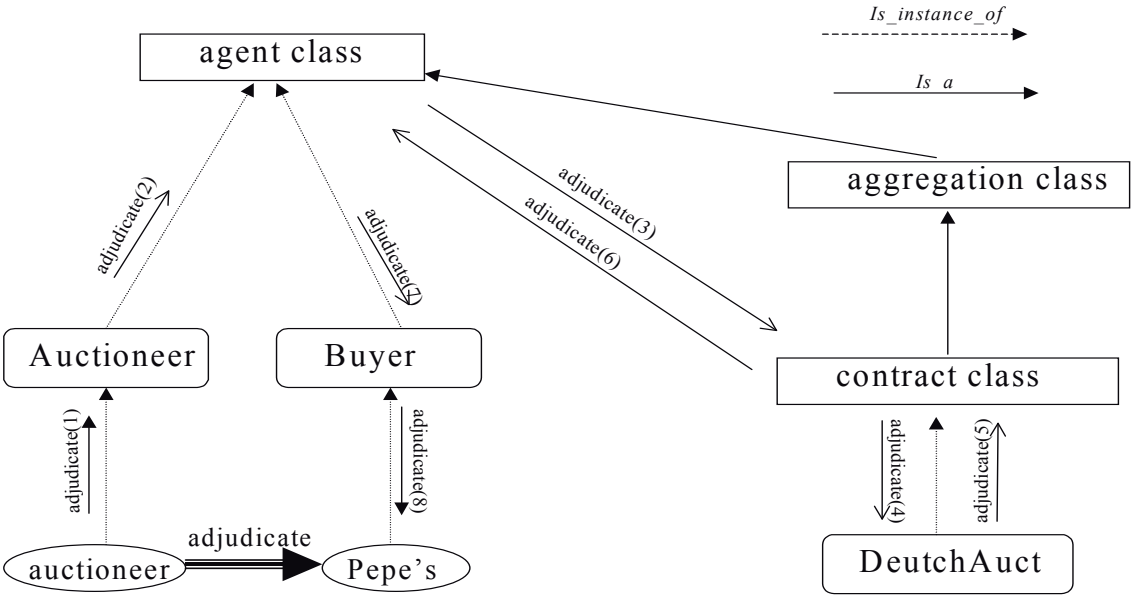


Figure 6: Distribution management by reflection

(3)(4)(5)(6) This call is intercepted by the contract which defines the adjudication protocol (*DutchAuction*). If the contract execution ends in a correct way (without exceptions) the selling process is right and

(7)(8) the buyer meta-agent, which knows the location of the buyer, will communicate the adjudication by sending it a message.

5.2 MOBILE AGENT MANAGEMENT

The distributed multiagent system definition languages should deal with the possible existence of mobile agents that is agents that could be executed in several locations.

OASIS deals, by reflection, with the modification of the agents' location. To modify or assign the location of one agent, the corresponding event will be activated in its metaentity, passing as a parameter the new location.

6 DYNAMIC ARCHITECTURES: DYNAMISM IN THE COMPOSITIONAL ASPECT

6.1 AGENT/COMPONENT MODEL

[Lor01] presents OASIS as an architecture definition language (ADL), and the entities that conform its model are:

- *Component*: Each component is a software artifact which provides a well-defined functionality.
- *Contract*: A contract is a software connector which relates components and defines their interaction properties.

There are two types of components in OASIS:

- *Elemental component*: They represent roles of participants and their evolution to other ones and their possibility to extract the properties of a generic pattern. The concept of role describes a partial view of a component behaviour and is constituted as a basic abstraction which allows it to participate in several compositions.
- *Activity component*: In its finer granularity, they emerge from the aggregation of elemental components (or other activities components) and one contract.

In a higher level of granularity, the components get into a composition process to construct components which offer a higher functionality.

The software architecture associated to a system will be defined by an activity component (configuration) which offers the global system functionality. The same composition mechanisms are able to be translated into the agent composition.

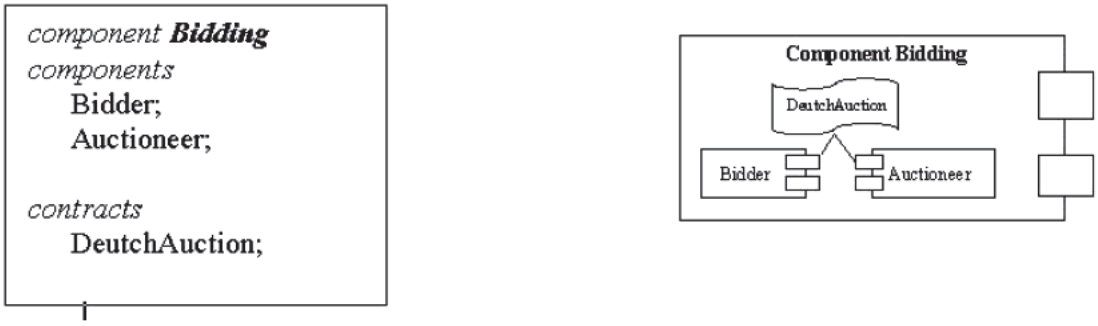


Figure 7: *Bidding* component

Figure 7 shows the *Bidding* component, which offers the functionality of the bidding scene, so it will be the functional aggregation of the components *Bidder* and *Auctioneer* by the negotiation properties defined at the *DutchAuction* connector (downward-bidding).

6.2 DYNAMIC ARCHITECTURES

A common problem in ADLs is that they have to deal with the evolution of the architectures that they define, in component level as well as in a connector one.

As in [Loq00], metalevel deals with the evolution of architectures defined with OASIS. That is because components and contracts reify metaentities which will offer the necessary services to modify their properties. Their extension migration semantics are indicated too in the services definition

The modification of system architecture is a kind of evolution totally different to that described in chapter 3, this is due to the modification or addition of components and/or contracts does not affect to the properties defined at the agents or classes that constitute these components or interact with these contracts.

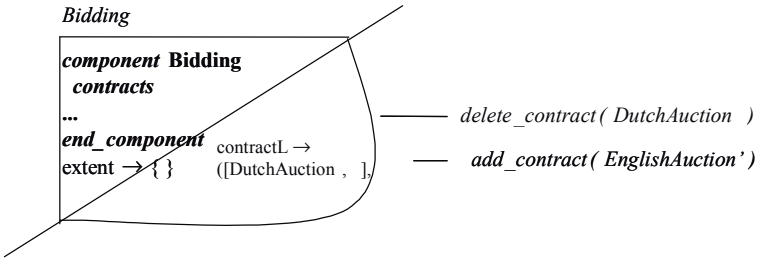


Figure 8: Connector modification into a component

Figure 8 shows the events to be activated in order to modify the contract type to use among the components of the *Bidding* component. As **Figure 8** shows, the *Bidding* scene will not use de *Dutch auction* protocol as described in chapter 2, instead of this, the *English auction* protocol (upward-bidding) will be used. The events *delete_contract* and *add_contract* are defined in the *Component* metaclass signature. Their activation modifies the state of certain component type, concretely, removes or adds types of connectors used among their components.

7 CONCLUSIONS AND FUTURE WORK

The agent-oriented design is a complex activity which manages active and reactive software modules which have to include knowledge and learning notions. This is because the model has several crosscutting concepts that make it difficult to maintain or scale.

There exist several formalisms to model multiagent systems, but the OASIS model goes further. It allows dealing in an easier way than the others, the dynamism and evolution aspects that exist in an agent society. More and more, the model deals with the system architecture and agent mobility. The reflection that exists in the model allows dealing with the schema and architecture evolution. This is because there are several services which allow to modify the properties of the entities which populate the agent system, this includes classes of objects, agent types, contract types, component types and connector types.

Another interesting aspect considered in the model is the implicit evolution that exists in the mental notion of learning that agents have. That is, an agent will be able to modify its properties as it acquires new knowledge from the environment.

Metaentities deals with the information about the location of their entities, and will be responsible of the management of this information. So the distribution dynamism (mobile agents) aspect is considered at this level.

Software agents have to be modeled from different points of views: functional, coordination, composition and distribution. All them can be modeled using a common language: OASIS. Moreover the language allows to model the dynamic aspect of the system.

Once the different aspects of a multiagent system are able to be modeled using OASIS, our future work is focused on the development of a model compiler that automatically generates programs which will follow the aspect oriented programming rules [Kic96] (i.e. using AspectJ [AspectJ]).

8 References

[And99] Andrade L., Fiadeiro J.: "*Interconnecting Objects via Contracts*". OOPSLA'99. Tutorial 56.

[AspectJ] *AspectJ Home Page,* http://www.aspectj.org/, Xerox PARC, USA.

[Car99] Carsí J. A.: "*OASIS como marco conceptual para la evolución del software*". Ph.D. Thesis 1999. DSIC (Universidad Politécnica de Valencia).

[Cor01] Correas J., Bueno F, "*A Configuration Framework to Develop and Deploy Distributed Logic Applications*". ICLP'01, CLPSE'01 (Constraint) Logic Programming of Software Engineering. 1/12/01. Chipre.

[Fer99] Ferrer J. "*Evolución de OASIS/OO-METHOD hacia el diseño por contratos*". Research work DSIC'99 (Universidad Politécnica de Valencia).

[Gir01] Giret A., Ramos I., "*Full software agents as OASIS 3.0 objects*". ICLP'01, CLPSE'01 (Constraint) Logic Programming of Software Engineering. 1/12/01. Chipre.

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "*Un enfoque formal para el modelado conceptual orientado a objeto*". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721-663-0, 1998.

[Loq00] Loques, O., Sztajnberg, A., Leite, J. and Lobosco, M., "*On the Integration of Meta-Level Programming and Configuration Programming*", In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[Lor01] Lorenzo A, Ramos I., Carsí J.A, Ferrer J.: "*Hacia un modelo de Arquitectura Software: Un caso de estudio*". Sevilla I Jornadas Dolmen pag 1- 10

[Kic96]] Kiczales G. *Aspect-oriented programming.* ACM Comp. Sur, 28(4), Dec 1996.

[Nor97] Noriega P. "*Agent mediated auctions: The Fishmarket Metaphor*". Ph.D. Thesis. Universitat Autònoma de Barcelona. 1997.

[Wie95] Wieringa R. et al, "*Using Dynamic Classes and Role Classes to Model Object Migration*", in theory and Practice of Object Systems, Vol.1, Ed. John Wiley and Sons, 1995

PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos

Jennifer Pérez, Isidro Ramos, Ángeles Lorenzo, Patricio Letelier, Javier Jaén

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
46022 Valencia – España
{jeperez|iramos|alorenzo|letelier|fjaen}@dsic.upv.es

Resumen. El desarrollo de sistemas software actuales es necesario abordarlo mediante plataformas que permitan describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables. PRISMA es un modelo arquitectónico basado en aspectos y componentes que utiliza un lenguaje de definición de componentes (componentes, conectores y sistemas) para definir los elementos de la arquitectura a un elevado nivel de abstracción y un lenguaje de configuración para diseñar la arquitectura de sus sistemas software. El lenguaje de definición de componentes de PRISMA potencia la reutilización, permite la importación de COTS y aglutina dos aproximaciones de desarrollo de sistemas software de actualidad, el Desarrollo Software Basado en Componentes (DSBC) y el Desarrollo Software Orientado a Aspectos (DSOA). El lenguaje de configuración permite definir instancias a partir de los elementos definidos o de las COTS importadas, especificando la topología existente para un sistema software específico. El metanivel del PRISMA y las propiedades reflexivas de los lenguajes diseñados permiten tanto la evolución de los elementos, como la reconfiguración dinámica de la topología.
Palabras clave: Arquitecturas software, lenguaje de definición de componentes, lenguaje de configuración, componentes, aspectos, reutilización, evolución.

1. Introducción

Los sistemas de información cada vez son más difíciles de desarrollar debido a sus complejas estructuras y a la dinámica y competitividad del mercado que implica una evolución constante de los requisitos del sistema. Por este motivo, los entornos de desarrollo deben permitir construir aplicaciones con arquitecturas complejas, distribuidas, evolutivas y reutilizables. El enfoque orientado a objetos por sí solo no permite abordar los requisitos de los sistemas software complejos actuales. Debido a esto, han surgido dos tendencias en el desarrollo de sistemas software, el Desarrollo de Software Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA).

En la actualidad es fundamental que los procesos de producción de sistemas software minimicen el tiempo y el coste de desarrollo y mantenimiento. Por este

Pérez, J., Ramos, I., Lorenzo, M. A., Letelier, P., Jaén, J. PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos. Actas de las VII Jornadas de Ingeniería del Software y Bases de Datos. 2002.

motivo, no sólo es importante que los entornos software permitan definir elementos reutilizables y evolutivos, sino que también han de posibilitar la importación de componentes COTS (comercial *off-the-shelf*).

Este trabajo presenta un enfoque para la construcción de modelos arquitectónicos basado en aspectos y componentes llamado PRISMA. Dicho enfoque utiliza una extensión de OASIS para la definición formal de la semántica de los modelos de arquitectura software. Dicha extensión ha sido necesaria porque OASIS [Let98] es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos y no permite especificar arquitecturas software. Sin embargo, OASIS permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos (compilación de modelos).

La extensión de OASIS ha dado lugar a un lenguaje de definición de componentes para definir los elementos de la arquitectura y a un lenguaje de configuración para especificar modelos arquitectónicos. Ambos pueden ser compilados para generar automáticamente el código del producto siguiendo el paradigma de la programación automática.

El lenguaje de definición de componentes de nuestro PRISMA potencia la reutilización, permite la importación de COTS y aglutina dos aproximaciones para el desarrollo de sistemas: el DSBC y el DSOA. Por ello, un elemento de nuestro modelo arquitectónico esta constituido por un conjunto de aspectos: funcional, distribución, coordinación, etc.

El metanivel de PRISMA y las propiedades reflexivas de los lenguajes diseñados dan soporte tanto para la evolución de los elementos, como para la reconfiguración dinámica de la topología. Esta característica del lenguaje permite definir un metanivel por aspecto que reifica las propiedades que deseen ser evolucionadas basándose en la reflexión de OASIS ([Car99]).

El modelo arquitectónico PRISMA que presentamos a continuación incorpora las ventajas de los DSBS y DSOA, reduciendo el tiempo de desarrollo y mantenimiento gracias a que sus elementos son reutilizables y evolutivos, y que además, es capaz de importar componentes COTS.

La estructura del trabajo es la siguiente: en primer lugar se presenta el modelo arquitectónico PRISMA y el metamodelo asociado. En el apartado 3, se presenta el ejemplo que se va a desarrollar a lo largo del documento. En los apartados 4 y 5 se describen los lenguajes de definición de componentes y de configuración respectivamente. En el apartado 6 se comenta el estado del arte del desarrollo de arquitecturas basadas en componentes y aspectos. Finalmente, en el apartado 7 se presentan las conclusiones y trabajos futuros.

2. Modelo de Arquitectura

PRISMA es un modelo para definir arquitecturas de sistemas software complejos. Dicho modelo permite definir los elementos mediante la composición de aspectos: funcional, distribución, coordinación, etc. Dependiendo del tipo de elemento de la arquitectura que se defina y del tipo de sistema que se esté desarrollando, el elemento tendrá especificados unos aspectos u otros.

El problema de la integración sin solape de código que existe en la programación por aspectos [Kic01] se resuelve en nuestro modelo gracias a que la integración se realiza a un alto nivel de abstracción y a que la generación de código de cada uno de estos aspectos se realiza de forma separada.

A continuación se detallan algunos de los aspectos que puede incorporar un elemento PRISMA, que son los que se van a considerar en este trabajo:

Aspecto Funcional: El aspecto funcional captura la semántica de la organización, mediante la definición de su estructura y su comportamiento.

Aspecto de Coordinación: El aspecto de coordinación nos define las reglas de negocio y la sincronización entre elementos durante su comunicación. Así como la coreografía o protocolo, extendiendo el concepto de *"contrato"* de Andrade y Fiadeiro ([And99]).

Aspecto Distribución: El aspecto de distribución captura aquellas propiedades que caracterizan el mecanismo de ubicación dinámico de las instancias de un elemento dentro del modelo de arquitectura software definido.

Aspecto de Evolución: El aspecto de evolución refleja la evolución del elemento debido al cambio de los requisitos de la organización. Este aspecto proporciona la dinámica del elemento, es decir, la capacidad de cambio de su estructura y comportamiento basándose en la evolución del software que plantea la reflexión de OASIS.

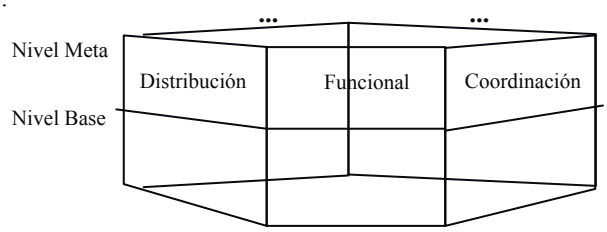


Fig. 1. PRISMA: Plataforma OASIS para Modelos Arquitectónicos

Un elemento (componente, conector o sistema) de nuestro modelo de arquitectura puede ser visto como un prisma con tantas caras como aspectos considere el tipo de elemento que representa, teniendo en cuenta que el aspecto de evolución no es una cara del prisma, sino la parte superior de la división en niveles base y meta de cada cara (ver figura 1). Cada una de ellas tendrá definido un nivel meta, siempre y cuando en el elemento se reifique alguna de las propiedades del nivel base del aspecto que representa dicha cara. La reificación puede afectar a cualquier sección de la especificación OASIS del aspecto, es el que le proporciona al lenguaje de definición de arquitecturas su capacidad reflexiva.

La reificación se realiza a partir del nivel base, lo que implica que no puede existir un elemento que tenga un aspecto especificado a nivel meta y no a nivel base. Por lo tanto, la relación entre el nivel meta y el nivel base de un aspecto es uno a muchos.

Nuestro modelo arquitectónico consta de distintos tipos de elementos. Todos los elementos de nuestro modelo arquitectónico forman parte del metamodelo (ver figura 2) para así poder modificarlos de forma dinámica a través de los mecanismos de reflexión del lenguaje. A continuación se citan y definen cada uno de ellos.

Interfaz: Una interfaz proporciona un conjunto de servicios con su aridad. Existen diferentes tipos de interfaces, uno para cada tipo de aspecto.

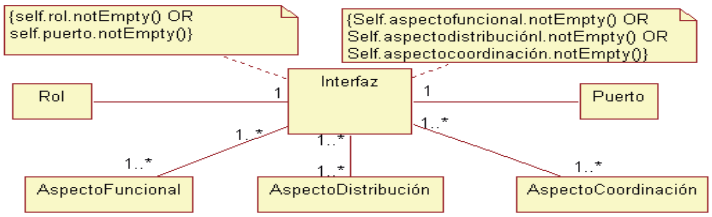


Fig. 2. Sección de interfaces del metamodelo de PRISMA

Aspecto: Un aspecto define de forma completa la estructura y el comportamiento de la componente desde un determinado punto de vista. Un aspecto puede verse como la unión de un conjunto de interfaces del tipo de dicho aspecto, más la especificación de la semántica de su estructura y comportamiento (cambios de estado, disparos, protocolos, etc). Existen diferentes tipos de aspectos: funcional, distribución, coordinación, etc. Definiéndose cada uno de ellos de forma disjunta.

Componente: Una componente es un elemento del sistema de información que no actúa como coordinador entre otros elementos. Está formado por un conjunto de aspectos (funcional, distribución, etc) y por uno o más puertos de entrada y de salida, cuyo tipo es una determinada interfaz. Las componentes interaccionan con el resto de elementos de la arquitectura a través de sus puertos.

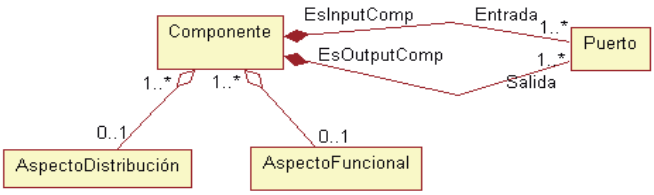


Fig. 3. Sección de componentes del metamodelo de PRISMA

Conector: Un conector (ver figura 4) es un elemento del sistema de información que actúa como coordinador entre varios elementos. Se compone de un aspecto de coordinación, un aspecto funcional, un aspecto distribuido y un conjunto de roles de entrada y salida que tienen como tipo una interfaz de carácter funcional o distribuido. Los conectores conectan y sincronizan componentes, conectores o sistemas a través de sus roles.

Sistema: Un sistema (ver figura 5)es una componente que encapsula un conjunto de conectores, componentes y otros sistemas correctamente conectados, pudiendo surgir de dicha composición propiedades emergentes. Además, se caracteriza por la definición de *binding links* entre los puertos del sistema y los puertos de las componentes que encapsula.

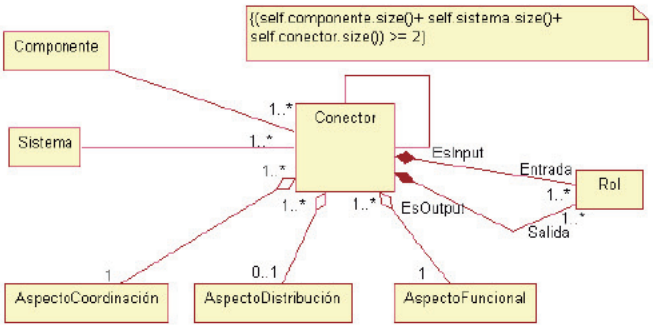


Fig. 4. Sección de conectores del metamodelo de PRISMA

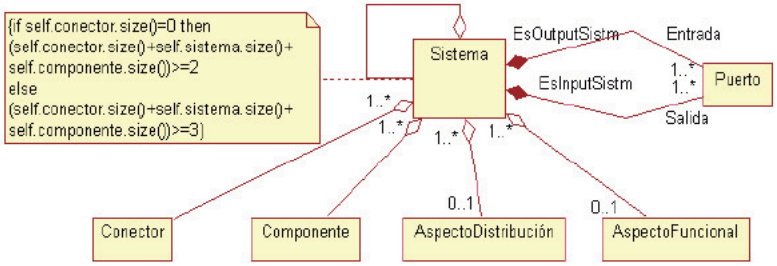
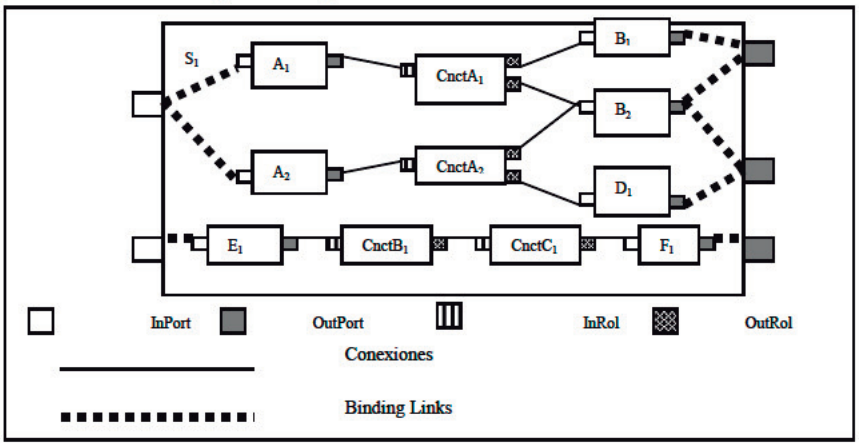


Fig. 5. Sección de sistemas del metamodelo de PRISMA

3. Ejemplo

A continuación se presenta un pequeño ejemplo definido en base a los conceptos de nuestro modelo arquitectónico. Este ejemplo se utilizará en el resto del documento para mostrar una posible especificación de arquitectura utilizando los lenguajes que se presentarán en los siguientes apartados.



S_1: Instancia del sistema S definida mediante el lenguaje de definición de componentes
A_1, A_2: Instancias del componente A definido mediante el lenguaje de definición de componentes
B_1, B_2: Instancias del componente B definido mediante el lenguaje de definición de componentes
D_1: Instancia del componente D definido mediante el lenguaje de definición de componentes
$CnctA_1$, $CnctA_2$: Instancias del conector CntcA definido mediante el lenguaje de definición de componentes
E_1: Instancia del componente E definido mediante el lenguaje de definición de componentes
F_1: Instancia del componente F definido mediante el lenguaje de definición de componentes
$CnctB_1$: Instancia del conector CntcB definido mediante el lenguaje de definición de componentes
$CnctC_1$: Instancia del conector CntcC definido mediante el lenguaje de definición de componentes

Fig. 6. Ejemplo de Sistema

4. Lenguaje de Definición de Componentes

El lenguaje de definición de componentes PRISMA permite definir los tipos de elementos de la arquitectura. Este lenguaje tiene como ciudadanos de primer orden: interfaces, aspectos, componentes y conectores, potenciando así la reutilización. De este modo, una interfaz puede ser utilizada por varios aspectos y un aspecto por varios elementos. Así mismo, tanto componentes como conectores y sistemas podrán reutilizarse en diferentes modelos arquitectónicos.

La importación de COTS se expresa gracias al carácter opcional de los aspectos, es decir, a la especificación de la estructura y comportamiento de las distintas facetas de la componente o el conector. Esta opcionalidad permite ver a estos elementos como cajas negras y con ello la posibilidad de incorporar componentes COTS.

4.1. Interfaces y Aspectos

En la especificación de una interfaz funcional se incluyen un conjunto de servicios con una signatura, describiendo la visibilidad parcial, que tiene un elemento del sistema o un elemento externo, de un determinado aspecto de un elemento. Un aspecto funcional se especifica mediante la plantilla OASIS presentada en [Let98], de forma que se definen los atributos, servicios, evaluaciones, derivaciones, precondiciones, disparos, restricciones de integridad y el **protocolo** de la componente que incluya este aspecto.

A continuación se muestra el extracto del código perteneciente a las interfaces funcionales y al aspecto funcional que especifica los servicios que incluyen dichas interfaces entre otras características funcionales.

```
Functional Interface I1(someone)
    Services(service1, service2)
End Functional Interface I1;
Functional Interface I2(someone)
    Services(service3)
End Functional Interface I2;
Functional Interface I3(someone)
    Services(sevice4)
End Functional Interface I3;
```

```
Functional Aspect F1
    Attributes …
    Services
        service1(…);
        service2(…);
        service3(…);
        service4(…);
         …
End Functional Aspect F1;
```

La especificación de una interfaz distribución es análoga a una de tipo funcional, la única diferencia es que los servicios ofrecidos en dicha interfaz contribuyen a la gestión del carácter distribuido de un elemento. A nivel de componentes dicha interfaz ofrece servicios de obtención de valores de métricas para la implementación de estrategias de distribución complejas tales como las que se describen en [Zin97, Che02]. A nivel de conectores y sistemas dichas interfaces ofrecen servicios de gestión del carácter distribuido de una colección de componentes. Un aspecto distribución especifica las políticas y estrategias de distribución a seguir en función de los valores obtenidos en tiempo real de las métricas establecidas.

La especificación de una interfaz de coordinación es análoga a las dos anteriores. Un aspecto de coordinación especifica los atributos, servicios, sincronizaciones entre interfaces y la coreografía de un conector, ya que este aspecto no se puede dar en las componentes o en los sistemas.

Las **sincronizaciones** definen las conexiones permitidas entre tipos de rol y de puerto. Esta especificación se realiza mediante interfaces, y no mediante puertos y roles de los elementos, para conseguir así un mayor nivel de abstracción y que un aspecto de coordinación pueda ser utilizado por distintos tipos de conecto, de no ser así, el aspecto de coordinación dejaría de ser un ciudadano de primer orden del lenguaje.

La **coreografía** especifica el proceso de coordinación que aplica el conector para sincronizar a los tipos de elementos que conecta. Al igual que las sincronizaciones, se especifica mediante interfaces.

Un ejemplo de especificación de una interfaz de coordinación y de un aspecto de coordinación con sus sincronizaciones, atributos, servicios y coreografía es el que se muestra a continuación:

```
Coordination Interface
I4(someone)
   services(all)
End Coordination Interface I4;

Coordination Aspect Co1
   Attributes …
   Services …
   Synchronizing Strategies
   {
    I1 << Ii;
    I2 >> Ij;
    I3 >> Ik;
    I3 >> Ir;
   }
   Coreography(…);
End Coordination Aspect Co1)
```

4.2. Componentes, Conectores y Sistemas

La **componente** contiene una series de puertos de entrada y de salida cuyos tipos son una interfaz funcional o distribuida. De forma que tenemos dos tipos de puertos, los que sirven para acceder aspectos funcionales de la componente y los que permiten acceder a aspectos distribuidos. Los puertos de entrada especifican los atributos y servicios que oferta la componente a otros elementos (comportamiento servidor). Los puertos de salida especifican los atributos y servicios externos que utiliza de otros elementos (comportamiento cliente). Siempre que se defina una componente se ha de especificar su aspecto funcional, excepto cuando son COTS. Mientras que la especificación de su aspecto distribución únicamente se realizará cuando se quiera dotar a la componente de un comportamiento distribuido. Se ha de tener en cuenta que

los tipos de los puertos de una componente únicamente podrán ser aquellas interfaces que formen parte del aspecto funcional o distribuido de dicha componente.

La definición de **conector** contiene un conjunto de roles de entrada y de salida, que tienen la misma funcionalidad y semántica que los puertos en la componente. Siempre que se defina un conector se ha de especificar su aspecto funcional y de coordinación, mientras que el de distribución es opcional tal y como ocurre en la componente. Al igual que en las componentes, únicamente se podrá asociar un tipo a los roles del conector con aquellas interfaces que formen parte del aspecto funcional o distribuido que incluye dicho conector. Sin embargo, las interfaces de coordinación no se utilizan en los roles, ya que expresan la visibilidad del aspecto de coordinación a elementos externos al sistema. Además, un conector incluye los tipos de componentes, sistemas y conectores que puede conectar.

A continuación se muestra la especificación del tipo de conector CntcA del ejemplo, incluyendo los aspectos funcional y de coordinación anteriormente definidos, los tipos de componentes A, B y D que puede conectar este conector, un rol de entrada y dos roles de salida, cuyas interfaces han sido definidas previamente.

```
Connector_type CntcA
   InRols IRol1: I1; End InRols;
   OutRols
        ORol1: I2;
        ORol2: I3; End OutRols;
   Connected Elements_Type: A, B, D;
   Functional Aspect F1;
   Coordination Aspect Co1;
End Connector_type CntcA;
```

Finalmente, la definición de **sistema** es equivalente a la de componente, ya que un sistema es una componente compleja. Por lo tanto un sistema es una componente formada por un conjunto de tipos de elementos conectados entre sí, cuya composición puede hacer emerger un aspecto funcional y un aspecto distribuido. Por ese motivo, un sistema incluye el conjunto de conectores, componentes y sistemas que conecta. También se ha de tener en cuenta que si el sistema incluye un tipo de conector deberá de contener todos los tipos de componentes, sistemas y conectores que conecte dicho tipo de conector. La especificación de la conexión entre los elementos que incluye el sistema y el propio sistema se realiza mediante los ***binding links***, que al igual que las sincronizaciones de los conectores se especifican mediante las interfaces.

4.3. Aspecto de Evolución

El lenguaje de definición de componentes puede construir automáticamente el metanivel a partir del nivel base, reificando solamente aquellos elementos de la especificación que sean susceptibles de evolucionar en el contexto del problema. Para expresar dicha reificación se hará uso del operador ***reify***, teniendo en cuenta que dicha reificación se realizará de forma automática.

La semántica del operador *reify* en cada uno de los niveles de granularidad de la especificación de los tipos de elementos será diferente y por lo tanto, se deberá asociar una generación diferente en el metanivel para cada uno de los casos detectados, que serán especificados mediante patrones. La evolución puede ser estructural o de comportamiento. La evolución estructural se consigue reificando las

propiedades de los elementos que definen a los elementos de una especificación. Mientras que la evolución de comportamiento se consigue reificando aquellas secciones de la especificación que definen el comportamiento del elemento. Este tipo de reificación nos da la propiedad cinemática del lenguaje (cambio de comportamiento sin cambio de estructura).

La especificación de los elementos puede expresarse como un término, pudiendo cualificar con el operador *reify* a cada uno de los subtérminos que constituyen ese término. Por lo tanto, una especificación puede verse como un árbol con diferentes niveles de profundidad, tantos como tipos de elementos de diferente granularidad existan en una especificación. Por esta razón, *reify* es un operador que tiene como entrada el árbol o subárbol de nivel base que se quiere reificar y produce como salida el árbol a nivel meta correspondiente. La función del operador consistirá en aplicar el patrón de generación del tipo de elemento que representa el árbol para generar las metapropiedades y metaeventos correspondientes y posteriormente, enraizarlos en el árbol de nivel meta existente que representa la especificación meta adecuadamente.

5. Lenguaje de Configuración

El lenguaje de configuración permite definir las instancias y especificar la topología del modelo arquitectónico. Para ello, se han de importar todos aquellos tipos de conectores, componente y sistemas, definidos mediante el lenguaje de definición de componentes, que se necesiten para un determinado modelo de arquitectura. Después se define el conjunto de instancias necesarias de cada uno de los tipo importados.

Finalmente se debe definir la topología del modelo, interconectando las instancias del modelo. Para ello se deberán instanciar las sincronizaciones de los conectores y los *binding links* de los sistemas.

```
Architecture Model  Ejemplo
Import
S,A,B,D,E,F,CnctA,CnctB,CntcC;
S1: S;
A1, A2: A;
B1,B2: B;
Connector CnctA1
  Coordination Aspect
   Connected Components
   { A1: A;
     B1,B2: B; }
   Synchronizing Strategies
   {IRol1 << A1.OPort1;
    ORol1 >> B1.IPort1;
    ORol2 >> B2.IPort1;}
  End Coordination Aspect;
End Connector CnctA1;
Connector CnctA2
  Coordination Aspect
   Connected Components
   { A2: A;
     B2: B;
     D1: D; }
D1: D;
E1: E;
F1: F;
CnctA1,CnctA2: CnctA;
CnctB1: CnctB;
CnctC1: CnctC;
   Synchronizing Strategies
   {IRol1 << A2.OPort1;
    ORol1 >> B2.IPort1;
    ORol2 >> D1.IPort1;}
  End Coordination Aspect;
End Connector CnctA2;
Connector CnctB1
  Coordination Aspect
   Connected Components
   { E1: E;
     CnctC1: CnctC;}
   Synchronizing Strategies
   {IRol1 << E1.OPort1;
    ORol1 >> CnctC1.IRol1;}
  End Coordination Aspect;
End Connector CnctB1;
```

```
Connector CnctC1
  Coordination Aspect
   Connected Components
   { F1: F;
     CnctB1: CnctB;}
   Synchronizing Strategies
   {IRol1 << CnctB1.ORol1;
    ORol1 >> F1.IPort1;}
  End Coordination Aspect;
End Connector CnctC1;
System S1
 A1, A2: A;
 B1,B2: B;
 D1: D;
 E1: E;
 F1: F;
 CnctA1,CnctA2: CnctA;
 CnctB1: CnctB;
 CnctC1: CnctC;

 Binding Links
 {S1.IPort1 >> A1.IPort1;
  S1.IPort1 >> A2.IPort1;
  S1.IPort2 >> E1.IPort1;
  S1.OPort1 << B1.OPort1;
  S1.OPort1 << B2.OPort1;
  S1.OPort2 << B2.OPort1;
  S1.OPort2 << D1.OPort1;
  S1.Oport3 << F1.OPort1;}
End System S1;
End Architectural Model Ejemplo;
```

6. Trabajos Relacionados

En la actualidad existen una gran variedad de trabajos relacionados con modelos arquitectónicos, componentes, lenguajes de definición de arquitecturas, reconfiguración dinámica de arquitecturas y todo aquello que guarda relación con el desarrollo de sistemas software complejos. Este gran esfuerzo de investigación persigue el objetivo de encontrar un consenso en los conceptos básicos de sistemas software complejos, así como su definición precisa y la metodología a aplicar para su desarrollo. Debido a que no se ha alcanzado un consenso, nuestra propuesta ha tenido en cuenta varios trabajos que incorporan las distintas áreas de interés para nuestro modelo. Cada uno de ellos aporta una parte de nuestra propuesta, mientras que nuestro trabajo integra dichas partes en un modelo arquitectónico único y hace emerger propuestas innovadoras.

El trabajo realizado por Garlan [Gar01] es un punto de referencia muy importante a la hora de establecer los elementos de un sistema software complejo, en el cual se introducen conceptos como sistema, conector, componente, puerto de entrada y puerto de salida entre otros. Nuestra propuesta ha considerado los trabajos de Andrade y Fiadeiro acerca del concepto de contrato, ya que hemos definido nuestros conectores como una extensión del contrato propuesto por su trabajo. Dicha extensión se produce debido a la incorporación del concepto de coreografía en el conector, concepto utilizado con una semántica distintita en los trabajos de [Mon02] y [Bra02].

En una línea cercana a la nuestra, se encuentra Model-Driven Architecture (MDA), un enfoque propuesto por la OMG (www.omg.org) para la integración e interoperabilidad de los sistemas. MDA enfatiza el modelado (con UML) independiente de las plataformas de implementación. MDA está basado en el modelado de diferentes aspectos y niveles de abstracción, explotando las interrelaciones y estableciendo transformaciones hacia su implementación.

Existe una gran variedad de lenguajes de definición de arquitecturas, cada uno de ellos presenta ventajas e inconvenientes, tal y como presenta el estudio realizado en [Med00]. Uno de los lenguajes que más se aproxima a los nuestros es el propuesto por Loques y Leite [Loq00] en su modelo R-RIO, en el cual incorporan aspectos de reconfiguración dinámica a través del metanivel. También, se encuentra en enfoques

orientados a un nivel de abstracción inferior, es decir, orientados a implementación como [McG02]. En GUARANÁ [Oli98] también se presenta un metanivel complejo que hay que definir durante la compilación, siendo su estructura ajena al modelo. Otro marco de referencia en nuestro trabajo es el Diseño Software Orientado a Aspectos (DSOA), en el que se encuentran trabajos como [Aos02]. Esta aproximación se aplica principalmente en la implementación y no se aplica para la especificación a un alto nivel de abstracción de sistemas arquitectónico que integren componentes y aspectos.

7. Conclusiones y Trabajos Futuros

Se ha propuesto un marco de trabajo para modelado arquitectónico basado en aspectos y componentes llamado PRISMA. Dicho marco se caracteriza por la integración que realiza de las aproximaciones de desarrollo DSBC y DSOA y las propiedades reflexivas a través de un metanivel que le da capacidad de evolución. PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables.

Los modelos arquitectónicos que siguen el PRISMA se especifican mediante dos lenguajes, un lenguaje de definición de componentes y un lenguaje de configuración, ofreciendo la posibilidad de importar componentes COTS y reutilizar aquellos componentes que se especifiquen.

Las líneas del trabajo futuras son muchas debido a la amplitud y complejidad del tema. Los trabajos a corto plazo se centran en la reconfiguración dinámica de los modelos arquitectónicos que se especifiquen y en los aspectos del prisma, tanto para incorporar nuevos aspectos como para especificar de una forma más detallada los aspectos de distribución, coordinación y evolución. Los trabajos a largo plazo consistirán en la aplicación de este modelo a distintos campos del desarrollo de software y la construcción de un compilador de modelos para PRISMA.

Referencias

[And99] Andrade L., Fiadeiro J., "*Interconnecting Objects via Contracts*". OOPSLA'99. Tutorial 56.

[Aos02] Aspect Oriented Software Development, http://aosd.net

[Bra02] Bracciali A., Brogi A., Canal C., "*Sistematic Component Adaptation*", In Proc IDEAS, La Habana, 2002.

[Car99] Carsí J. A., "OASIS como marco conceptual para la evolución del software". Ph.D. Thesis 1999. DSIC
(Universidad Politécnica de Valencia).

[Che02] Cheng S., Garlan D., Schmerl B., Steenkiste P., Hu N., *"Software Architecture-based Adaptation for Grid Computing"* , IEEE Conference on High Performance Distributed Computing. Scotland, July, 2002.

[McG02] McGurren F., Conroy D., "*X-ADAPT: An Arquitecture for Dynamic Systems"*, Workshop on Component-Oriented Programming, ECOOP, Málaga, Spain, 2002.

[Gar01] Garlan D., Cheng S., Kompanek A.J., "*Reconciling the Needs of Architectural Description with Object- Modeling Notations*", Science of Computer Programming Journal, Special UML Edition, Elsevier Science, 2001.

[Kic01] Kiczales G., Hilsdale E., Huguin J., Kersten M., Palm J., Griswold W.G., *"An Overview of AspectJ"*, proceeding of the European Conference on Object-Oriented Programming, Springer-Verlag, 2001.

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "*Un enfoque formal para el modelado conceptual orientado a objeto*". Universidad Politécnica de Valencia, SPUPV - 98.4011, ISBN 84-7721- 663-0, 1998.

[Loq00] Loques O., Sztajnberg A., Leite J., Lobosco, M., "*On the Integration of Meta-Level Programming and Configuration Programming*", In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[Med00] Medvidovic N., Taylor R.N., "*A classification and Comparison Framework for Software Architecture Description Languages",* IEEE Transactions of SW Engineering, Vol. 26, nº 1, January 2000

[Mon02] Monge R., Alves C. Vallecillo A., "*A Graphical Representation of COTS-based Software Architectures*", In Proc IDEAS, La Habana, 2002.

[Oli98] Oliva A., Garcia I.C., Buzato L.E.,*"The Reflective Architecture of Guaraná"*, Technical Report IC-98-14. Instituto de computación, Universidad de Campiñas, abril, 1998.

[Zin97] Zinky J., Bakken D., Schantz R., *"Architectural Support for Quality of Service for CORBA Objects,"* Theory and Practice of Object Systems 3(1), 1997.

Dinamismo en arquitecturas y sistemas multiagente

J. Ferrer, A. Lorenzo, I. Ramos, J.A. Carsí
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{alorenzo,iramos,pcarsi,jferrer}@dsic.upv.es

Resumen

Un lenguaje de especificación de sistemas multiagente, debe ofrecer la posibilidad de modelar, entre otros, el aspecto de evolución. OASIS permite, mediante la utilización del concepto de reflexión, definir el aspecto de evolución de un sistema. Esta facilidad es utilizada para abordar, de una forma homogénea y mediante un lenguaje único el dinamismo que se produce en los aspectos funcionales, de interacción, de distribución y de composicionalidad de sistemas multiagente.

1 Introducción

El modelo del dominio que representa un sistema software evoluciona a lo largo del tiempo, por tanto, los lenguajes de especificación, deben poder tratar este aspecto para que los modelos que describen sean válidos.

No sólo evoluciona la faceta estática de los sistemas, también cambia su comportamiento. Es decir, las reglas del negocio que se definen sobre un dominio, varían en estados posteriores en el tiempo.

En los sistemas orientados a agentes la dinámica es todavía mayor y se debe tener en cuenta la posibilidad que un mismo agente pueda jugar diferentes roles dependiendo del escenario donde se encuentre, es decir, un mismo agente mostrará diferentes facetas de estado y comportamiento a lo largo de su vida. Los lenguajes de especificación de sistemas multiagente deben poder tratar este aspecto de evolución.

Un agente software está dotado de la capacidad mental de aprendizaje, es decir, un agente puede modificar sus propiedades a lo largo de su existencia para poder lograr sus objetivos de una forma más sencilla o incluso plantearse unos nuevos, esta característica también se contempla en el aspecto de evolución.

Por último, cabe indicar que la arquitectura definida para un sistema software puede verse modificada por la inclusión o variación de sus componentes y conectores. Ante esta situación será importante mantener la consistencia entre la arquitectura elegida inicialmente y la nueva estructura que se ha generado para adecuarse a los requerimientos que han cambiado.

En este artículo se presenta cómo el modelo de evolución asociado a OASIS [Let98], permite dar cuenta del modelo básico de agentes, del modelo de coordinación, de distribución y de componentes de un sistema multiagente y permite tratar de una forma elegante y sencilla la problemática asociada a la dinámica de un sistema software. Debido al gran número de conceptos que abarca el trabajo, y por motivos de restricción de espacio, la problemática se aborda de forma horizontal, dejando el tratamiento detallado de cada uno de los aspectos nombrados para otros trabajos. Por otra parte, el artículo trata nociones tales como *meta-entidad, reflexión, reificación, evolución del esquema*, etc, que el lector no familiarizado con ellos puede encontrar detallados en [Let98], [Cue00] y [Car99].

El documento se estructura de la siguiente forma: tras la introducción, en el apartado segundo se presenta el ejemplo seguido. A continuación se muestra el tratamiento de la evolución en sistemas multiagente según el modelo propuesto por OASIS. El cuarto punto presenta el modelo de interacción, así como el tratamiento de su evolución. A continuación se introduce cómo trata OASIS el aspecto de distribución de un sistema y la movilidad de agentes. En el sexto punto se presenta el modelo de

Ferrer, J., Lorenzo, M. A., Ramos, I., Carsí, J. A. Dinamismo en arquitecturas y sistemas multiagente. Actas de las V Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2002). 2002.

composicionabilidad así como sus propiedades ante arquitecturas dinámicas. Para finalizar se establecen las conclusiones del trabajo.

2 Descripción del ejemplo

Para mostrar las propiedades del modelo de evolución de OASIS se presenta un ejemplo de sistema multiagente propuesto en [Nor97]:

El sistema representa una institución que se dedica a la subasta de bienes de acuerdo a las convenciones que se siguen en las Lonjas de Pescado tradicionales. El organismo está preparado para realizar las siguientes actividades, a cuyo cargo hay un representante de la organización como coordinador de las tareas realizadas:

- Los barcos llegan a puerto con el pescado, el cual se distribuye en lotes para ser registrados como bienes a subastar por el registrador de bienes.
- Simultáneamente, los compradores comunican su intención de participar en la subasta al registrador de compradores (*BA*), que toma los datos del comprador y pide al gestor de créditos la apertura de su crédito. El *BA* puede denegar la participación en la sesión al cliente que esté ya registrado en el sistema como moroso. Para que la sesión de puja comience debe existir un número mínimo de participantes.
- El género se da a conocer a los compradores a través de la exposición de los productos antes de comenzar el proceso de venta. El gestor de la subasta (*Auctioneer*) establece un precio de salida normalmente por encima del costo real de mercado. La venta consiste en rebajar el importe inicial de salida en una cantidad constante mientras ningún participante lo acepte y el valor ofertado represente ganancia para el vendedor. Si durante la rebaja un comprador detiene el proceso podrá adquirir al valor pujado, mientras tenga el crédito abierto, todos los artículos que desee, dejando el resto para los otros asistentes que hayan secundado la puja en riguroso orden de participación hasta que se termine el género.
- El proceso continúa animado por nuevas rebajas en el precio hasta que finaliza la sesión cuando termina la venta de los lotes de un barco.

El presente artículo se basa en la escena de puja, definida por el agente subastador (*Auctioneer*), los clientes jugando el rol de pujadores (*Bidder*) y el protocolo de interacción de subasta alemana (*DutchContract*).

3 Evolución en el modelo de agentes: Dinamismo en el aspecto funcional

3.1 Modelo de agentes software

El modelo de objetos que se presenta en [Let98] es fácilmente ampliable para poder abordar las nociones propias del aspecto funcional de un agente [Gir01]:

- *Autonomía*: Un agente autónomo es capaz de trabajar de forma independiente, sin tener que ser guiado por la solicitud de acciones por parte de otros agentes.
- *Reactividad*: Un agente reactivo será capaz de reaccionar (o no, dependiendo de sus objetivos) ante la solicitud de servicios.
- *Sociabilidad*: Los agentes se comunicarán mediante pase de mensajes. En el aspecto de coordinación se definen las propiedades de las interacciones entre los agentes, consiguiendo así mayor modularidad al independizar las propiedades funcionales del agente de las de sus interacciones.
- *Pro-actividad*: Un agente reactivo será capaz de reaccionar (o no, dependiendo de sus objetivos) ante cambios de estado del entorno.
- *Nociones mentales*: Las nociones mentales de un agente vienen descritas por su ontología asociada y se representan por:
  - *Creencias*: Son los conocimientos que tiene el agente sobre el entorno.
  - *Deseos e intenciones*: Son los objetivos asociados al agente y la forma de lograrlos.
  - *Capacidad de razonamiento*: Un agente debe actuar en todo momento según sus intenciones, donde una intención es un plan para lograr un objetivo.
  - *Veracidad*: La información sobre la que se basa un agente para tomar sus decisiones debe ser en todo momento real y actualizada.

- *Capacidad de aprendizaje*: Es la característica que permite al agente adaptarse al entorno y capturar conocimiento de otros agentes. El agente hará evolucionar su estructura mental y comportamiento.

El modelo se completa definiendo las propiedades de distribución, coordinación, composición y evolución. Usando OASIS los agentes serán representados para cada aspecto mediante dos objetos del lenguaje: uno que implemente la funcionalidad completa y otro que reifique en su estado las propiedades del anterior que pueden cambiar.

Podemos visualizar gráficamente el modelo mediante un prisma en el que cada cara representa un aspecto: funcionalidad, coordinación, distribución, composicionalidad, evolución por roles y evolución por aprendizaje. Cada aspecto se divide en dos niveles: nivel base y metanivel, donde se recogen todos aquellos aspectos modificables de los objetos representados en el nivel base. La *Figura 1*, muestra una visión gráfica del modelo de agentes según sus aspectos.

Los aspectos que recoge el modelo son:

- *Funcionalidad***:** Ajustado a la propuesta dada en [Gir01] y concretada con el tratamiento de la reflexión que se usa en AFTER, herramienta descrita en [Car99].
- *Coordinación***:** Sigue la teoría de los contratos expuesta por [And99] llevada a OASIS en [Fer99].

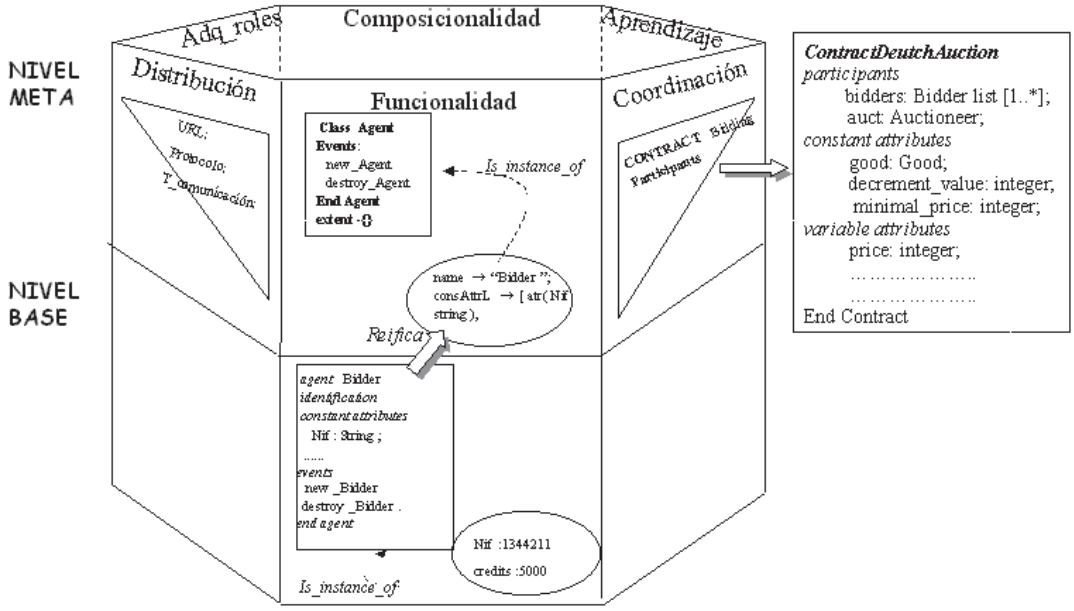


Figura 1: Modelo de agentes: Representación de los distintos aspectos

- *Distribución***:** También expresada sólo en el nivel superior bajo la arquitectura expuesta en [Cor01].
- *Aprendizaje***:** Está ligado al propio concepto de evolución por versionado. Es decir, el agente adquiere nuevas propiedades y su nueva definición es la que conserva la aplicación, sin necesidad de mantener la especificación anterior.
- *Adquisición de nuevos roles*: Trata de evolución temporal. Al igual que en el aprendizaje, el agente cambia sus propiedades en una escena determinada pero puede volver en cualquier momento a la especificación dada para ese agente en otra escena.

Finalmente concretar que cada agente está formado por la agregación de dos sociedades de objetos:

- El *meta_agente*: Metaobjetos que al agregarse, de acuerdo con la semántica de la agregación inclusiva, construye un metaobjeto que ofrece la funcionalidad de cambio agregada de todos sus aspectos (composición, distribución, interacción, funcionalidad y evolución).
- El *agente_base* (agente): Los objetos del nivel base, que presentan todos los aspectos del modelo y que tienen una funcionalidad, son distribuidos, pueden aprender, interaccionar, componerse, y ser utilizados bajo nuevos roles en distintas escenas.

Esta sociedad agregada de objetos es la realización en OASIS 3.0 del modelo de agentes bajo los aspectos representados gráficamente en la *Figura 1*. Establecidas las extensiones del lenguaje para expresar todos los aspectos tratados, un compilador de modelos traduce de forma automática a la tecnología adecuada cada uno de ellos.

3.2 Dinámica en el modelo de agentes

Con el modelo OASIS, la evolución del esquema no trata únicamente las modificaciones en el esquema conceptual que define el sistema multiagente, también trata la migración de las poblaciones existentes para que sean compatibles con el nuevo esquema [Car99].

El problema de la evolución se ha abordado en OASIS de una forma reflexiva incorporando al modelo el metanivel. De esta forma se obtiene un modelo con reflexión de comportamiento que permite al diseñador definir, validar y modificar sistemas orientados a objeto aplicable a la evolución de esquemas multiagente.

Un esquema de agentes es reificado como el estado de una metaentidad. Esta metaentidad se compone de otras metaentidades, las cuales representa cualquier aspecto del modelo de agentes. El estado de las metaentidades define la estructura y el comportamiento de todas las posibles entidades derivadas de ella: su población. Estas propiedades pueden modificarse gracias a los servicios que ofrece la metaentidad en su interfaz.

Las metaentidades ofrecen una doble visión: por una parte son entidades con estado local que ofrecen una serie de servicios para modificar el tipo que definen. Por otra parte son entidades que pueden crear y destruir sus entidades asociadas.

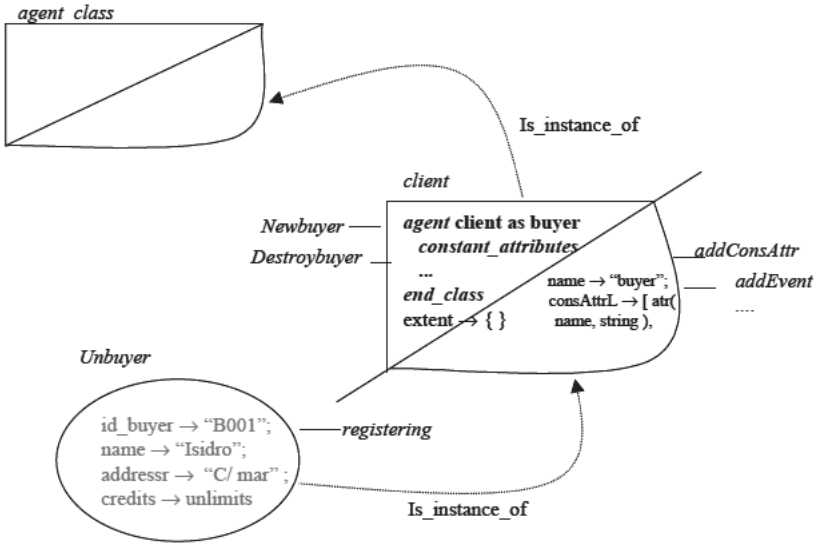


Figura 2: Doble visión del agente *Comprador*

Asociado a los servicios que modifican el estado de la mentaentidad (propiedades del tipo), se define la semántica asociada al cambio que representa incluyendo la evolución de su población.

Para cerrar el potencial ciclo ante la existencia de una jerarquía infinita de metaentidades, el cierre transitivo del metanivel se construye indicando que la metaentidad *Agent_Class* es instancia de sí misma.

La *Figura 2* muestra la doble visión de clase y objeto que presenta el agente *Comprador:*

- La entidad muestra su faceta de objeto al ser instancia de la clase *agent_class*, además de tener una serie de atributos (*name, consAtrrL ...)* y ofrecer servicios que modifican el estado de dichos atributos (*addConsAtrr, addEvent...).*
- La faceta de clase la muestra al tener objetos que son instancia de ella (*Unbuyer...*), atributos propios de clase (*extent...*) y eventos de clase (*Newbuyer, Destroybuyer...*).

3.3 Roles

Podemos definir al sistema multiagente como una institución compuesta por varias escenas. En cada una de estas escenas se lleva a cabo una funcionalidad determinada por parte de un conjunto de agentes que desempeñan un papel bien definido. El orden en el que se pueden dar las diferentes escenas viene establecido en la estructura preformativa (diagrama de transición entre escenas) de la institución [Nor97].

El rol describe una vista parcial del comportamiento de un agente y se constituye como la abstracción básica que le permite adaptarse para participar en diferentes escenas.

El comportamiento de un agente varía en función del escenario donde se encuentre, por tanto un mismo agente puede ofertar diferentes funcionalidades a lo largo de su existencia. Esto indica que las propiedades globales de un agente vendrán definidas como una proyección de las propiedades individuales de los roles que puede jugar. En la entidad que representa el tipo del agente, se definirá el ciclo de vida que pueden seguir los agentes de dicho tipo respetando la estructura performativa de la institución, es decir, en el tipo se define la secuencia posible de roles que puede jugar un agente a lo largo de su existencia.[Wie95]

El modelo de evolución de OASIS permite tratar el dinamismo adjunto a los distintos roles que puede jugar un agente. Un cambio de estado en el entorno, o la llegada de un mensaje, puede hacer que el agente adquiera una responsabilidad motivada por sus objetivos. En dicho caso, el agente activará un evento en su *meta-agente* del cual es instancia, que lo hará migrar hacia el rol capaz de tratar la responsabilidad adquirida.

En la *Figura 3*, se muestra un ejemplo de evolución horizontal de roles. En concreto se indica cómo un agente de tipo *Client*, juega diversos roles en momentos diferentes, pasando de ser un *Buyer* a *Bidder* y finalmente a *Payer*. Los acontecimientos que le harán pasar de jugar un rol a otro serán *open_acution* y *close_auction* (en caso de que haya pujado).

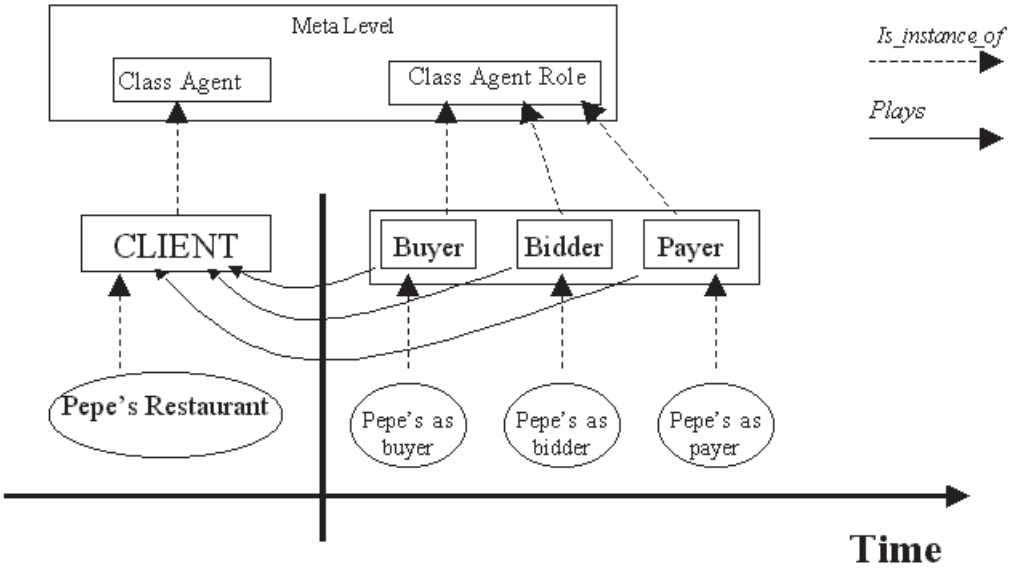


Figura 3: Evolución horizontal de roles

3.4 Aprendizaje

Una de las nociones mentales que poseen los agentes, es la capacidad de aprender, es decir, modificar sus propiedades dinámicamente. El tratamiento del aprendizaje se aborda de una forma sencilla mediante la utilización de la reflexividad que ofrece el modelo OASIS. Para que un agente pueda modificar sus propiedades, deberá cambiar el estado de la entidad agente que representa su tipo (*meta-agente*).

Cuando un agente desea modificar el estado de su *meta-agente* debe tener en cuenta si quiere que el resto de población de dicho *meta-agente*, también modifique sus propiedades y evolucionen

adquiriendo el aprendizaje indicado. En caso negativo, el agente creará un nuevo *meta-agente*, en la mayoría de los casos una copia (especialización) del anterior, con las nuevas propiedades aprendidas.

4 Evolución en el modelo de interacción: Dinamismo en el aspecto de coordinación

4.1 Modelo de interacción

Tal como se describe en [And99] y [Fer98], la definición explícita de contratos entre agentes, permite especificar sistemas fácilmente escalables y modificables. Básicamente el contrato se define como una entidad que relaciona a varios participantes y supervisa su coordinación mediante reglas del negocio, de una forma totalmente transparente. Otra tarea a realizar por un contrato es la gestión de las posibles excepciones que pueden acontecer durante el transcurso de una interacción. Con todo ello se consigue separar los conceptos propios del dominio a nivel agente de las reglas del negocio que regulan su comportamiento.

El modelo de interacción de un sistema multiagente puede llegar a ser más complejo que el de una sociedad de objetos, ya que normalmente el objetivo de una interacción no es únicamente el pase de un mensaje y la gestión de posibles excepciones. La finalidad de una interacción entre agentes es la obtención de un objetivo, y ello suele suponer el llegar a un acuerdo entre los participantes mediante previa negociación.

La *Figura 4* muestra un ejemplo de codificación en OASIS de un contrato. En concreto, se presenta parte del contrato de puja alemana (*DutchAuction*), el cual coordina la interacción entre los pujadores (*bidders*) y el subastador (*auctioneer*). Como se aprecia, el contrato es una clase más que puede tener atributos (*good, decrement_value...)* y eventos que modifican dichos atributos (*adjudicate, decrement...*), pero el apartado más relevante es el de *protocols*, donde se indica la posible sucesión de eventos que pueden acaecer entre los participantes de una subasta alemana.

```
contract DutchAuction
participants
   bidders: Bidder list [1..*];
   auct: Auctioneer;

constant attributes
   good: Good;
   decrement_value: integer;
   minimal_price: integer;

variable attributes
   price: integer;

events
    new
    open_auction(g:Good,dv:int,mp:int
    eger);
   destroy adjudicate;
   destroy close;
   decrement;

valuations
    [decrement]  price := price -
    decrement_value;
    [bidder:auctioneer::enter]
    bidders :=
    insert(bidders,bidder);
…

trigger
   close when {price <=
   minimal_price};

protocols
    open:
    OPEN =
    auctioneer:Buyer::open_auction.(E
    NTER + CLOSE);
    ENTER =
    bidder:auctioneer::enter.(ENTER +
    BID);
    BID: (bidder:auctioneer::bid.
    auctioneer:bidder::adjudicate) +
    DECREMENT;
    DECREMENT: close or
    auctioneer:bidders::offer_decreme
    nt & bid;

    end_contract
```

Figura 4: Contrato entre pujador y subastador usando el protocolo de subasta alemana

4.2 Dinamismo en el modelo de interacción

A lo largo del *apartado 3* ya se indica el tratamiento de la evolución de las entidades propias de dominios volátiles. El inconveniente que se presenta ante dicha solución, es que la variación en las propiedades de una interacción implica la modificación de sus agentes participantes. Este inconveniente

se resuelve fácilmente con el modelo de interacción con contratos, ya que la evolución de un contrato es transparente a sus participantes.

De nuevo, el tratamiento del dinamismo de las interacciones entre agentes se resuelve de forma reflexiva: Ante la modificación de una propiedad de una interacción, el analista activará los correspondientes servicios del metanivel solamente en los aspectos de coordinación y distribución que modifican el estado de la metaentidad que representa el contrato a modificar, de esta forma los agentes implicados no se ven afectados.

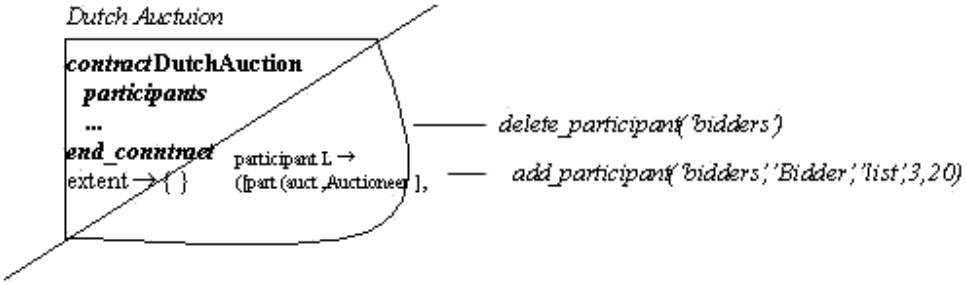


Figura 5: Modificación en la cardinalidad máxima de pujadores

En la *Figura 5*, se muestran los eventos a lanzar para modificar la cardinalidad de los posibles pujadores que participan en la subasta de un lote. Como se puede apreciar en el nuevo contrato, para que se lleve a cabo la subasta, deben de existir como mínimo tres pujadores y como máximo veinte. Los eventos *delete_participant* y *add_participant* están definidos en la signatura de la metaentidad *Contrato*. Su activación modifica el estado de un tipo de contrato, en concreto, elimina o añade participantes, indicando el tipo.

5 Movilidad en los agentes: Dinamismo en el aspecto de distribución

5.1 Modelo de distribución

Siguiendo la misma filosofía que otras aproximaciones [Cor01], mediante el uso de la reflexión, la creación de agentes y la manipulación de su dirección física es responsabilidad del nivel superior usando arquitecturas y protocolos estándar. Aunque el sistema sea distribuido, la solución propuesta debe ser independiente de la estructura final del sistema, ya que la aplicación debe funcionar bajo topologías estructurales diferentes. Tratar la localización de los agentes de forma independiente a sus otros aspectos permite asegurar un marco de trabajo en el cual el desarrollo de la aplicación no esté determinado por la ubicación física de las partes del sistema, incluso se ofrece la posibilidad de ejecutar el mismo prototipo sobre topologías diferentes pudiendo evaluar la más apta para el cumplimiento de los requisitos establecidos.

En [Car99] se muestra a las entidades del metanivel como fábricas de instancias. Por tanto, serán estas entidades las que almacenen la información asociada a la distribución de su población, añadiendo semántica adicional a la relación *is_instance_of*. Las entidades que consulten esta información serán las definidas en el modelo de interacción: los *contratos*.

En el ejemplo propuesto, el agente *Buyer*, almacenará información sobre la distribución de los diferentes compradores que participan en la subasta. En caso de que el *subastador* adjudique un lote a un *comprador* la comunicación se llevará a cabo tal como muestra la *Figura 6:*

(1)(2) Deberá comunicar la adjudicación a su meta-agente (*Auctioneer*). Dicha entidad enviará el mensaje al meta-agente del *comprador* (*Buyer*)[1],

[1] El proceso incluye un nivel más de reflexión, ya que debe existir una meta-entidad que conozca la distribución de las meta-entidades del sistema.

(3)(4)(5)(6) Esta llamada será interceptada por el contrato bajo cuyas reglas debe ocurrir la adjudicación (protocolo de la subasta alemana). Si la ejecución del contrato termina sin excepciones dicha venta es correcta y

(7)(8) El meta_agente comprador, que conoce la ubicación del comprador final le comunicará la entrega, reenviándole de nuevo el mensaje.

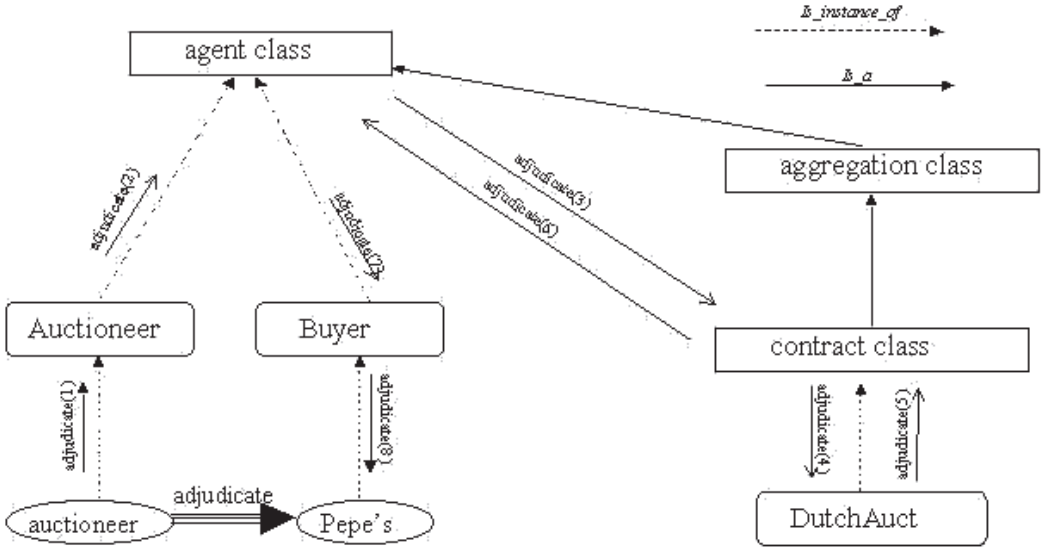


Figura 6: Tratamiento de la distribución usando reflexividad

5.2 Gestión de agentes móviles

Los lenguajes de definición de sistemas multiagente distribuidos, deben contemplar la posibilidad de la existencia de agentes móviles. Esto indica que el modelo debe tratar la posibilidad de que un agente se ejecute en ubicaciones diferentes a lo largo de su existencia.

En OASIS, su modelo de distribución permite tratar la modificación de localizaciones de agentes utilizando de nuevo la reflexividad. Para modificar o asignar durante su creación, la ubicación de un agente se activará el evento correspondiente para que las metaentidades que contemplan dicha información modifiquen su estado.

6 Arquitecturas dinámicas: Dinamismo en el aspecto de composicionalidad

6.1 Modelo de componentes / agentes

En [Lor01] OASIS se presenta como un lenguaje de definición de arquitecturas basado en un modelo constituido por las siguientes entidades:

- *Componente*: Cada componente es un artefacto software que provee una funcionalidad bien definida.
- *Contrato*: Un contrato es un conector software que relaciona componentes y define las propiedades de las interacciones entre ellos.

  Los componentes en OASIS pueden ser de dos tipos:
- *Componentes Elementales*: Representan los roles de los participantes y su evolución a otros nuevos junto con la posibilidad de extraer las propiedades de un patrón genérico. El rol describe una vista parcial de comportamiento del componente y se constituye como la abstracción básica que le permite adaptarse para participar en diferentes composiciones.
- *Componentes Actividad*: En su granularidad más fina, emergen por la agregación de componentes elementales y un contrato, que recoge la interacción de dos o más participantes al realizar una actividad.

En un nivel de mayor granularidad, los componentes entran en un proceso de composición para formar a la vez componentes que ofrecen una funcionalidad mayor.

La arquitectura software asociada a un sistema vendrá definida por un *componente actividad* (*configuración*) que ofrezca la funcionalidad global del sistema. Los mismos mecanismos de composición de componentes pueden traducirse para la composición de agentes.

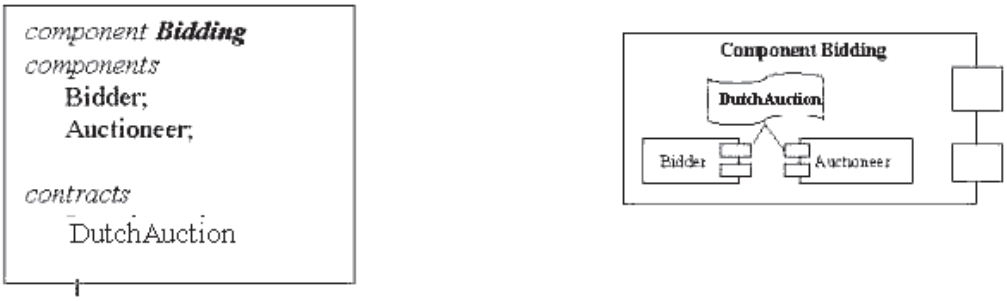


Figura 7: Componente *Bidding*

Como se aprecia en la *Figura 7*, el componente *Bidding,* ofrecerá la funcionalidad de la escena de puja, es decir, será la agregación de funcionalidad de las componentes *Bidder* y *Auctioneer* utilizando para negociar las propiedades del conector *DutchAuction*.

6.2 Arquitecturas dinámicas

Un problema asociado a los lenguajes de definición de arquitecturas, es que deben de tratar la evolución de las arquitecturas que definen, tanto a nivel de componentes como de conectores.

Al igual que en [Loq00], de nuevo, el tratamiento de la evolución de arquitecturas en OASIS, se trata en el metanivel, ya que tanto los *componentes* como los *contratos* reifican *metaentidades* que ofrecerán los servicios necesarios para modificar sus propiedades, indicando la semántica asociada a la migración de sus poblaciones.

La modificación o ampliación de la arquitectura de un sistema constituye un tipo de evolución totalmente ortogonal a los descritos en el *apartado 3*, ya que la modificación o inclusión en el esquema de *componentes* y/o *contratos*, no afecta a las propiedades definidas para los agentes o clases que constituyen dichos componentes o interactúan mediante dichos contratos.

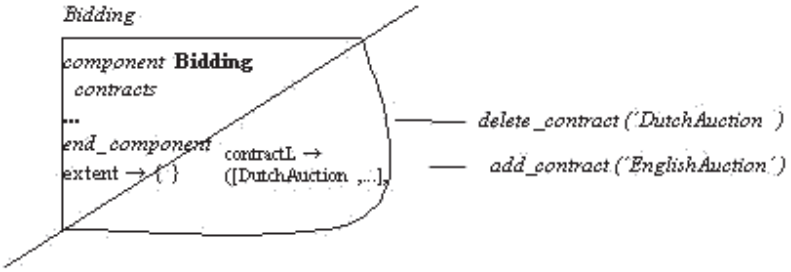


Figura 8: Modificación de protocolo de puja alemana a inglesa

En la *Figura 8*, se muestra los eventos a lanzar para modificar el tipo de contrato a utilizar entre los componentes que forman la componente *Bidding*. Como se puede apreciar, en la escena de puja (*Bidding*) ya no se utilizará el protocolo de puja a la baja (*DutchAuction*), si no que se utilizará la puja a la alta (*EnglishAuction*). Los eventos *delete_contract* y *add_contract* están definidos en la signatura de la metaclase *component*. Su activación modifica el estado de un tipo de componente, en concreto, eliminan o añaden tipos de conectores utilizados entre sus subcomponentes.

7 Conclusiones

El diseño orientado a agentes es una actividad compleja que maneja módulos software activos y proactivos que deben incluir el tratamiento del conocimiento y aspectos de aprendizaje. Existen diversos formalismos para el modelado de sistemas multiagente, pero el modelo que ofrece OASIS va más allá, ya que permite tratar de una forma sencilla el aspecto de dinamismo y de evolución presente en una sociedad de agentes, su arquitectura definida y movilidad.

Gracias a la reflexividad existente en el modelo, la evolución del esquema y su arquitectura puede tratarse dinámicamente y en tiempo de explotación. Esto es debido a que la reflexión ofrece una serie de servicios que permite la modificación de las propiedades de las entidades que pueblan el sistema de agentes, ya sean clases de objetos, tipos de agente, tipos de contrato o tipos de conectores y componentes.

Otro aspecto interesante que contempla el modelo es el de la evolución que existe de una forma implícita en la noción mental de aprendizaje que poseen los agentes. Es decir, un agente podrá modificar sus propiedades en función de los conocimientos que vaya adquiriendo del entorno.

Las metaentidades contemplan información sobre la ubicación de sus entidades, y serán las responsables de la gestión de la localización de sus instancias. Así pues, el dinamismo en la distribución (agentes móviles) queda tratado a este nivel.

Resumiendo, cabe indicar que un agente software se debe modelar desde diferentes aspectos: funcional, de coordinación, de composicionalidad y de distribución. Todos estos aspectos se pueden especificar utilizando un lenguaje común: OASIS. Pero la propuesta va más allá, ya que el mismo lenguaje y mediante la utilización de la reflexión, puede tratar otro aspecto: el de la dinámica del sistema. Donde dicho aspecto de evolución será tratado de forma homogénea al resto de aspectos.

Referencias

[And99] Andrade L., Fiadeiro J.: "Interconnecting Objects via Contracts", in UML'99 –Beyond the Standard, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583..

[Car99] Carsí J. A.: "OASIS como marco conceptual para la evolución del software". Ph.D. Thesis. Universidad Politécnica de Valencia 1999.

[Cor01] Correas J., Bueno F, " A Configuration Framework to Develop and Deploy Distributed Logic Applications". ICLP'01, CLPSE'01 (Constraint) Logic Programming of Software Engineering. 1/12/01. Chipre.

[Cue00] Carlos E. Cuesta, Pablo de la Fuente, y Manuel Barrio-Solórzano. *Arquitecturas del Software Dinámicas Basadas en Reflexión*. V Jornadas Ingeniería del Software y Bases de Datos. Pag 203-214. Valladolid, Noviembre 2000

[Fer99] Ferrer J. "Evolución de OASIS/OO-METHOD hacia el diseño por contratos". Trabajo de Investigación DSIC'99. Universidad Politécnica de Valencia 1999.

[Gir01] Giret A., Ramos I.; "Full software agents as OASIS 3.0 objects". ICLP'01, CLPSE'01 (Constraint) Logic Programming of Software Engineering. 1/12/01. Chipre.

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "Un enfoque formal para el modelado conceptual orientado a objeto". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721-663-0, 1998.

[Loq00] Loques, O., Sztajnberg, A., Leite, J. and Lobosco, M., "On the Integration of Meta-Level Programming and Configuration Programming", In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[Lor01] Lorenzo A, Ramos I., Carsí J.A, Ferrer J.: "Hacia un modelo de Arquitectura Software: Un caso de estudio".I Jornadas Dolmen pag 1- 10, Sevilla 2001.

[Luc95] Luck M., d'Inverno M. :"A formal framework for agency and autonomy. En *Proceedings of the First International Conference on Multi-Agent Systems*, 254-260. AAAI Press / MIT Press, 1995.

[Nor97] Noriega P. "Agent mediated auctions: The Fishmarket Metaphor". Ph.D. Thesis. Universitat Autònoma de Barcelona. 1997.

[Wie95] Wieringa R. et al, "Using Dynamic Classes and Role Classes to Model Object Migration", in theory and Practice of Object Systems, Vol.1, Ed. John Wiley and Sons, 1995

A Conceptual Model for Context-Aware Dynamic Architectures

Javier Jaén Martínez, Isidro Ramos Salavert
{fjaen,iramos}@dsic.upv.es
Departamento de Sistemas Informaticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n 46002 Valencia SPAIN

Abstract

Dealing with architectural complexity and software evolution imposes new challenges to traditional software development methodologies. One of the key challenges is capturing the context where applications exist so that effective adaptive strategies can be implemented. However, existing architectural approaches for software adaptation perform context acquisition and analysis in an ad hoc way, hard-wiring those specifications in any part of a system's definition. In this paper we present a conceptual model for context-awareness that can be integrated into existing architectural approaches for auto-adaptive systems. Managing context as a separate concern or aspect allows for context models reutilization in different architectural definitions and what is more important allows different implementation infrastructures to be used for context awareness.

1. Introduction

Component based software development has been in recent years a very promising research field. The main goal of this new kind of systems is to achieve software reusability at a very high level of abstraction and, as a result, reduce development costs and time. However, implementing component-based software from scratch is not an easy task because systems are architecturally complex, user requirements evolve during the development process, and the "context" where these systems will be deployed is not always taken into account.

Dealing with architectonic complexity and changing user requirements imposes new challenges to traditional software development methodologies which do not cope with such new software artifacts known as components. The main problem that remains unsolved is the definition of software construction frameworks: methodologies, specification languages, and automatic code generation tools that are dynamic in nature and are able to handle evolving component-based architectures. Software evolution in this sense must be understood as the process of changing the specification (structure or behavior) of a given component or changing the architectural layout of a collection of interconnected components to cope with new user requirements.

One could say, first, that this is a ***requirements-driven*** software evolution where changing user requirements is the key factor to be taken into account and, second, that this is a development-time process. However this is not the only type of software evolution that component-based systems require. We will argue in this paper that there is a wide number of modern applications where changes are not driven by new user requirements at development time but rather as a result of analyzing the "context" where applications are actually run. We call this type of dynamic behavior ***context-driven*** software evolution and should take place with little user intervention at execution time.

In both cases, existing architectural approaches for software adaptation are based on a separation of concerns: computation and coordination as presented in [1] where coordination contexts may be defined to achieve evolution; or functional, coordination and evolution aspects as defined in [2] where a metalevel is required similar to the one proposed in [3]. However, a great limitation of these approaches is that no matter what architectural model is selected, context acquisition and analysis is always performed and expressed in an ad hoc way, hard-wiring those specifications in any part of a system's definition.

Our claim is that a new concern to deal with context-awareness must be explicitly defined in every approach that deals with architectonic auto-adaptation. By doing so, software components as defined in [4], and even connectors and systems as presented in [1, 2] will be able to explicitly gather context information, automatically perform context-analysis tasks and even notify events of interest. Managing context as a separate concern or aspect allows for reutilization of context models in different architectural definitions and what is more important allows different implementation infrastructures to be used

Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)
0-7695-1921-0/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

Jaén, J., Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures. Proc. of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03). 2003.

for context awareness: monitoring services, probes, or repositories for context acquisition; neural networks or rule-based systems for context analysis; and distributed event services for context notification.

The remaining sections are structured as follows: section 2 illustrates the need for "context-awareness" by example and introduces a new category of context-aware applications; section 3 defines the components of our conceptual model and finally section 4 concludes and describes our immediate future work.

2. Why Context-Awareness?

Context awareness is not a new idea. A detailed study of context aware applications described in [5] divides such applications in three categories. The first category includes applications that present context dependent information and services to the user. The second category are applications that start services automatically when certain contextual conditions are met. The third category involves applications that tag information with some description of context (e.g. the context in which the information was obtained). Examples of applications in the previous categories are

CyberGuides [6] are applications that use some type of context awareness (usually location) to offer more accurate and appropriate information than that traditionally offered by guidebooks.

Grid Applications are computing or data intensive applications [8, 9] that make use of resources existing in multiple organisations that constitute together a "virtual organisation". Efficient task partitioning, assignment and control can only be achieved by having a good understanding of the existing conditions: available resources, execution progress of subtasks. Existing architectural approaches make use of software sensors to capture such information and analyse the "context". These are however defined at the implementation level.

Web Services-based Applications [10] are built by gluing together distributed services that speak standard protocols and exchange information represented using standard formats. Future applications build under this new concept will have to be aware of the existing services and their status to dynamically change their configuration by adding, replacing, and/or replicating services in order to accommodate for the best performance or satisfy some specified quality requirements.

Virtual Rooms Videoconferencing System (VRVS) [11] is a distributed and efficient networked software application for multi-point audio, video and web conferencing and collaboration over regional, national, and global IP networks. In this type of application the runtime topology of network servers is a critical factor to guarantee real-time multi-point videoconferencing. This topology has to be automatically and dynamically updated based on contextual information about the participants, the network conditions, and the nature (audio, video/audio, video/audio/data) of the videoconferences.

Mediacup, SmartDoorplate, HotClock, CoffeePump [12] these examples are focused on augmenting everyday objects with context awareness so that situations like "there is no coffee" or "cold coffee", or if a user is drinking from the cup or if a set of cups are close together can be detected so that this information can be used to start automatically contextual activities.

According to the three categories enumerated above these applications can be classified as follows:

Presentation of information and services (PIS)
- CyberGuides

Automatic execution of services
- Mediacup, SmartDoorplate, HotClock, …

Tagging information with context
- None

It seems obvious that a new generation of application have arisen and do not fit well into the previous categorisation for traditional context-aware applications. These are known as **self-configurable or auto-adaptive** applications because they perform architectural evolution based on contextual information.

Achieving context awareness for this new kind of applications could be done either by including code arbitrarily in any part of an application or by using toolkits that implement such functionality [13]. On the one hand, ad hoc code insertion makes it difficult to reuse context-gathering processes and what is more important, makes it impossible to reuse the underlying context models. On the other hand, existing toolkits for context representation do have a reusable context model but impose specific communication schemas among components and implement specific persistence mechanisms limiting the technological choices. To overcome these problems a conceptual framework that proposes a context model and is at the same time technology-independent is needed.

3. A Conceptual Model for Context-Aware Dynamic Architectures

For successfully constructing applications such as the ones presented in the previous section it is fundamental to understand what context means and what its dimensions

IEEE COMPUTER SOCIETY

and characteristics are. Many definitions have been given in the past to define context:

"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." ([5])

"Circumstance, situation, phase, position, posture, attitude, place, point; terms; regime; footing, standing, status, occasion, surroundings, environment, location, dependence". (Source: www.thesaurus.com)

"Discourse that surrounds a language unit and helps to determine its interpretation [syn: linguistic context, context of use] 2: the set of facts or circumstances that surround a situation or event; "the historical context" (Source: WordNet ® 1.6)

"knowledge about the user's and IT device's state, including surroundings, situation, and, to a lesser extent, location". ([14])

From the list shown above it can be concluded that there are too many definitions about context which are not always unambiguous. Fortunately, a more structured approach for modeling context is presented in [5] where four types of primary context information have been identified: Location, Identity, Time and Activity; in [15] where a three dimensional context model is defined: Environment, Self and Activity and also in [16] where a top-down hierarchical view is given starting at the top level with a separation between Human Factors and the Physical Environment and then splitting them up into sub-categories: User, Social Environment, Users Task; and Conditions, Infrastructure and Location respectively. These are categorical or data-oriented views of context-awareness. However, our approach builds on a process oriented view of context-awareness. The reason for this is that we conceive software systems as societies of autonomous interconnected entities (components, connectors) [1][2] . Such entities are defined in terms of orthogonal behavioral dimensions or aspects to cope with their assigned responsibilities (functional, distribution, coordination, quality and evolution). In our approach, context-awareness is an additional behavioral aspect or process that any entity (component, connector, or subsystem) may exhibit.

Having this process oriented view in mind we have identified a number of tasks that eventually may be performed by context-aware entities:

Context Provision: The basic building blocks of any context-aware system are information providers. These not only provide context-information but also must support inspection in order to retrieve the structural properties of the provided information.

Context acquisition as the process of gathering and storing information from providers. Conceptual artifacts for context acquisition have properties like the sampling period and the caching size.

Context aggregation as the process of joining information that was acquired from different sources to offer a new type of correlated information. Aggregation may be done following different strategies which may be very application specific, therefore, these will be defined at specification time.

Context analysis as the process of categorizing situations that may result after some analysis procedure has been carried out on the acquired/aggregated information. Such categorization process may be as simple as a "gauge" establishing whether some information is within some bounds or as complex as a recursive associative Neural Network that clusters incoming training data. To cope with such heterogeneity context analysis artifacts will provide mechanisms to establish/retrieve information about classification categories and their corresponding descriptors.

Context notification as the process of delivering raw, aggregated or analyzed context data to subscribers

This process and aspect oriented view of context-awareness can be expressed conceptually as a UML metamodel, (see [2] for a detailed description of the remaining aspects) in which any entity (component, connector or subsystem) exhibiting context-awareness must give specifications for the processes listed above in terms of five interfaces: ISource, IGatherer, IAggregator, IAnalyzer and INotifier.

A trivial but illustrating example is shown in Figure 2. A context-aware component *GridComputingNode* (GCN) gathers context information such as the CPU load, number of waiting processes, and available memory on a Grid computing node. Therefore its context-aware aspect would give an implementation- independent specification for the IGatherer interface. A connector obtaining such information from multiple nodes, could aggregate and analyze it in order to detect certain context categories like "Cluster Overloaded", "Node n collapsed", "Node n empty", etc. Such a connector would include within its Context-Awareness aspect both an specification for both the IAggregator and the IAnalyzer interfaces. Note that having such a type of connector defined at a high level of abstraction allows for reutilization, e.g. a LoadBalancer (LB) component could be connected to this connector so

that adaptive actions like removing a CPU consuming job or addressing a job to a free node can be performed but also the same connector could be used by a ClusterManager (CM) component in order to automatically add/remove nodes from the cluster. The connections between concrete components and connectors to form a specific configuration are initially established using a Configuration Language [2] but a detailed description is outside of the scope of this paper.

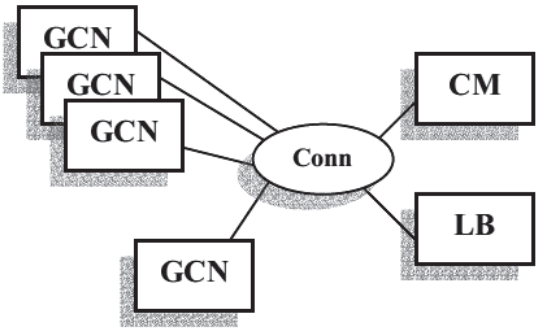


Figure 2. Context-aware components and connectors

For our model to be complete and not a mere conceptual artifact a definition for each of the proposed interfaces must be given. We have proposed a preliminary definition for them but we are aware that more specialized interfaces may be derived for specific applications. These interfaces should be taken as the basic conceptual framework from which architects may define new specific services.

The behavior associated to the methods in such interfaces must be specified by each concrete component or connector using a high-level language. In our case, artifacts of the Context-Aware Aspect of every PRISMA[1] component or connector are specified in terms of OASIS specifications [17]. Once these specifications are obtained, automatic code generation techniques may be applied [18] to translate every context-aware aspect specification into a concrete implementation platform (J2EE Enterprise Java Beans, CORBA components, .NET Services, etc).

3.1. Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) [19] are key elements of our model. These are short strings that have been traditionally used to identify resources in the web: documents, images, downloadable files, services, electronic mailboxes, and other resources. However URIs are in our model a naming schema for components, connectors, sources of context information, etc. The naming scheme is as follows:

```
prisma://<instance_name>/<aspect>/<interface>
```

3.2. ISource

This is the most elementary interface of our conceptual model. Any entity that delivers a continuous stream of information must give an specification for it. No caching nor any other elaborated service is provided. The most important services to be considered in this interface are listed below and most of them are self-explanatory. Only the services related to schema management need a special attention. Obtaining information in some format would be useless if there are no mechanisms that describe the structural properties of such information. For this reason we have associated schemas (in a conceptual sense) to every source of information. Concrete examples would be an XML source of context information with its associated XML schema. We assume that every schema provides the appropriate decoding/encoding mechanisms.

It is also important to point out that the ISource interface does not impose any implementation choice and consequently once could define a PRISMA component whose specification would access a remote Grid Monitoring Service obtaining information in XML format, and another component that would access a local repository with a proprietary format. The choice of a particular infrastructure is done at specification time.

```
Object getMeasure()
URI getSchemaURI()
setSchemaURI(URI schema_uri)
```

3.3. IGatherer

Another primary process to be carried out by context-aware PRISMA elements is gathering context information from sources at a given rate (sampling period) and caching the corresponding information. A PRISMA element sampling and caching data from a data source must provide a specification for the IGatherer interface. Such interface may be used by other PRISMA entities (connectors, components, systems) to perform more elaborated activities.

```
- setSamplingPeriod(Long period)
- setCachingSize(Integer size)
- flush()
- Object getNextMeasure()
- Collection getAllMeasures()
- Long getSamplingPeriod()
- Integer getCachingSize()
```

[1] PRISMA is our architectural model and component platform

IEEE COMPUTER SOCIETY

3.4. IAggregator

Aggregating information is a process aiming at elaborating complex information by correlating information provided by other parties. Natural PRISMA elements to perform such processing are connectors. These may connect several components that have an specification for the ISource or IGatherer interface, obtain context data from them, aggregate it following some aggregation process and offer the aggregated information to other parties (components or connectors) by offering an ISource or an IGatherer interface (depending on caching availability). The IAggregator interface will consist of services for controlling the components that participate in the aggregation process. The process itself is application dependent and is defined at specification time.

```
- Collection getComponentsURI()
- addComponent(URI component_uri)
- removeComponent (URI component_uri)
```

3.5. IAnalyzer

Analysis is a critical process of contex-awareness. Analysis processes may be too complex to be specified in a conceptual model but rather be implemented by a neural network, rule based-system, a regression system, etc. Making our model dependant of a specific analysis technique would invalidate our claim that we are at a high abstraction level. We conceived analysis as the process of establishing categories and assigning descriptors to such categories. The number of categories and the associated descriptors may be as complex as required by each concrete application. Each category descriptor has an associated schema to describe the structural properties of the descriptor. Some examples of categories and associated descriptors are: (neurons, adaptation weights) in a Kohonen Map, (input Space, estimation space) in a regression analysis model, (states, probabilities) in a fuzzy model, etc. Consequently, this interface provides mechanisms to obtain the schemas associated to both categories and descriptors and obtain at any time the collection of existing categories and their associated descriptors.

```
- setCategorySchemaURI(URI category_uri ,
URI category_schema_uri)
- setCategoryDescriptorSchemaURI(URI
category_uri, URI
category_descriptor_schema_uri)
- Collection getCategories()
- Object getDescriptor(URI category_uri)
- URI getCategoryDescriptorSchemaURI(URI
category_uri)
- URI getCategorySchemaURI(URI
category_uri)
```

3.6. INotifier

Finally, a key process of every context-aware infrastructure is the asynchronous notification of information. A context-aware PRISMA element may give support to publishers and subscribers to information of interest. This information may be plain gathered data, or aggregated one or more elaborated analysis categories. Again we are not bound to any concrete event-driven infrastructure at specification time. The underlying runtime-infrastructure (which is not the scope of this paper) will support the proposed functionality. The automatic code generation step will bind the specification of the INotifier services to a concrete distributed event service like the ones described in [20],

```
- publish(Event event)
- subscribe(URI listener_uri, Expression
expr)
- unsubscribe(URI listener_uri,
Expression expr)
```

4. Conclusions and Future Work

We have presented in this paper a conceptual model for context-awareness to be used within the scope of auto-adaptive architectures. We aim at providing the right conceptual tools for system architects to define component based architectures which are context-aware. The requirements that we have in mind when defining such a conceptual framework are

Separation of Concerns: this is a fundamental requirement for complex systems that can be modeled from different points of view [21]. Dynamic software architectures may be analyzed in terms of their functional, coordination, distribution, and evolution properties [2]. Considering these different views as separate software development aspects allows for reusability of aspects and complexity reduction. In this paper we have focused on a new separate aspect to deal with context awareness and have defined a conceptual framework that analyzes the different building blocks for context-aware auto-adaptive systems.

Component-Based Software Development: rapid prototyping and cost reduction of the development process are also key factors of our approach. It has been demonstrated [4] that building Software Architectures in terms of interconnected components or services with well known interfaces is a very promising approach when considering the previous factors. Therefore our context model has been defined to be seamlessly integrated into existing component-based conceptual approaches such as the ones presented in [1, 2].

High-level of Abstraction: Modeling context-aware dynamic architectures must be done at high levels of abstraction so that conceptual patterns for context gathering, integration and analysis can be identified and described with high level artifacts and languages. Our model does not impose any formalism for the specification of the services of every interface although OASIS will be the reference formal language when integrating this new aspect into the PRISMA model.

Automatic Code Generation: Having OASIS as a reference language with well known semantics automatic code generation mechanisms can be implemented such as the ones we have already defined in [18]. This strategy allows different technological component based approaches (.NET, Enterprise JavaBeans) to be selected as the target implementation platforms.

Our immediate future work is to build some prototype context-aware PRISMA components and connectors based on this model and using OASIS as the specification language. Then, implement the code generation engines and runtime system to animate our specifications.

5. References

[1] Andrade L., Fiadeiro J.L. "An Architectural Approach to Auto Adaptive Systems" CAiSE 2001, Insbruck Suiza

[2] Pérez J., Ramos I., Jaen J., Letelier P., "*PRISMA: Development of Software Architectures with Aspect Oriented, Reflexive and Dynamic Approach"*, Submitted to DEXA 2003.

[3] McGurren F., Conroy D., "*X ADAPT: An Arquitecture for Dynamic Systems"*, Workshop on Component Oriented Programming, ECOOP, Málaga, Spain, 2002.

[4] Szyperski C. Component Software: Beyond Object Oriented Programming, Addison Wesley 1998

[5] Dey A. K. & Abowd, G. D. (2000a). Towards a better understanding of context and context awareness. Proceedings of the Workshop on the What, Who, Where, When and How of Context Awareness, affiliated with the CHI 2000 Conference on Human Factors in Computer Systems, New York, NY: ACM Press.

[6] Yang, Jie, Weiyi Yang, Matthias Denecke and Alex Waibel (1999). Smart Sight: A Tourist Assistant System. In the *Proceedings of the 3rd IEEE International Symposium on Wearable Computers (ISWC '99)*, pp. 73 78, San Francisco, CA, IEEE. October 18 19, 1999.

[8]Foster I., Kesselman C., Tuecke S., The Anatomy of the Grid. Intl. J. Supercomputer Applications, 15(3), 2001

[9] Hoschek W., Jaen Martinez J., Samar A., Stockinger H., Stockinger K., Data Management in an International Data Grid Project. In Proc. 1st IEEE/ACM International Workshop on Grid Computing, 2000, Springer Verlag

[10] W3C "Web Services Architecture Working Draft", 14 November 2002 http://www.w3.org/TR/2002/WD ws arch 20021114/

[11] The VRVS Video Conferencing System http://www.vrvs.org

[12] Gellersen H., Beigl M., Krull H. "The MediaCup: Awareness Technology embedded in an Everyday Object". 1st Int. Sym. Handheld and Ubiquitous Computing (HUC99) Karlsruhe, Germany, 1999

[13] Anind K. Dey and Gregory D. Abowd "The Context Toolkit: Aiding the Development of Context Aware Applications". *Workshop on Software Engineering for Wearable and Pervasive Computing ,* Limerick, Ireland, June 6, 2000.

[14] Van Laerhoven K., Aidoo K. A., "Teaching Context to Applications", In *Personal Technologies: Situated Interaction and Context Aware Computing*, A. Schmidt, G. Kortuem, D. Morse and A. Dey (Eds). Springer Verlag, Berlin, 2000

[15] Schmidt A., Aidoo K.A., Takaluoma A., Tuomela U., Van Laerhoven K., and Van de Velde W. "Advanced Interaction in Context". In H. Gellersen, editor, Proc. of Intl. workshop on Handheld and Ubiquitous Computing (HUC99), number 1707 in LNCS, Heidelberg, Germany.

[16] Schmidt A., Beigl M., Gellersen H. W., "There is more to context than location", *Proceedings of the International Workshop on Interactive Applications of Mobile Computing (IMC98)*, November 1998, Rostock, Germany

[17] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "*Un enfoque formal para el modelado conceptual orientado a objeto*". Universidad Politécnica de Valencia, SPUPV 98.4011, ISBN 84 7721 663 0, 1998.

[18] Letelier P., Sánchez P., Ramos I. "Prototyping a requirements specification through an automatically generated concurrent logic program". G. Gupta (Ed.) Practical Aspects of Declarative Languages: First International Workshop; PADL'99, LNCS 1551, pp. 31 45, ISBN 3 540 65527 1, Springer Verlag, 1999.

[19] W3C "Naming and addressing: URIs, URLs" http://www.w3.org/Addressing/

[20] Meier R., "State of the Art Review of Distributed Event Models", 2000 ftp://ftp.cs.tcd.ie/pub/tech reports/reports.00/TCD CS 2000 16.pdf

[21] Aspect Oriented Software Development, http://aosd.net

PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures

Jennifer Pérez, Isidro Ramos, Javier Jaén, Patricio Letelier
Department of Information Systems and Computation
Politechnic University of Valencia
Camino de Vera s/n
E-46071 Valencia Spain
{jeperez | iramos | fjaen | letelier}@dsic.upv.es

Elena Navarro
Department of Computer Science
University of Castilla-LaMancha
Avda. España s/n
02071 Valencia Spain
enavarro@info-ab.uclm.es

Abstract

The development of software systems must be done using platforms that allow the description of quality, complex, distributed, dynamic and reusable architectural models. We present in this paper PRISMA, an architectural modelling approach based on aspects and components, that uses a component definition language (components, connectors and systems) to define architectural types at a high abstraction level and a configuration language to design the architecture of software systems. The component definition language increases reuse allowing importation of COTS and reduces complexity by integrating two modern software development approaches: Component-Based Software Development and Aspect-Oriented Software Development. The configuration language designs the architecture of software systems by creating and interconnecting instances of the defined types including possible imported COTS. PRISMA has a metalevel with reflexive properties for these two languages. For this reason, the types of PRISMA may evolve and the topologies of PRISMA may be reconfigured dynamically.

Key Words: Software Architecture, Component Definition Language, Configuration Language, Components, Aspects, Reuse, Evolution, Quality

1. Introduction

Nowadays, information systems are more and more difficult to develop due to their complex structures and dynamic nature. The high competitiveness of the market and the volatile business rules generate continuous changes on the information systems requirements. For these reasons, the development of software systems must be done using platforms that allow the description of quality, complex, distributed, dynamic and reusable architectural models. Moreover, these architectural models must satisfy the quality requirements established by the product and the user.

The object oriented approach cannot account for all the requirements of current and complex information systems. For this reason, two approaches for developing software systems have emerged: Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD).

The high competitiveness increases the relevance of the evolution and reuse in order to minimize the time and the cost invested in the development and maintenance processes. As a consequence, the COTS (commercial off-the-self) importation has acquired relevance, because tools that allow the reuse of its components and the COTS importation achieve the highest reuse and a quality code.

Concerning software evolution, it is important to take into account the two types of software evolution: requirements-driven and context-driven. The first one is the change of the user requiremens and second one is the change of the context where the applications are actually run. If a model can support these two types of software evolution in a simply and natural way, it will improve the quality behavior of its applications.Apart from these quality factors, there are others that should be considered in the development of architectural models in order to improve the architectural systems.

Currently, it is difficult to find an approach that overcome all these issues. For this reason, we present in this paper an approach which achieves to cover them.

Proceedings of the Third International Conference On Quality Software (QSIC'03)
0-7695-2015-4/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. Proc. of the 3rd IEEE International Conference on Quality Software (QSIC 2003). 2003.

This approach is called PRISMA and uses an extension of OASIS [15] to define the semantics of architectural models in a formal way. OASIS is a formal language to define conceptual models of object-oriented information systems. Its extension has been necessary because despite of the fact that it verifies and generates applications automatically from the object-oriented conceptual models (Model compiler), does not allow to specify software architectures. In this way, PRISMA preserves the OASIS advantages that are very useful for validating its architectural models.

In PRISMA two languages are defined: a component definition language and a configuration language. Both languages can be compiled to generate automatically the code of the product following the paradigm of automatic programming [3].The component definition language defines architectural types with a high abstraction level, its main advantages are software reuse allowing importation of COTS and complexity reduction by integrating two modern software development: CBSD and AOSD. The configuration language designs the architecture of software systems by creating and interconnecting instances of the defined types including all the imported COTS. In other words, this language specifies the topology of an specific software system.

Every type of our architectural model consists of a set of aspects: functional, distribution, coordination, quality, context-awareness, etc. This is the PRISMA way of combining CBSD and AOSD. In addition, PRISMA supports the requierements-driven and context-driven evolution. The former is solved by means of a metalevel and reflexive properties for the two presented languages and the latter defining a context-awareness aspect. The PRISMA metalevel allows the evolution types and the dynamic reconfiguration of topologies. Each aspect of a type has a base level and may have a metalevel. The metalevel reifies the properties that may be evolved following the OASIS reflection ([6]). Apart from the quality provided by evolution and reuse, PRISMA takes into account other quality attributes in the quality aspect in order to achieve quality architectural models

The structure of the chapter is the following: first, a brief summary of related work is given. In Section 3 the PRISMA architectural model, the metamodel, and the component definition and configuration languages are presented and described. In section 4 conclusions are shown and finally, future work is presented in section 5.

2. Related Work

A great deal of work has been done so far dealing with architectural models, components, architecture definition languages, dynamic reconfiguration and other aspects related with the implementation of complex systems. Part of the work is focused on finding a consensus about the different concepts and approaches but due to the lack of a semantic commitment in the field, an important formalization effort is required to achieve a reasonable level of integration. We take into account different solutions of interest for our model: Component Based Design and the Separation of Concerns or Aspect Oriented Programming together with a powerful reflexive metalevel. The smooth integration of the considered solutions and the high abstraction level are the main features of PRISMA.

The work developed by Garlan ([10]) has been of great inspiration for us in our effort to define the first order citizens of our model. The main types defined are: systems, connectors, components, in/out ports, in/out roles and interfaces. Also the work by Andrade & Fiadeiro ([And01]) about the notion of contract has been considered making a difference from the one presented in [18] and [5].

Our motivation is similar to the Model-Driven Architecture (MDA) approach proposed by the OMG group ([20]) for system's interoperability and integration. MDA encourages modelling using UML extensions in an independent way of the final implementation platforms. They propose separation of concerns and different abstraction levels in the modelling process. Transformation techniques and inter-level relationships are used in order to produce the final implementation.

A wide variety of Component Definition Languages has been proposed. Each one has positive and negative aspects. An interesting comparison between these languages is done in the work by [17]. The proposal by Loques & Leite ([16]) is very similar to our approach. Their Model R-RIO incorporates dynamic reconfiguration by means of a reflexive architecture but the use of a metalevel is quite different from ours. Other approaches use metaprogramming techniques but at an implementation level such as the work by [9]. In GUARANA [19] a complex metalevel is defined at compilation time in an ad-hoc and independent way of the proposed model.

The Separation of Concerns approach followed in the Aspect Oriented Design methods such as the work by [2] has also been taken into account in our model. But we use a higher abstraction level, namely conceptual design.

We consider that it is very important to provide a graphical notation for the PRISMA languages. There are several works that study the use of UML for the description of software architectures ([10, 14]). All of them agree with the fact that UML (at least 1.4 version) is not the solution because there are types (ports, connectors, etc) that do not have a direct representation. For this reason, there are some works that propose modifications using UML extension mechanisms

IEEE COMPUTER SOCIETY

[18,23]. However, the announced 2.0 UML version could solve these problems. Due to the strong momentum of UML and its support by modern software production tools we consider UML as the first alternative for the graphical representation of PRISMA.

Nowadays, there are different proposals to measure the quality attributes, some of them measure attributes qualitatively and the remaining ones quantitatively. Moreover, some proposals automate some of their activities as Win-Win[11], NOFUN[4], QRL[22]. We decided to select QRL to specify our quality attributes, because it has a precise operational semantics allowing an automatic trading and negotiation process.

Context awareness is not a new idea but it is a very important issue when developing and maintaining self-configurable or auto-adaptative aplications. A detailed study of context aware applications is described in [8] and divides such applications in three categories. The first category includes applications that present context dependent information and services to the user. The second category are applications that start services automatically when certain contextual conditions are met. The third category involves applications that tag information with some description of context (e.g. the context in which the information was obtained).

Concerning the languages supporting the model we will use extensions of our OASIS specification language to cope with the specifications of the different aspects at the base level and the meta-level of the PRISMA Model. Model Compiler Techniques will be used to generate the final implementation code for each aspect.

3. Architectural Model

PRISMA is a model to define architectures of complex software systems. The PRISMA types are defined by composition of aspects: functional, quality, distribution, coordination, etc, and may vary depending on the architectural type and the information system kind under consideration.

The code overlapping [13,21] problem of the AOSD is solved by our model because aspects integration is done with a high abstraction level and code generation for each aspect is done independently. Next we present several aspects that a PRISMA type may include.

Functional Aspect: The functional aspect captures the semantics of the information system by defining its structure and behaviour.

Coordination Aspect: The coordination aspect defines the business rules and the synchronisation between types during its communication. Moreover the choreography or protocol is defined, extending the "contract" concept of Andrade & Fiadeiro ([1]).

Distribution Aspect: The distribution aspect specifies the features that define the dynamic relocation of the instances.

Quality Aspect: The quality aspect specifies the non functional quality requirements. The Quality Requirements Language (QRL) [22] is used to describe the quality attributes of a type. QRL has a precise operational semantics allowing an automatic trading and negotiation process. In this way we include in our PRISMA model the automatic generation of Multi-Organisational Web-based Systems applications.

Context-Awareness Aspect: The context-awareness aspect provides context-information and supports inspection in order to retrieve the structural properties of the provided information.

Evolution Aspect: The evolution aspect specifies the evolution of types due to changes on the information system's requirements. This aspect is based on the concept of OASIS reflection to provide types dynamic behaviour. The reflection allows to change the structure and the behaviour of a type.

A type (component, connector, system) of our architectural model can be seen as a prism. This prism has one side for each aspect. We must take into account that the evolution aspect is not a separate side of the prism but rather is on the top of each side of the prism at the metalevel (see figure 1). Each side will define a metalevel if some types in the side reifies some property of the base level. Any section of the OASIS specification can undergo this reification process giving the languages their reflexive capability.

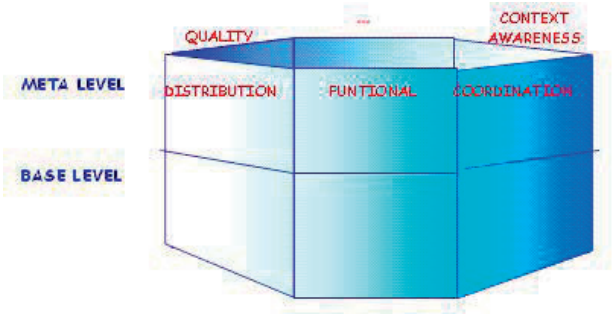


Figure 1. A Prisma Type

The reification is done from the base level. For this reason, it is impossible that a type specifies its metalevel without its base level. The relationship between metalevel and base level within one aspect is 1:N.

Our architectural model consists of different types. All them are in the metamodel of PRISMA (figures 2-6) so that they may be modified by means of the reflection mechanism. Next the different types are described.

Interface: An interface defines a set of services. There are different types of interfaces, one for each type of aspect.

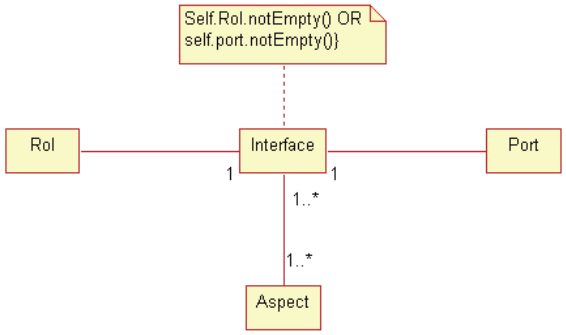


Figure 2. Interfaces section of the PRISMA metamodel

Aspect: An aspect defines completely the structure and the behaviour of an element from a specific point of view. An aspect is the join of a set of interfaces and the semantics specification of its structure and behaviour (state changes, triggers, protocols, etc). There are different types of aspects: functional, distribution, coordination, quality, context-awareness, etc. Each one is defined in an orthogonal way.

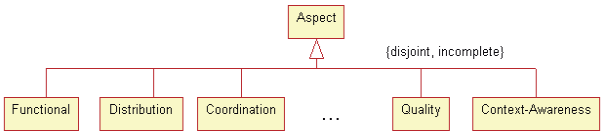


Figure 3. Aspects section of the PRISMA metamodel

Component: A component is a type of the information system which does not act like a coordinator between other elements. It is composed by a set of aspects (functional, distribution, quality, context-awareness, etc) and by one or more **inports** and **outports**, whose type is an specific interface. Components interact with other architectural types by means of their ports.

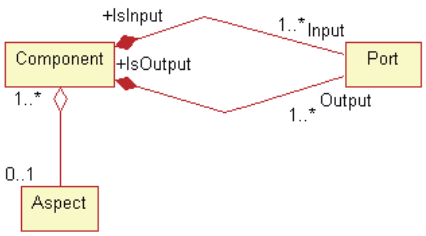


Figure 4. Components section of the PRISMA metamodel

Connector: A connector is a type of the information system which acts as a coordinator between elements. It is composed by aspects (coordination, functional, distribution, quality, context-awareness, etc) and a set of **inroles** and **outroles**, whose type is a specific interface. Connectors link and synchronise components, systems and other connectors by means of their roles.

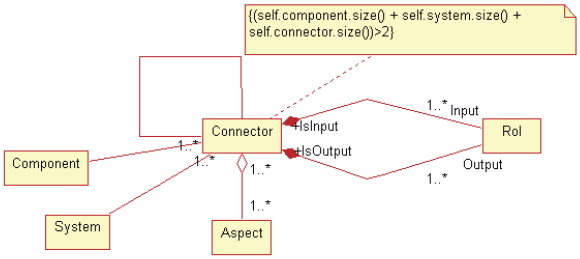


Figure 5. Connectors section of the PRISMA metamodel

System: A system is a component that includes a set of connectors, components and other systems correctly connected. From this composition emergent properties may arise. Moreover, a system defines its bindings between its ports and the ports of the included components.

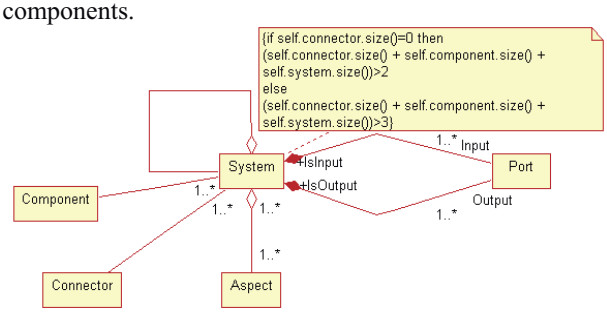


Figure 6. Systems section of the PRISMA metamodel

3.1. Component Definition Language

The component definition language of PRISMA defines the types of the architecture. This language has as first-class constructors: interfaces, aspects, components and connectors. The fact that interfaces and aspects are first-class constructors increases reusability because an interface may be used by several aspects and an aspect may be used by several types of elements. Moreover, components, connectors and systems may be reused in different architectural models.

The model allows for COTS importation because the specification of aspects is optional. In this case, elements of PRISMA are like black boxes with specific quality requirements. Whenever the type is a COTS, the functional aspect is not present and the quality aspect is interpreted as a query. The query process consist on

IEEE COMPUTER SOCIETY

searching and trading in the web components or web services that satisfy the established non-functional quality requirements. At the end of this process, the component is instantiated to the one found maximising the utility criteria specified in the non-functional quality requirements. However, there are COTS that make the valuation of the product's quality difficult because their quality attributes are either not specified in a precise way or not defined at all. This is a disadvantage because no automatic selection and valuation processes of COTS based on these attributes may be implemented.

If the functional aspect is present, i.e. non imported component, the non-functional quality requirements adds quality information to the component. In this case, the quality aspect establishes the quality rating of the type and its instances must satisfy this quality rating in all theirs life cycles.

3.1.1. Interfaces and Aspects. A functional interface specifies a set of services, their parameters and types. An interface describes the partial view that an element has one of the aspect of another element. A functional aspect is specified using the OASIS elemental class template presented in [15]. This template describes attributes, services, valuations, derivations, preconditions, triggers, integrity constrains and the **protocol** of a component.

Next, interface definitions and one functional aspect are presented. In this example we can see the functionality of a component which bids in on an electronical auction. The *Bidder* defines the semantics of *Bidding* and *Maintenance* interfaces and specifies this functionality modifying attributes (constants and variables) by means of valuations and protocols. In the valuations section of the *Bidder* aspect, the effect of executing *ChangeLowPlayer* is described. In the protocols section, the allowed lives of a component instance are specified based on relevant states.

```
Interface Bidding
{
   Bid;
   Pay;
}
```

```
Interface Maintenance
{
   Create;
   Update;
   ChangeLowPlayer(state: boolean);
   Destroy;
}
```

```
Functional Aspect Bidder specifies Bidding,
Maintenance
{

  Constant Attributes
       Identifier: string;
       Name: string;
       SurName: string;

  Variable Attributes
       Address : string;
       VisaNumber : string;
       ExpireVISA: date;
       LowPlayer: boolean;
              DEFAULT false;

  Private Services
       Create;
       Update;
       ChangeLowPlayer(state:boolean);
       Destroy;
       Bid;
       Pay;

  Valuations
       [ChangesLowPlayer(state)] LowPlayer
       estate;

  Protocols
    CREATION   create.BIDDER;
    BIDDER   update +
           ChangeLowPlayer.LOWPLAYER + bid
           + pay + destroy;
    LOWPLAYER   update +
                ChangeLowPlayer.BIDDER +
                destroy;
}
```

In the component, distribution aspect specifies services to obtain metrics values in order to implement complex distribution strategies as the ones shown in [24, 7]. In connectors and systems, it defines the distribution policies and strategies to be applied at run time following the gathered measures of the established metrics.

The specification of co-ordination interfaces is very similar to that proposed for other types of interfaces. A coordination aspect specifies the attributes, services, synchronizations and connectors' choreography. This aspect cannot be specified in components and systems.

Synchronizations define the allowed connections between roles types. This specification is realized by means of interfaces in order to achieve a higher abstraction level and to increase the reuse of coordination aspects so that a coordination aspect can be used by several types of connectors.

The **choreography** specifies the coordination process, the glue that the connector applies to synchronise its connected types. The choreography is specified by means of interfaces.

The quality interfaces offer services to measure the quality of a component. A quality aspect specifies the quality attributes that the system wants to evaluate, the quality level that the type must satisfy and the different metrics to use in order to measure the quality of a type.

The quality can be measured at different levels of granularity inside a product software (complete application, component, interfaces, use case, web service, etc). In our model, the quality aspect measures the quality of components and connectors. However, a systems quality is the combination of the valuations of the types that are included.

Context-awareness must give specifications of five interfaces [12]: ISource, IGatherer, IAggregator, IAnalyzer and INotifier. These interfaces are defined in the context-awareness aspect of PRISMA.

- ISource: its services are related to schema management.
- IGatherer: its services gather context information from sources at a given rate (sampling period) and cache the corresponding information
- IAggregator: its services elaborates complex information by correlating information provided by other parties.
- IAnalyzer: its services establish categories and assign descriptors to such categories.
- INotifier: its services provides the asynchronous notification of information.

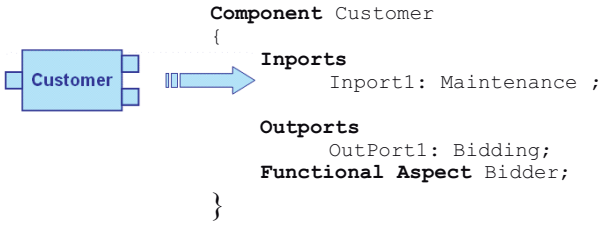
3.1.2. Components, Connectors and Systems. A component has a collection of input and output ports whose types are functional or distributed interfaces. Consequently, two kinds of ports can be defined, those acting on a component's functional and a distributed aspects respectively. Input ports define all the attributes and services that are offered to external types (server behaviour) whereas output ports define attributes and services that are required from other types (client behaviour).

A component definition must always contain a functional aspect (except COTS). However, having a distributed aspect is optional and only components exhibiting a distributed behaviour should incorporate it. It is important to note that only the interfaces (functional or distributed) existing in a component's definition are valid candidate types when defining new ports.

A **connector's** definition consists of two types: a set of input and output roles, and a list of allowed types of components, systems or other connectors that can be eventually connected.

Roles have the same semantics as ports in components. In other words, each role in a connector must have a type which is selected among the available functional or distributed interfaces defined in the connector. However, coordination interfaces may not be used when defining roles. These interfaces are used instead to make visible a given coordination aspect to external types. All the above requirements make functional and a co-ordination aspects mandatory whereas distributed ones are still optional.

Finally, a **system's** definition is equivalent to that of a component because a system is a complex component. Therefore, a system is a composition of connected types (components, connectors and nested subsystems) with emerging functional and distributed aspects. Let us note that including a connector in a system's definition results in including also all the types that are connected by the connector (components, other connectors and systems). Additional connections must be specified when defining a system. These are called bindings and connect the system with its internal types by means of well known interfaces (similar to every synchronization in connectors).



3.1.3. Evolution Aspect. A metalevel can be constructed automatically from the component's definition language. This metalevel serves the purpose of enabling evolution of certain linguistic types in a specification. In order to achieve this goal, i.e. being able to automatically obtain a metalevel, a reification process for those evolving types must be defined.

In our approach a reify operator is proposed whose operational semantics will allow for an automatic reification process. Its semantics will certainly depend on the linguistic granularity to which it is applied. Each linguistic construction to be reified will have a different metalevel generation which will be defined by means of specific transformation patterns.

Evolution can be achieved at two levels: structural and behavioural. Structural evolution is achieved by reifying the properties or attributes of the types in a specification whereas behavioural evolution is achieved by reifying those sections that define some behaviour. This last kind of evolution incorporates kinematics to the language (behavioural evolution without structural change).

All the types presented so far can be linguistically expressed as a term with a tree-like structure. So we can see the reify operator as a mapping between trees, one representing a sub-term in the specification language and another representing a sub-term at the metalevel. In order to perform this transformation the reify operator applies the proper pattern. The selected pattern will express how the matching term in the definition language is transformed into a proper term (describing meta-properties and meta-events) at the metalevel.

IEEE COMPUTER SOCIETY

3.2. The Configuration Language

The configuration language is used to define instances and the topology of the architectural model. In order to do so all needed connectors, components and systems types must be imported. Then all the necessary instances of the imported types can be defined and, finally, the topology of the model is expressed by interconnecting such instances. Such interconnection process is achieved by creating instances of connector's synchronizations and system's binding.

Next a simple arquitectural model for a simple auction system is presented. Figure 7 illustrates this example: two components *Customer1* and *Product1* are connected by means of the Auction1 connector and these three elements are encapsulated by a system, elements and the system are comunicated using bindings.

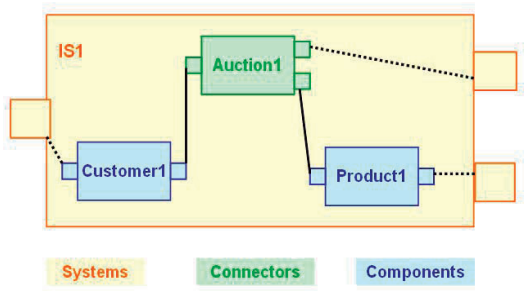


Fig. 7. An Architectural Model Example

```
Architectural Model  Auction

  Import Customer, Auction, Product,
InformationSystem;

  Customer1: Customer;
  Auction1: Auction;
  Product1: Product;
  IS1: InformationSystem;

  Attachements AuctionSession{

     Bidder1.OutPort1 >> Auction1.InPort1;
     Auction1.OutPort1 >> Product1.InPort1;
  }

  System IS1{
             Import Elements Bidder1, Product1,
                    Auction1;

             Import Connections Auction Session;



             BindingLinks{
                           IS1.Inport1 >>
                           Bidder1.Inport1;
                           Auction1.Outport2>>
                           IS1.Outport1;

                           Product1.Outport1>>
                           IS1.Outport2;
                         }
  }
End Arquitectural Model Auction;
```

4. Conclusions

In this chapter a new model: PRISMA has been presented for software architectural modelling. Its main characteristics are the blending of Component Based Design and the Separation of Concerns or Aspect Oriented Programming. Both aspects together with the full use of reflexive techniques are introduced at a very abstract level: the Conceptual Model Specification Level. By means of a 3-tiers metalevel architecture the dynamic reconfiguration aspects are solved in a powerful way. PRISMA is a flexible and integrated model for the specification of complex architectural models of distributed and dynamic systems. The high abstraction level used facilitates the reuse of software artefacts and the independence of concerns by means of a separate compilation process using a Model Compiler approach.

Two languages are used for specifying architectural models: the component definition language used for defining the different artefact types and the Configuration Language allowing the definition and combination of different instances of the previous types. COTS components are integrated in the system directly or as a result of a web-based trading process based on the Component Quality Aspect.

Concerning the quality of the architectural models designed using PRISMA, first, they satisfy the established non-functional quality requirements, because the types must have into account the quality attributes specified in their quality aspect and the second, the architectural models have quality behaviour and code structure due to the reuse and evolution of PRISMA.

5. Future Work

In the medium term a lot of work will be necessary for the PRISMA implementation. Model Compiler Techniques will be used for the generation of the Base level code (Top-Down Compiler) and the Metalevel Code (Bottom-up Compiler) from the different aspects specifications.

We have selected UML as the graphical notation for the PRISMA languages and we are studying the correspondences between our types and its UML

representations. In a short time we want to present our graphical notation.

A deeper understanding of the distribution, quality, co-ordination and dynamic aspects together with the introduction of new aspects will be also necessary. Long term work will include the beta- test of our approach in the problem arena.

Bibliography

[1] L. F. Andrade, J. L. Fiadeiro, J. Gouveia, G. Koutsoukos, M. Wermelinger, "Coordination for Orchestration " , *5th International Conference on Coordination Models and Languages COORDINATION*, Lecture Notes in Computer Science 2315 Springer, York, UK, April 8 11, 2002, ISBN 3 540 43410 0, pp. 5 13.

[2] Aspect Oriented Software Development, http://aosd.net

[3] R. Balzer, "A 15 Year Perspective on Automatic Programming", IEEE Transactions on Software Engineering, November 1985, vol.11, num.11, pages 1257 1268.

[4] P. Botella, X. Burgués, S. Franch, M. Huerta and G. Salazar, *"*Modeling non functional requirements*"*, *Actas de las Jornadas de Ingeniería de Requisitos Aplicada (JIRA),* junio 2001.

[5] A. Bracciali, A. Brogi, C. Canal, "Sistematic Component Adaptation", *In Proc IDEAS*, La Habana, 2002.

[6] J. A. Carsí, "OASIS como marco conceptual para la evolución del software". Ph.D. Thesis 1999. DSIC (Universidad Politécnica de Valencia).

[7] S. Cheng, D. Garlan, B. Schmerl, P. Steenkiste & N.Hu, *"*Software Architecture based Adaptation for Grid Computing*"* , *IEEE Conference on High Performance Distributed Computing*, IEEE Computer Press, Scotland, July, 2002.

[8] A. K. Dey & G. D. Abowd, " Towards a better understanding of context and context awareness", *Proceedings of the Workshop on the What, Who, Where, When and How of Context Awareness*, affiliated with the CHI 2000 Conference on Human Factors in Computer Systems, NY: ACM Press ,New York,.

[9] F. McGurren, D. Conroy, "X ADAPT: An Arquitecture for Dynamic Systems*"*, *Workshop on Component Oriented Programming, ECOOP*, Málaga, Spain, 2002.

[10] D. Garlan, A. Kompanek, "Reconciling the Needs of Architectural Description with Object Modeling Notations", *Proceedings «UML» 2000*, Springer, York, UK, October 2 6, 2000, LNCS 1939, pp.498 512.

[11] H. In, D. Olson, T. Rodges, "Multi criteria preference analysis for systematic requirements negotiation", *in Proceedings of IEEE International Computer Software and Applications Conference (COMP SAC'2002)*, 2002.

[12] J. Jaen, I. Ramos, "A Conceptual Model for Context Aware Dynamic Architectures" Accepted at *the 23rd International Workshop on Distributed Auto adaptive and Reconfigurable Systems*. To appear in IEEE Press, Rhodes Island, 2003

[13] G. Kiczales, E. Hilsdale, J. Huguin, M. Kersten, J. Palm, W.G. Griswold, "An Overview of AspectJ", *proceeding of the European Conference on Object Oriented Programming*, Springer Verlag, 2001.

[14] C. Kobrym, "Modeling Components and Frameworks with UML". Comunications of ACM, 43(10); pp. 31 38. Octubre 2000

[15] P. Letelier, P. Sánchez, I. Ramos, O. Pastor, OASIS 3.0: "Un enfoque formal para el modelado conceptual orientado a objeto". Universidad Politécnica de Valencia, SPUPV 98.4011, ISBN 84 7721 663 0, 1998.

[16] O. Loques, A. Sztajnberg, J. Leite, M. Lobosco, "On the Integration of Meta Level Programming and Configuration Programming", *In Reflection and Software Engineering (special edition)*, Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, Springer Verlag, Heidelberg, Germany, June, 2000 pp.191 210.

[17] N. Medvidovic, R. N. Taylor, "A classification and Comparison Framework for Software Architecture Description Languages*"*, *IEEE Transactions of SW Engineering*, Vol. 26, nº 1, January, 2000

[18] R. Monge, C. Alves, A. Vallecillo, "A Graphical Representation of COTS based Software Architectures", *In Proc IDEAS*, La Habana, 2002.

[19] A. Oliva, I. C.Garcia, L. E. Buzato*, "The Reflective Architecture of Guaraná"*, Technical Report IC 98 14. Instituto de computación, Universidad de Campiñas, abril, 1998.

[20] Object Management Group (OMG), http://www.omg.org

[21] A. Rashid, "A Hybrid Approach to Separation of Concerns: The Story of SADES", *Third International Conference, REFLECTION 2001*, Proceedings. Lecture Notes in Computer Science 2192 Springer , Kyoto, Japan, September 25 28, 2001, ISBN 3 540 42618 3, pp. 231 249

[22] A. Ruiz,"Una Aproximación Semicualitativa al Tratamiento Automático de Requisitos de Calidad: Aplicación a la Obtención Automática de Acuerdos de Nivel de Servicio en MOWS", Ph.D. Thesis 2002. Departamento de Lenguajes y Sistemas Informáticos (Univesidad de Sevilla)

[23] P. Kruchten, B. Selic, W. Kozaczynski," Describing Software Architecture with UML", *Proceedings of ICSE* , Toronto, Ontario, Canada, 12 19 Mayo, 2001 pp. 715 716.

[24] Zinky J., Bakken D., Schantz R., *"Architectural Support for Quality of Service for CORBA Objects,"* Theory and Practice of Object Systems 3(1), 1997.

IEEE COMPUTER SOCIETY

PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes

Jennifer Pérez, Nour H. Ali, Isidro Ramos, Jose A. Carsí
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46022 Valencia – España
{jeperez | nourali | iramos | pcarsí}@dsic.upv.es

Resumen. El desarrollo de sistemas software actuales es necesario abordarlo mediante plataformas que permitan describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables. PRISMA es un modelo que integra el Desarrollo de Software Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA) para la construcción de modelos arquitectónicos. Dicho modelo permite evolucionar sus arquitecturas gracias a sus capacidades reflexivas implementadas mediante su meta-nivel. Este trabajo presenta la visión orientada a aspectos de PRISMA: el soporte que ofrece al DSOA, su capacidad de evolucionar aspectos y las ventajas que estos aportan al modelo.

Palabras clave: Arquitecturas Software Orientadas a Aspectos, Análisis y Diseño de Software Orientado a Aspectos.

1. Introducción

Los sistemas de información cada vez son más difíciles de desarrollar debido a sus complejas estructuras y a la dinámica y competitividad del mercado que implica una evolución constante de los requisitos del sistema. Éstas, entre otras razones, hacen que los entornos de desarrollo deban permitir construir aplicaciones con arquitecturas complejas, distribuidas, evolutivas y reutilizables.

El modelo PRISMA [Per03] se basa en componentes y aspectos para la construcción de modelos arquitectónicos. Dicho modelo se caracteriza por la integración que realiza del Desarrollo de Software Basado en Componentes (DSBC) y del Desarrollo de Software Orientado a Aspectos (DSOA) y por sus propiedades reflexivas. El DSBC propone el desarrollo de sistemas software mediante el acoplamiento de entidades que proporcionan servicios específicos. El paradigma orientado a aspectos integrado al DSBC separa funcionalidades de las entidades (componentes) que pueden ser descritas independientemente de las otras. De esta forma, es posible analizar, manipular y adaptar estas funcionalidades de forma separada al resto. Además, la capacidad de reutilización no solo se limita a las entidades, también es posible reutilizar estas funcionalidades separadas en aspectos. Estas ventajas son las que consigue PRISMA mediante la integración de DSBC y DSOA. La capacidad reflexiva se consigue a través de un meta-nivel que permite evolucionar al modelo arquitectónico. PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables.

En este trabajo se presenta la definición de aspecto, el modo en que el modelo PRISMA da soporte al desarrollo de software orientado a aspectos y las ventajas que éste le proporciona. Su estructura es la siguiente: en primer lugar se presenta el modelo arquitectónico PRISMA. En el apartado 3, se presenta la perspectiva de aspectos PRISMA. En el apartado 4 se comenta el estado del arte del desarrollo de arquitecturas basadas en componentes y aspectos. Finalmente, en el apartado 5 se presentan las conclusiones y trabajos futuros.

2. Modelo PRISMA

El modelo arquitectónico PRISMA que se presenta a continuación integra dos aproximaciones de desarrollo de software, el DSBC y el DSOA. Esta integración se consigue definiendo los tipos mediante la composición (monotónica) de aspectos.

Pérez, J., Ali, N., Ramos, I., Carsí, J. A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes. Workshop Desarrollo de Software Orientado a Aspectos (DSOA). Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003.

La mayoría de modelos arquitectónicos analizan cuáles son los tipos básicos para la especificación de sus arquitecturas y exponen su sintaxis y semántica. El modelo PRISMA, además de definir sus tipos, también especifica los aspectos que cubren las necesidades de cada uno de ellos.

Un tipo del modelo de arquitectura puede ser visto como un prisma con tantas caras como aspectos considere (ver figura 1). Dichos aspectos están definidos desde la perspectiva del problema y no de su solución, aumentando el nivel de abstracción y evitando el solapamiento de código (no monotonicidad) que puede producir la programación orientada a aspectos [Kic01].

El metanivel de PRISMA y las propiedades reflexivas de los lenguajes diseñados dan soporte tanto para la evolución de los elementos, como para la reconfiguración dinámica de la topología. Esta característica del modelo permite definir un metanivel por aspecto que reifica las propiedades que deseen ser evolucionadas basándose en la reflexión de OASIS [Car99]. De este modo, PRISMA da soporte a la evolución de sus modelos reduciendo el esfuerzo de mantenimiento de sus productos. La evolución proporciona la dinámica del tipo, es decir, la capacidad de cambio de su estructura y comportamiento basándose en la evolución del software que plantea la reflexión de OASIS.

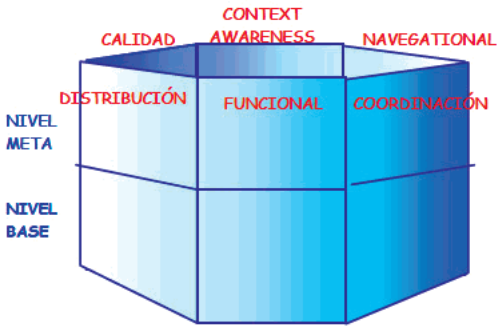


Figura 1. Tipo PRISMA

El modelo arquitectónico consta de tres tipos: componentes, conectores y sistemas. Cada uno de estos tipos esta compuesto por tantos **aspectos** como se consideren relevantes para definir el sistema de información con precisión. Esto permite que el modelo PRISMA pueda analizarse desde dos perspectivas diferentes (ver figura 2), la orientada a aspectos y la orientada a componentes. En este trabajo se presenta la visión de aspectos del metamodelo PRISMA.

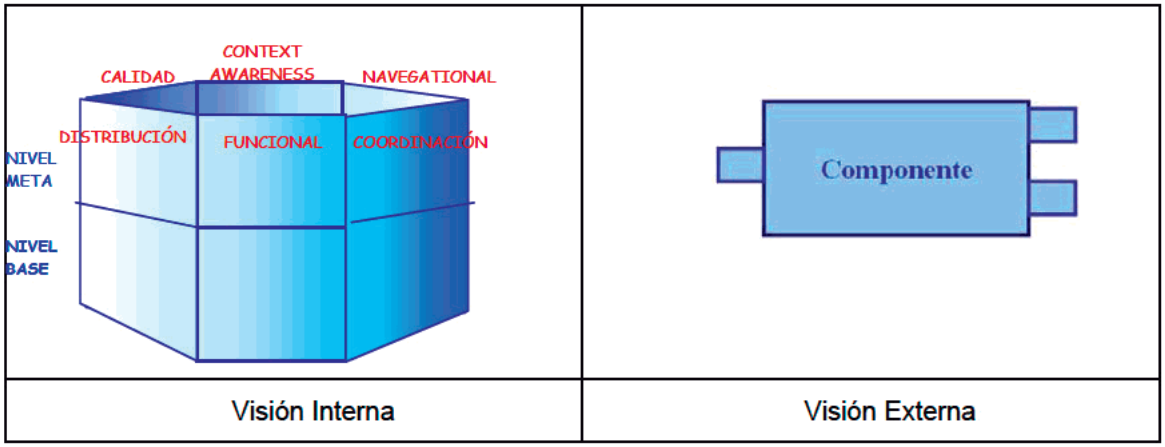


Figura 2. Vistas de un componente PRISMA

PRISMA esta basado en el lenguaje de especificación formal OO llamado OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos, que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA preserva las ventajas de OASIS, garantizando la compilabilidad de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos arquitectónicos, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

3. DSOA en PRISMA

Los aspectos se definen en [Kiz97] como propiedades de un sistema que tienden a estar presentes en varias componentes funcionales, dando lugar al término inglés *crosscutting concerns*. La distribución y sincronización son ejemplos claros de *crosscutting concerns* a nivel de implementación, ya que no están presentes en un solo componente, sino que están dispersas en varios.

En este trabajo se presenta una nueva definición de aspecto, entendiendo un aspecto como la definición completa de la estructura y el comportamiento de un tipo desde un determinado punto de vista (*concern*). Hay que tener en cuenta que dicho punto de vista puede ser cualquiera, incluso el funcional. Por lo tanto, el *crosscutting* no sólo se realiza sobre propiedades funcionales sino sobre cualquier *concern.*

El hecho de que el paradigma de la orientación a aspectos (AOP) y el DSOA hayan surgido de los lenguajes de programación orientados a aspectos ya consolidados como AspectJ [Kic01], ha ocasionado que la mayoría de aproximaciones abstraigan el concepto de aspecto a partir del nivel de implementación, estando claramente influenciados por dichos lenguajes. Por este motivo, estas aproximaciones asocian siempre el concepto de aspecto con propiedades no funcionales.

Sin embargo, en PRISMA se define el concepto de aspecto sin tener en cuenta las peculiaridades sintácticas de dichos lenguajes. Su definición se basa en las características comunes de un sistema susceptibles de ser reutilizadas. Atendiendo a esta definición, menos dependiente del lenguaje de desarrollo, la funcionalidad es susceptible de reutilización en distintos tipos e incluso dicha funcionalidad se puede subdividir en distintos aspectos. Los últimos trabajos de Ana Moreira [Brio03] empiezan a apuntar hacia esta última idea, a pesar de no haberse llevado todavía a la práctica.

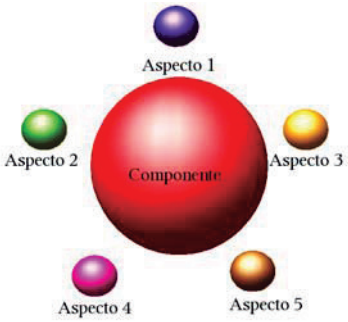


Figura 3. Aspectos asociados a un componente funcional

Este cambio de la definición de aspecto proporciona un conjunto de ventajas al modelo, permitiendo que un tipo se construya mediante un operador composicional de aspectos (ver figura 4) y no mediante un tipo funcional al que se le añaden aspectos (ver figura 3). De esta manera, se consigue un modelo más uniforme, ya que los parámetros origen a partir de los cuales se construye el tipo son homogéneos. Por otro lado, se consigue una mayor reutilización y modularidad al reutilizar el aspecto funcional y considerar que puede ser común a un conjunto de tipos. Este aporte es importante, ya que la definición de [Kic97] limita la posibilidad de que existan un conjunto de tipos que tengan en común su funcionalidad y que lo que les diferencie unos de otros sean otros aspectos. Finalmente, se ha de tener en cuenta que esta definición de aspecto hace que el modelo sea completamente independiente del lenguaje de implementación, teniendo la posibilidad de compilarse tanto a lenguajes orientados a componentes como a lenguajes orientados a aspectos.

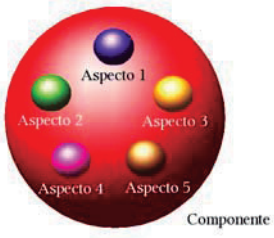


Figura 4. Aspectos que forman un componente

En PRISMA, un aspecto puede verse como la unión de un conjunto de interfaces que incluyen servicios del tipo del aspecto, más la especificación semántica de su estructura y comportamiento. Por lo tanto, un aspecto es una plantilla que recoge las propiedades de estructura y comportamiento compartidas por todas las instancias del componente que tengan dicho aspecto.

Un aspecto ($\Sigma_{aspecto}$) se define mediante una función de identificación (I) y una plantilla (P). De forma que dicha plantilla esta representada por la tupla (A, X ,Φ,∏), siendo:

A: El conjunto de atributos
X: El conjunto de servicios
Φ: El conjunto de formulas en lógica dinámica
∏: Términos en álgebra de procesos

A su vez, A y X forman la signatura o conjunto de interfaces que describen las propiedades visibles del aspecto.

Existen diferentes tipos de aspecto, definiéndose cada uno de ellos de forma independiente. El número de tipos de aspecto no esta limitado, ya que gracias al metanivel, se pueden definir nuevos. La necesidad de nuevos tipos de aspecto emerge de los requisitos de los sistemas arquitectónicos a los que se aplique PRISMA. Por lo tanto, a través de los requisitos del sistema se identificarán tanto los aspectos del sistema como las necesidades que han de cubrir cada uno de ellos [Nav03].

Existe un conjunto de conceptos que forman un aspecto de forma general, además de aquellos propios de cada aspecto. Estos estan definidos en el metamodelo de PRISMA, tal y como se muestra en la figura 5.

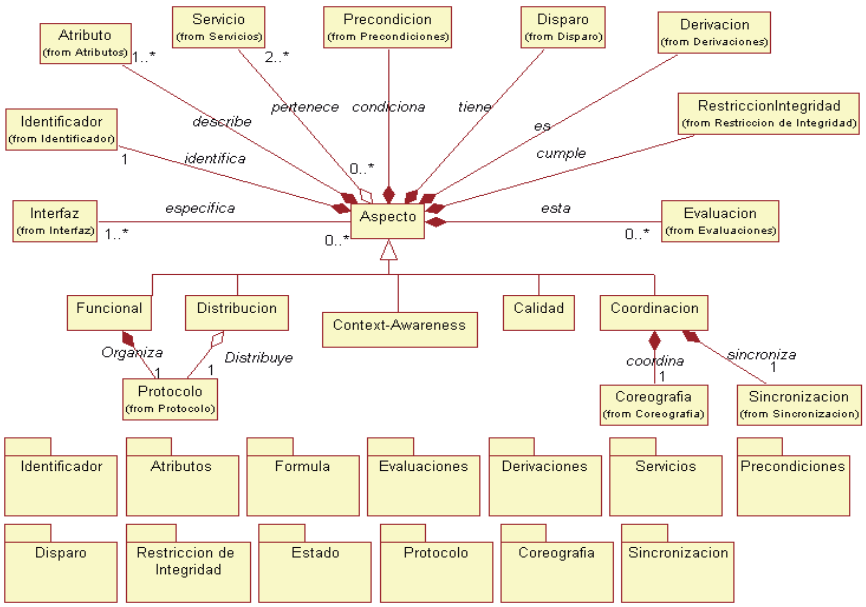


Figura 5. Paquete aspecto del metamodelo de PRISMA

La especificación de la relación inter-aspectos se realiza en el componente mediante el *weaving*. El *weaving* es necesario para sincronizar los aspectos que componen un componente. Esta sincronización es necesaria ya que la ejecución de un servicio en un aspecto puede desencadenar acciones en otro aspecto de la componente y a que las decisiones que se toman en un aspecto pueden estar influenciadas por el valor de los atributos de otros aspectos. Por ejemplo, el seguir un enlace del aspecto navegacional puede implicar un cambio en el aspecto de presentación de la componente.

El weaving se realiza en el componente porque de ese modo se preserva la independencia y reutilización de los aspectos y se aumenta la flexibilidad del modelo. La independencia y reutilización se consiguen al no definir dentro de cada aspecto su sincronización con el resto, ya que no se establecen referencias que generen dependencias con otros aspectos. Mientras que la flexibilidad se aumenta debido

a que es posible establecer un comportamiento diferente en componentes que estén formados por los mismos aspectos, definiendo distintas sincronizaciones inter-aspectos. Por ejemplo, la invocación de un link del aspecto navegacional de un componente se puede sincronizar con un servicio que realice un cambio en el aspecto de presentación del componente; sin embargo, en otro componente, en el que el servicio que se desencadene en el aspecto de presentación sea diferente, obteniendo así en cada caso una presentación de la información en un formato diferente.

La sincronización entre servicios PRISMA de diferentes aspectos se puede realizar de tres formas diferentes: ***after***, ***before*** o ***around***. Estas tres formas de sincronizar los servicios inter-aspectos son las que propone la programación orientada a aspectos. PRISMA mantiene los tres modos de sincronización, no sólo por seguir el enfoque orientado a aspectos, también porque cubren todas las necesidades de sincronización inter-aspecto que el modelo PRISMA necesita.

4. Estado del arte

En la actualidad existen una gran variedad de trabajos relacionados con modelos arquitectónicos, componentes, aspectos, lenguajes de definición de arquitecturas, reconfiguración dinámica de arquitecturas y todo aquello que guarda relación con el desarrollo de sistemas software complejos. Este gran esfuerzo de investigación persigue el objetivo de encontrar un consenso en los conceptos básicos de sistemas software complejos, así como una definición precisa y una metodología a aplicar para su desarrollo.

El concepto de arquitectura software está muy extendido, sin embargo no existe una definición única y estándar para este concepto. Un autor, cuya contribución en el mundo de las arquitecturas software y las componentes ha sido muy aceptada, es David Garlan. Sus trabajos han sido un gran precedente en la elaboración de este trabajo y su definición de arquitectura es una de las más utilizadas en la literatura [Gar95].

*La **arquitectura software** esta compuesta por la estructura de los componentes de un programa o sistema, sus interrelaciones, y los principios y reglas que gobiernan su diseño y evolución a lo largo del tiempo.*

En una línea cercana a la nuestra, se encuentra Model-Driven Architecture (MDA), un enfoque propuesto por la OMG (www.omg.org) para la integración e interoperabilidad de los sistemas. MDA enfatiza el modelado (con UML) independiente de las plataformas de implementación. MDA está basado en el modelado de diferentes aspectos y niveles de abstracción, explotando las interrelaciones y estableciendo transformaciones hacia su implementación.

El desarrollo de software orientado a aspectos [Aos03] esta teniendo un gran auge en la comunidad informática debido a que su código modular consigue que los costes de desarrollo, mantenimiento y evolución del software se reduzcan. Por este motivo, existe una tendencia hacia la incorporación del concepto de aspecto desde las primeras fases del ciclo de vida, existiendo trabajos elaborados en el área de la especificación de requisitos [Ras02].

El principio de *separation of concerns* propone la encapsulación de propiedades en entidades separadas que localicen los cambios y traten una misma cuestión. Se han propuesto varios modelos de separación de propiedades, sin embargo, de todos ellos el que ha tenido una mayor aceptación ha sido el paradigma orientado a aspectos (AOP).

Dentro de los distintos modelos de aspectos que se han creado hasta el momento, se pueden observar dos claras tendencias que permiten clasificarlos en dos grandes grupos: los modelos estáticos y los dinámicos. La diferencia entre ambos se encuentra en el proceso de generación de código. Los modelos estáticos (AspectJ) generan un solo componente en el que se encuentra mezclado el código funcional y el código de los aspectos, mientras que en los modelos dinámicos se generan distintas entidades para los distintos aspectos y el componente (Modelo de disfraces) [Her03] . La ventaja de los modelos dinámicos es la facilidad que proporcionan para incluir o eliminar aspectos durante el proceso de ejecución.

Alguna de las aproximaciones que han integrado el DSBC y el DSOA a nivel de implementación han extendido los java beans utilizando aspectos para su descripción y añadiendo el concepto de conectores [Suv03]. Además, los aspectos pueden ser manipulados en tiempo de ejecución. Un trabajo que describe

los asuntos que se tienen que tener en consideración para la integración de ambos aproximaciones es introducido en [Con00]. Los asuntos y requisitos descritos en este trabajo son contemplados en PRISMA.

5. Conclusiones y Trabajos Futuros

Se ha presentado el modelo PRISMA desde su punto de vista orientado a aspecto. Dicho modelo se caracteriza por el tratamiento de los aspectos a un alto nivel de abstracción, su capacidad de evolución en el marco de la orientación de aspectos, su definición de *concern* y el modo en el que se realiza el weaving monotónico entre aspectos. Todas estas propiedades dotan a PRISMA de una gran flexibilidad, reutilización, modularidad y evolución dentro del desarrollo orientado aspectos.

Existen una gran variedad de trabajos que se desean abordar dentro de los distintos aspectos que soporta el modelo PRISMA. Las líneas de trabajo a corto plazo a destacar se centran en el operador monotónico de *weaving* y su especificación en el lenguaje de definición de arquitecturas PRISMA y la elaboración de un profile UML para la especificación gráfica de modelos arquitectónicos PRISMA.

6. Bibliografía

[Aos03] Aspect Oriented Software Development, http://aosd net

[Bri03] I Brito, A. Moreira. *Towards a Composition Process for Aspect-Oriented Requirements*. Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 2nd International Conference on Aspect-Oriented Software Development,Boston, USA, 17 March 2003.

[Car99] Carsí J. A., "OASIS como marco conceptual para la evolución del software". Ph.D. Thesis 1999. DSIC (Universidad Politécnica de Valencia).

[Con00] Constantinides C.A., Errad T., *"On the Requirements for Concurrent Software Architectures to Support Advanced Separation of Concerns*", Proceedings of OOPSLA 2000 Workshop on Advanced Seperation of Concerns in Object-Oriented Systems

[Kic01] G. Kiczales, E. Hilsdale, J. Huguin, M. Kersten, J. Palm, W.G. Griswold, *An Overview of AspectJ*, proceeding of the European Conference on Object-Oriented Programming, Springer-Verlag, 2001.

[Kiz97] G. Kiczales et al., *Aspect-Oriented Programming*, Proc. European Conference on Object-Oriented Programming (ECOOP' 97). LNCS 1241, Springer Verlang, 1997.

[Her03] Herrero, Propuesta de una Plataforma, Lenguaje y Diseño para el desarrollo de Aplicaciones Orientadas a Aspectos*,* Tesis Doctoral, Extremadura, España, 2003.

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "*Un enfoque formal para el modelado conceptual orientado a objeto*". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721- 663-0, 1998.

[Nav03] E. Navarro, I. Ramos, J. Pérez, *Software Requirements for Architectured Systems*, 11t h IEEE International Requirements Engineering Conference (RE'03), September 8-12, 2003 in Monterey, California. (Pendiente de Publicación)

[Per03a] J. Pérez, I. Ramos, J. Jaén and P. Letelier, E. Navarro: "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*",* 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 (Pendiente de Publicación)

[Ras02] A. Rashid, P. Sawer, A. Moreira, J. Araújo. "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", International Conference on Requirements Engineering (RE 2002), University of Essen, Germany, September 9-13, 2002.

[Suv03]] Sube D., Vanderperren W., Jonckers V., JAsCo: an Aspect-Oriented approach tailored for CBSD. In Proceedings of AOSD International Conference, Boston USA. ACM Press. March 2003.

Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA

Jennifer Pérez, Nour Hussein, Isidro Ramos

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
46071 Valencia, España
{jeperez | nourali | iramos}@dsic.upv.es

Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez

Departamento Tecnologías de la Información y las Comunicaciones
Universidad Politécnica de Cartagena
Campus Muralla del Mar s/n
30202 Cartagena, España
{juanangel.pastor | pedro.sanchez | balvarez }@upct.es

Resumen. El desarrollo de software basado en componentes y aspectos esta despertando mucho interés, no sólo en el ámbito académico, sino también en el industrial. En este trabajo se aplica el modelo PRISMA a un caso de estudio de ámbito industrial en el marco del proyecto europeo EFTCoR (Environmental Friendly and Cost-effective Technology for Coating Removal). El caso de estudio consiste en el diseño y construcción de un sistema robótico teleoperado dedicado a la limpieza de cascos de buques. El diseño de la arquitectura de estos sistemas se debe abordar desde un alto nivel de abstracción de forma que sea posible reutilizar los diseños para distintos tipos de arquitecturas y que sean fácilmente modificables. Además, la evolución de los diseños debe realizarse tanto en tiempo de compilación como en tiempo de ejecución, de forma correcta y preservando los requisitos de calidad del sistema. El desarrollo del caso de estudio muestra como PRISMA reúne todos los requisitos necesarios para la construcción de arquitecturas complejas, distribuidas, evolutivas y reutilizables, como los sistemas de tele-operación, de forma sencilla a través de las propiedades reflexivas de sus lenguajes: el lenguaje de definición de componentes y el lenguaje de configuración de modelos arquitectónicos. **Palabras Clave:** Modelos de Arquitectura, Evolución, Reconfiguración Dinámica, Sistemas de Teleoperación

1. Introducción

Los sistemas software cada vez son más difíciles de desarrollar debido a sus complejas estructuras, la dinámica y la competitividad del mercado, que implica una evolución constante de sus requisitos. Los entornos de desarrollo deben permitir construir aplicaciones con arquitecturas complejas, distribuidas, evolutivas y reutilizables.

En este trabajo se presentan las propiedades y ventajas que aporta PRISMA (Plataforma OASIS para Modelos Arquitectónicos) [23] en la construcción de modelos arquitectónicos complejos mediante la especificación de un caso de estudio. El desarrollo del caso mostrará como PRISMA potencia la reutilización, soporta la evolución de sus tipos y topologías mediante sus capacidades reflexivas y aglutina

Pérez, J., Ali, N., Ramos, I., Pastor, J. A., Sánchez, P., Álvarez, B. Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003.

dos aproximaciones actuales para el desarrollo de sistemas: el Desarrollo de Software Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA). Entre los distintos tipos de sistemas software que tienen arquitecturas complejas, distribuidas, evolutivas y reutilizables, en este trabajo se ha elegido como ámbito de aplicación de PRISMA los sistemas de tele-operación. Los sistemas de tele-operación son sistemas de control que dependen del software para realizar sus operaciones. Su diseño es una tarea muy compleja que debe integrar elementos mecánicos, eléctricos y electrónicos y componentes software en un mismo sistema. Estos se utilizan para tele-operar mecanismos (robots, vehículos y herramientas) que llevan a cabo actividades de inspección y mantenimiento en entornos cuyas condiciones de habitabilidad o riesgo desaconsejan o impiden la presencia de operarios humanos.

El dominio de la tele-operación es muy extenso, existiendo sistemas de muy diverso tipo y cada uno de ellos con diferente arquitectura. El grupo DSIE (División de Sistemas e Ingeniería Electrónica) de la UPCT ha venido utilizando una arquitectura de referencia para sistemas de tele-operación [2] en diferentes proyectos [13]. Dicha arquitetura se ha evaluado mediante la metodología ATAM (Architecture Tradeoff Analysis Method) [16] del SEI (Software Engineering Institute) para determinar su adecuación a nuevos proyectos en el ámbito de la tele-operación de robots más complejos en los que el entorno de trabajo no es estructurado, esto es, no se conoce completamente de antemano [11][10]. Pero los resultados de dicha evaluación determinaron la inviabilidad de utilizarla en los nuevos desarrollos, entre otros motivos, debido a la necesidad de tener que -modificar de forma dinámica los patrones de interacción entre componentes.

En el marco de el proyecto europeo EFTCoR (Environmental Friendly and Cost-effective Technology for Coating Removal), en el que participamos junto con varios países, se está trabajando actualmente en el diseño y construcción de un sistema robótico tele-operado. Dicho sistema esta dedicado a la limpieza de cascos de buques de gran calado [10], que minimice el impacto medioambiental y corrija los defectos detectados en los prototipos anteriores [11]. Dentro de este proyecto, surge con fuerza la necesidad de diseñar arquitecturas para sistemas de tele-operación a un mayor nivel de abstracción, de forma que sea posible reutilizar los diseños para distintos tipos de arquitecturas y que sean fácilmente modificables. Además, aparece la necesidad de la evolución de los diseños de forma correcta y preservando los requisitos de calidad del sistema, tanto en tiempo de compilación como en tiempo de ejecución. De estos complejos requisitos software surge el modelo PRISMA, su potencia expresiva, facilidad de uso y novedad se demuestran en este trabajo. De esta manera se cubre el hueco existente en el modelado de sistemas software altamente reconfigurables y reutilizables dentro de un marco de calidad controlada.

Otros trabajos PILAR [9], LEDA [7], Wright [1]… abordan el mismo problema pero su ámbito de aplicación es el de procesos distribuidos sencillos. Sin embargo, PRISMA pretende tener una proyección industrial en la que los modelos arquitectónicos son complejos y los sistemas de información que se especifican son abiertos y activos (sistemas hipermedia como subastas electrónicas y museos híbridos [15], sistemas de teleoperación, etc.)

La estructura del trabajo es la siguiente: en primer lugar se detallan los trabajos existentes de lenguajes de descripción de arquitecturas y sus ámbitos de aplicación.

En el apartado 3, se presenta el modelo arquitectónico PRISMA En el apartado 4, se presenta el sistema de tele-operación EFTCoR como caso de estudio a desarrollar. En el apartado 5, se aplica PRISMA para la especificación de EFTCoR y se muestra cómo soporta sus necesidades de evolución y reutilización. Finalmente, en el apartado 6 se presentan las conclusiones y trabajos futuros.

2. Trabajos Relacionados

Existe una gran variedad de lenguajes de definición de arquitecturas, cada uno de ellos presenta ventajas e inconvenientes, tal y como presenta el estudio realizado en [Med00]. Nuestros lenguajes se basan en el trabajo realizado por Garlan [12] para la descripción de arquitecturas y extienden el concepto de contrato introducido en los trabajos de Andrade y Fiadeiro [3]. Dicha extensión se produce debido a la incorporación del concepto de coreografía en el conector, concepto utilizado con una semántica distintita en los trabajos de [19] y [6].

Uno de los lenguajes que más se aproxima a los nuestros debido a sus capacidades de evolución y reconfiguración es el presentado en el trabajo propuesto por Loques y Leite [18] en su modelo R-RIO. Existen otros enfoques que utilizan técnicas con un nivel de abstracción inferior, orientadas a implementación [20]. En GUARANÁ [21] se presenta un meta-nivel complejo que hay que definir durante la compilación, siendo su estructura ajena al modelo.

En una línea cercana a la nuestra, se encuentra Model-Driven Architecture (MDA), un enfoque propuesto por la OMG [22] para la integración e interoperabilidad de los sistemas.

Nuestro trabajo PRISMA reúne las ventajas de las arquitecturas software previamente presentadas, más las ventajas que permite el desarrollo orientado a aspectos [4]. Los conceptos del desarrollo orientado a aspectos se abordan a nivel de diseño combinándolos con los del desarrollo orientado a componentes. De esta forma, nuestra arquitectura no solo incorpora la reconfiguración dinámica de las topologías sino también la evolución de los tipos de la arquitectura a través de aspectos, consiguiendo la manipulación, evolución y reutilización de diferentes aspectos de los elementos no solo en tiempo de compilación sino también en tiempo de ejecución.

El ámbito de aplicación de los trabajos PILAR, LEDA, Wright es el de procesos distribuidos sencillos de ámbito académico en los que no hay complejidad y evolución. Normalmente, los ejemplos que utilizan son filtros de tuberías y buffers. Sin embargo, PRISMA pretende tener una proyección industrial en la que los modelos arquitectónicos son sistemas de información complejos, abiertos y activos.

3. PRISMA

PRISMA es un enfoque para la construcción de modelos arquitectónicos basado en aspectos y componentes. Dicho enfoque utiliza una extensión de OASIS[17] para la definición formal de la semántica de los modelos de arquitectura software. Dicha extensión ha sido necesaria porque OASIS es un lenguaje formal para definir modelos

conceptuales de sistemas de información orientados a objetos y no permite especificar arquitecturas software. Sin embargo, OASIS permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos mediante técnicas de compilación de modelos. La extensión de OASIS ha dado lugar a un lenguaje de definición de componentes para definir los tipos de la arquitectura (componentes, conectores y sistemas) y a un lenguaje de configuración para especificar modelos arquitectónicos. Ambos pueden ser compilados para generar automáticamente el código del producto siguiendo el paradigma de la programación automática [5].

El lenguaje de definición de componentes de PRISMA [23] potencia la reutilización y aglutina el DSBC y DSOA. Por ello, un elemento de nuestro modelo arquitectónico esta constituido por un conjunto de aspectos: funcional, distribución, coordinación, evolución [8], context-awareness [14], calidad [24], etc. Este lenguaje tiene como ciudadanos de primer orden: interfaces, aspectos, componentes y conectores.

Un tipo o una instancia (de componente, conector o sistema) de nuestro modelo de arquitectura puede ser visto como un prisma con tantas caras como aspectos considere el tipo de elemento que representa, teniendo en cuenta que el aspecto de evolución no es una cara del prisma, sino la parte superior de la división en niveles base y meta de cada cara (ver figura 1). Cada una de ellas tendrá definido un nivel meta, siempre y cuando en el elemento se reifique alguna de las propiedades del nivel base del aspecto que representa dicha cara. La reificación puede afectar a cualquier sección de la especificación OASIS de la entidad o propiedad del aspecto, lo que proporciona al lenguaje de definición de arquitecturas su capacidad reflexiva.

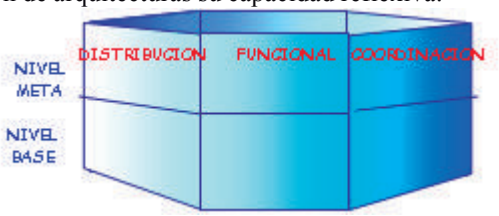


Fig. 1. El tipo PRISMA

El lenguaje de configuración permite definir las instancias y especificar la topología del modelo arquitectónico. Para ello, se han de importar todos aquellos tipos de conectores, componente y sistemas, definidos mediante el lenguaje de definición de componentes, que se necesiten para un determinado modelo de arquitectura. Después se define el conjunto de instancias necesarias de cada uno de los tipos importados. Finalmente se debe definir la topología, interconectando adecuadamente las instancias del modelo.

A diferencia de otros ADL's (Architecture Definition Languages), PRISMA tiene dos lenguajes de especificación en lugar de uno. Esta diferenciación de lenguajes proporciona importantes ventajas frente al uso de un único lenguaje. Una de ellas es que permite gestionar de forma independiente los tipos de elementos y las topologías específicas de cada sistema, obteniendo de este modo un mayor nivel de reutilización y un mejor mantenimiento de las librerías de tipos. Además, permite discernir claramente entre la evolución de tipos (lenguaje de definición de componentes) y la

evolución arquitectónica (lenguaje de configuración), teniendo meta-eventos diferentes en cada lenguaje para hacer evolucionar a los tipos o a las conexiones existentes entre sus instancias.

4. El sistema de teleoperación EFTCoR

El sistema EFTCoR tiene como objetivo muy ambicioso realizar una plataforma robótica tele-operada para operaciones de mantenimiento de cascos de buques respetuosa con el medio ambiente y apta para ser utilizada en ambientes hostiles. De cara a minimizar las posibilidades de error en la construcción del sistema y siendo conscientes de la existencia de una serie de riesgos asociados, se ha optado por un desarrollo incremental del sistema EFTCoR. Cada incremento da como resultado una mayor autonomía del sistema final Actualmente se está trabajando en la construcción de la primera fase del sistema EFTCoR. En la **Primera fase,** debido a la distancia existente entre el operador que maneja el dispositivo de limpieza y el propio cabezal de limpieza se hace imprescindible la incorporación de varias cámaras que muestren al operador en tiempo real la imagen de la superficie que va a ser tratada. En esta primera fase el operador decide qué movimientos realiza el sistema tele-operado y en qué puntos se precisa la limpieza mediante el chorreado. En la **segunda fase**, el robot capturará en tiempo real imágenes de la superficie del casco y tomará las decisiones automáticas de ubicación del cabezal en la zona afectada y el posterior tratamiento de limpieza. Mientras que en la **tercera fase,** se tendrá en cuenta la coexistencia de una familia de robots de limpieza.

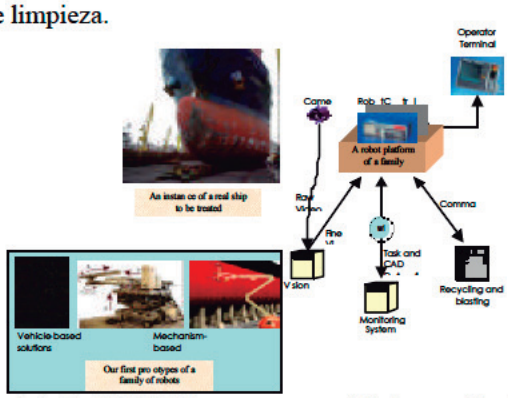


Fig.2. EFTCOR: un esquema del sistema final

La figura 2 muestra el detalle de las distintas partes del sistema de tele-operación considerando una familia de sistemas robóticos dependiendo de las condiciones de limpieza (inaccesibilidad, etc.).

La inclusión de PRISMA desde el principio del proyecto representa un reto en el ámbito de la ingeniería del software debido a la constante evolución de las especificaciones y a la combinación de dos áreas tradicionalmente separadas: la construcción de sistemas industriales y la utilización de métodos formales de desarrollo de software.

A continuación se incluyen los pasos realizados según los objetivos planteados y se presenta de qué manera el enfoque PRISMA repercute positivamente en el alcance de los requisitos establecidos para el sistema EFTCoR.

5. Enfoque arquitectónico para el sistema de teleoperación utilizando PRISMA

El sistema EFTCoR no es un único producto, sino una familia de productos relacionados que comparten un conjunto común de requisitos, lo suficientemente amplio como para diseñar teniendo como objetivo la reutilización. Pero al mismo tiempo, estos requisitos exhiben una variabilidad significativa respecto de los productos. La forma en que se trate tal variabilidad determinará el éxito o fracaso de la arquitectura, razón por la cual es necesario caracterizarla de la forma más precisa posible. En este apartado se presenta, por un lado las principales fuentes de variabilidad que afectan al diseño arquitectónico del sistema de estudio y por otro, las soluciones que aporta PRISMA para el tratamiento de dicha variabilidad.

5.1 Fuentes de Variabilidad.

Para los sistemas considerados, las principales fuentes de variación afectan al funcionamiento y a la posibilidad de adaptación o crecimiento del sistema. Las fuentes de variación que influyen en el funcionamiento del sistema vienen dadas por:
- Existencia de diferentes modos de operación del sistema (modo normal, modo calibración, modo mantenimiento, etc.).
- La posibilidad de cambiar los procedimientos que gobiernan el desarrollo de las operaciones de mantenimiento. Por ejemplo, al introducir nuevos pasos en el procedimiento, cambiando su orden, añadiendo nuevos eventos y transiciones, etc. Estas variaciones pueden implicar cambios en la lógica de la máquina de estados que gobierna la ejecución automática de las mismas (nuevos eventos, transiciones y estados) y cambios en la coordinación entre los diferentes componentes del sistema.

Mientras que las fuentes de variación que afectan la adaptación y crecimiento vienen dadas por:
- La posibilidad de ampliar las operaciones de mantenimiento (lavado, pintura, mecanizados, soldaduras, etc).
- La posibilidad de utilizar diferentes tecnologías (y por tanto diferentes herramientas) para un mismo proceso de mantenimiento. Por ejemplo, limpieza por granallado o por agua a presión ultra-alta: cada herramienta tiene sus propios parámetros y algoritmos de control. Esta variación implica cambios en los algoritmos de cálculo de la cinemática, la adición, eliminación o modificación de comandos, cambios en los rangos de sus parámetros, etc.
- La posibilidad de utilizar diferentes sistemas de posicionamiento y diferentes combinaciones de sistemas de posicionamiento (primarios y secundarios). Esta variación tiene como consecuencia los mismos cambios que la anterior.

Los cambios en la arquitectura que implican estas fuentes de variación pueden clasificarse para su estudio y tratamiento en tres grandes categorías: **cambios en la funcionalidad** de los componentes, **cambios en los patrones de interacción** entre los componentes y **cambios en la infraestructura de ejecución** y de comunicaciones entre componentes. Los **cambios en la funcionalidad** implican cambiar dinámicamente las propiedades que tiene un componente en un momento dado, añadir nuevos componentes, construir componentes complejos a partir de otros más sencillos, sustituir unos componentes por otros en tiempo de ejecución .Los **cambios en los patrones de interacción** pueden deberse a nuevos requisitos de disponibilidad (posibilidad de redundancia), rendimiento, distribución, seguridad, ..., que dan lugar a variaciones en el modo de funcionamiento y a diferentes necesidades de coordinación entre componentes. Los **cambios en la infraestructura** de ejecución y de comunicaciones tienen que ver con cambios en las plataformas hardware.

5.2 Solución PRISMA

En este apartado se presenta la solución arquitectónica PRISMA para el sistema EFTCoR y se muestra de qué modo PRISMA es capaz de cubrir todas las necesidades del sistema expuestos en el apartado 5.1.

A continuación se presenta la arquitectura correspondiente a EFTCoR y su especificación PRISMA incluyéndose en esta, tanto la definición de tipos utilizando el lenguaje de definición de componentes como la definición de la topología con el lenguaje de configuración:

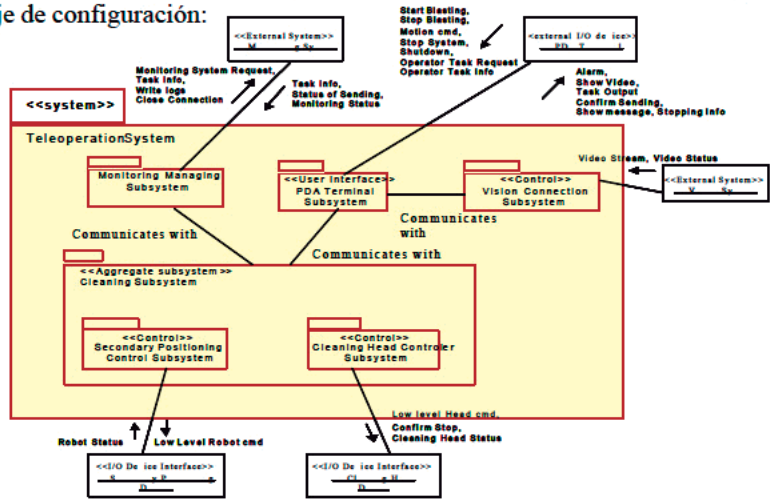


Fig. 3. Diagrama preliminar del diseño de la arquitectura

```
/*Definición de tipos*/
Interface Visual
   Services
      VideoStream;
      VideoStatus;
End_Interface Visual;

Interface Status
   Services
      RobotStatus;
End_Interface Status;

... ...

Interface Cmd
   Services
      LowLevelCmd;
End_Interface Cmd;
```

```
/*Definición de
Aspectos*/ … …
Component_type
PositioningControl
  InPort Port1:
Status;
  End InPort;
  OutPort Port2: Cmd;
  End OutPort;
  /*Aspectos*/
End_Component_type
PositioningControl;
… …
Component_type Vision
  InPort Port1:
Visual;
  End InPort;
  /*Aspectos*/
End_Component_type
Vision;
Connector_type
PDAVision
End_Connector_type
PDAVision;
… …
Connector_type
MonitoringCleaning
End_Connector_type
MonitoringCleaning;
System_type Cleaning
  Import Types
   Component Types:
PositioningControl,
  CleaningHead;
  End_Import Types;
/*Bindings*/
End_System_type
Cleaning;
System_type
TeleoperationSystem
  … …
End_System_type
TeleoperationSystem;
```

/*Definición Modelo Arquitectónico*/

```
Architectural Model
EFTCoR
  Import
CleaningHead,…
TeleoperationSystem;
  CH1: CleaningHead;
  … …
TeleoperationSystem;
  Attachements
TeleoperationSystem

PDATerminal1.OutPort1
>> PDAVision1.InRol1;

PDAVision1.OutRol1 >>
Vision1.InPort1;
  … …
End_Attachements
TeleoperationSystem;
  System TS1
    Inports
      Inport1:
PDAResponse;
      Inport2:
InMonitoring;
      Inport3: Status;
      Inport4: Visual;
    Outports
      OutPort1:
PDARequest;
      OutPort2:
OutMonitoring;
      OutPort3: Cmd;
      OutPort4:
Cleaning;
    Import Elements
CH1, PDATerminal1,…
    Import Connections
TeleoperationSystem;
    BindingLinks
      TS1.Inport1 >>
PDATerminal1.Inport1;
      … …
End_System TS1;
End Arquitectural
Model EFTCoR;
```

Existe un gran número de fuentes de variación dentro de un sistema tele-operación. PRISMA es capaz de tener en cuenta cada una de ellas, consiguiendo un modelo arquitectónico que satisface al 100% los requisitos de estos sistemas y las propiedades de abstracción y reutilización necesarias durante su proceso de desarrollo. Estos dos factores son mejorados gracias al concepto **sistema** de PRISMA, ya que permite construir componentes complejos a partir de otros más sencillos, permitiendo hacer un desarrollo iterativo e incremental. Un claro ejemplo es la especificación de EFTCoR que se presenta en este trabajo, ya que esta en proceso de elaboración inicial y el nivel de abstracción de la especificación es muy alto.

Una de las fuentes de variabilidad de los sistemas de teleoperación son los modos de operación. Los distintos modos de operación (normal, calibración, etc.) se corresponden en PRISMA con distintas interfaces de carácter funcional. La variabilidad en el modo de operación se puede conseguir en PRISMA reificando el tipo interfaz al meta-nivel, de forma que el meta-evento de cambio de interfaz en un

determinado puerto o rol del tipo, se puede invocar en tiempo de ejecución. La posibilidad de invocar meta-eventos en tiempo de ejecución permite la sustitución de tipos de forma dinámica y por tanto, la reconfiguración del modelo arquitectónico mientras se está ejecutando. La reificación de interfaces de carácter funcional al nivel meta, hace que el PRISMA de los sistemas de teleoperación tenga nivel base y nivel meta en su aspecto funcional.

Por otro lado, la ampliación de operaciones funcionales como el lavado, pintura, etc, implica la reificación del aspecto funcional y de las interfaces, para poder incluir nuevos servicios, y la reificación de los componentes, conectores y sistemas, para generar nuevos puertos o roles en aquellos casos que sea necesario generar nuevos canales de comunicación.

En el caso del cambio de proceso de mantenimiento y de sistema de posicionamiento, también es necesario la reificación del aspecto funcional. Pero esta vez no sólo con el objetivo de crear nuevos servicios, también de poder hacer todo tipo de modificaciones. Además, la fuente de variabilidad de cambio de proceso de mantenimiento, introduce cambios en la coordinación. Por lo que se hace necesario que el PRISMA también tenga meta-nivel en el aspecto de coordinación.

El alto nivel de abstracción de las especificaciones de modelos arquitectónicos PRISMA hace que éstas sean independientes de la tecnología aplicada. Por lo tanto, ante un cambio tecnológico, la especificación no se ve afectada, sólo el proceso de traducción automático de la especificación a código de la aplicación, evitando que los cambios de infraestructura sean una fuente de variabilidad.

Finalmente, el conjunto de fuentes de variabilidad y los distintos tipos de propiedades, que se han tenido en cuenta en este trabajo, de los sistemas de teleoperación, dan como resultado un tipo de PRISMA compuesto por los aspectos funcional y coordinación a nivel base y meta y los aspectos distribución y calidad únicamente a nivel base

6. Conclusiones y Trabajos Futuros

Se ha presentado un marco de trabajo para modelar y describir modelos arquitectónicos complejos, distribuidos, evolutivos y reutilizables llamado PRISMA, mediante la aplicación de un caso de estudio de ámbito industrial (EFTCoR). Dicho marco se caracteriza por la integración que realiza de las aproximaciones de desarrollo DSBC y DSOA y las propiedades reflexivas a través de un meta-nivel que le da capacidad de evolución.

Se ha mostrado que PRISMA es capaz de diseñar sistemas de teleoperación dinámicos, evolutivos y reutilizables, reduciendo el coste de diseño y mantenimiento de sus modelos arquitectónicos, gracias a que los componentes especificados se pueden reutilizar en diferentes arquitecturas.

Las líneas de trabajo futuras de cooperación entre ambos grupos de investigación (DSIE y ISSI) se van a centrar en la adaptación de los lenguajes PRISMA, para que sean capaces de soportar las nuevas fuentes de variabilidad que aparezcan en las siguientes fases de desarrollo del sistema EFTCoR.

Bibliografía

[1] Allen R, Douence R., Garlan D., Specifying and Analayzing Dynamic Software Architectures, Proceedings Conference on Fundamental Approaches to Software Engineering, volumen 1382 LNCS, Springer Verlang, Lisboa, Marzo, 1998

[2] Álvarez B., Iborra A., Alonso A., De La Puente J.A. Reference architecture for robot teleoperation: development details and practice use. Control Engineering Practice, April, 2001, pp. 395-402.

[3] Andrade L., Fiadeiro J., "Interconnecting Objects via Contracts". OOPSLA'99. Tutorial 56.

[4] Aspect Oriented Software Development, http://aosd.net

[5] Balzer R. A 15 Year Perspective on Automatic Programming. IEEE Transactions on Software Engineering, vol.11, num.11, pages 1257-1268, November 1985.

[6] Bracciali A., Brogi A., Canal C., "Sistematic Component Adaptation", In Proc IDEAS, La Habana, 2002.

[7] Canal C., Un Lenguaje para la Especificación y Validación de Arquitecturas Software, Tesis Doctoral, Universidad de Málaga, 2000.

[8] Carsí J. A., "OASIS como marco conceptual para la evolución del software". Tesis Doctoral, Universidad Politécnica de Valencia, 1999.

[9] Cuesta, C.E., Arquitectura de Software Dinámica Basada en Reflexión, Tesis Doctoral, Universidad de Valladolid, 2002.

[10] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794. 2002

[11] Proyecto FEDER (UPCT, Izar Carenas) nro FEDER IFD97-0823 (TAP) Robot Escalador para la limpieza de cascos de buques, respetuoso con el medio ambiente, 1997.

[12] Garlan D., Cheng S., Kompanek A.J., "Reconciling the Needs of Architectural Description with Object-Modeling Notations", Science of Computer Programming Journal, Special UML Edition, Elsevier Science, 2001.

[13] Iborra A., Pastor J.A., Álvarez B., Fernández C. and Fernández J.M. Operational Experiences using Robotics during Maintenance Services in PWR Nuclear Power Plants. IEEE Robotics&Automation Magazine.

[14] Jaén J., Ramos I., "A Conceptual Model for Context-Aware Dynamic Architectures", 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with ICDCS2003, May 19-22, 2003 Providence, Rhode Island USA.

[15] J. Jaen, J.H. Canos, A Grid Architecture for Building Hybrid Museums, 2nd International Human.Society@Internet Conference. To Appear at Springer-Verlag LNCS. Seul 2003 (pendiente de publicación)

[16] R. Kazman, M. Klein, P. Clemens, ATAM SM: Method for Architecture Evaluation, Technical Report, CMU/SEI-2000-TR-004, 2000.

[17] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto. Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721-663-0, 1998.

[18] Loques O., Sztajnberg A., Leite J., Lobosco, M., "*On the Integration of Meta-Level Programming and Configuration Programming*", In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[19] Monge R., Alves C. Vallecillo A., "*A Graphical Representation of COTS-based Software Architectures*", In Proc IDEAS, La Habana, 2002.

[20] McGurren F., Conroy D., "X-ADAPT: An Arquitecture for Dynamic Systems", Workshop on Component-Oriented Programming, ECOOP, Málaga, Spain, 2002.

[21] Oliva A., Garcia I.C., Buzato L.E., *"The Reflective Architecture of Guaraná"*, Technical Report IC-98-14. Instituto de computación, Universidad de Campiñas, abril, 1998.

[22] Object Management Group (OMG), http://www.omg.org

[23] Pérez J., Ramos I., Lorenzo A., Letelier P., Jaén J., PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos, Actas VII Jornadas de Ingeniería del Software y Bases de Datos, JISBD, Escorial (Madrid), noviembre 2002

[24] J. Pérez, I. Ramos, J. Jaén and P. Letelier, E. Navarro: "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 (Aceptado: Pendiente de Publicación)

Modelando con UML Arquitecturas Software Reflexivas desde la Aproximación Orientada a Aspectos: Un Caso de Estudio

A. Lorenzo, A.Peñarrubia, J.A. Carsí, I. Ramos

Departamento de Sistemas Informáticos y Computación

Universidad Politécnica de Valencia

{alorenzo, arpebla, pcarsi, iramos}@dsic.upv.es

Resumen

Bajo la convicción de que el concepto de Arquitectura Software es la verdadera realidad sobre la que podemos dar respuesta a las exigencias actuales del desarrollo software, los lenguajes de especificación de modelos conceptuales se han adaptado para dar un soporte formal no sólo a la definición de componentes y conectores sino que, su notación de alto nivel permite especificar las reglas y restricciones que aseguran la reconfiguración dinámica de estos elementos. Nuestro modelo es soportado por un Lenguaje de Descripción de Arquitectura (LDA) con capacidades reflexivas y que mantiene separadas estas dos facetas: *el Lenguaje de Definición* de entidades y *el Lenguaje de Configuración,* verdadero responsable del cambio de la topología de la arquitectura. La descripción del metanivel nos acerca al objetivo de la transparencia entre el desarrollo de la aplicación y el tratamiento independiente de los distintos aspectos ortogonales a la funcionalidad del sistema.

En este artículo se muestra cómo desde una aproximación de modelo y lenguaje se puede responder a las necesidades actuales que reclama *La Programación Orientada a Componentes* presentando el desarrollo completo de un caso de estudio sobre una de las actividades que está ganando fuerza dentro del comercio electrónico. Se trata de una apuesta de futuro fundamentada en los cambios impuestos por los avances tecnológicos y las necesidades del mercado al que abastece: *las subastas electrónicas bajo el modelo de web services*

Palabras clave: lenguaje de definición de arquitecturas software, lenguaje de configuración, reflexión, aspectos., componente y conector.

1.- Introducción

Las demandas, cada vez más complejas, del mundo de la empresa han determinado profundos cambios en la estructura de las aplicaciones software. La necesidad de rapidez en las respuestas exige un mercado global de componentes reutilizables y estándares abiertos que fijen las reglas y fronteras de las interacciones entre ellas. Por tanto, en el desarrollo actual de aplicaciones, la idea de *Descripción Arquitectónica* cobra especial importancia desde las primeras etapas del ciclo de vida como un marco de referencia para asegurar la correcta conexión entre los elementos arquitectónicos que darán lugar a la construcción de software de calidad.

Abordar esta idea desde la perspectiva de las Arquitecturas Software supone elegir un Lenguaje de Descripción Arquitectónica (ADL) capaz de modelar sistemas complejos, distribuidos y muy cambiantes. La mayoría de los ADL actuales [Med00] se han quedado en entornos de investigación y las *herramientas CASA* [Cue00] son poco conocidas en ambientes industriales. Con el fin de presentar un marco amigable de trabajo, en este artículo utilizaremos una extensión del estándar UML para arquitecturas

Lorenzo, M. A., Peñarrubia, A., Carsí, J.A., Ramos, I. Modelando con UML Arquitecturas Software Reflexivas desde la Aproximación Orientada a Aspectos: Un Caso de Estudio. Actas de las VI Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2003). 2003.

que se apoya. en una propuesta formal de modelo arquitectónico reflexivo bajo una aproximación Orientada a Aspectos: PRISMA [Per02].

Son muchos los autores que han escrito comparativas entre los conceptos UML y los elementos arquitectónicos ([Gar02],[Kob00]) y, aunque todos ellos coinciden en afirmar que las diferencias semánticas son claras, se están buscando soluciones para aproximar ambas notaciones ([Mon02],[Sel01]). En esta línea se presenta una solución que utiliza nuevos elementos para mapear los conceptos de PRISMA sobre la notación gráfica UML.

La aportación de este artículo consiste en presentar bajo una notación estándar un modelo Reflexivo y Orientado a Aspectos que fundamenta el estudio de un posible caso real (en este caso ficticio) que finalmente se implementa mediante *web services* [Sho02].

1.1 Atributos de calidad

Toda especificación de un sistema software requiere la definición de una serie de atributos de calidad. Éste es un proceso complicado, tanto por las relaciones existentes entre los mismos atributos, como por los intereses de los diferentes actores del sistema sobre éstos. Será necesaria una determinada política de negociación. Estos atributos se pueden clasificar en:

- **Requisitos técnicos del sistema:** derivados de la especificación propia del mismo. Se pueden clasificar a su vez en funcionales y no funcionales
- **Requisitos de negocio:** derivados de la estrategia empresarial entorno al producto.
- **Requisitos propios de la arquitectura:** independientes de los anteriores, para toda arquitectura, tales como la corrección o la completitud.

Para comprender qué aporta un modelo arquitectónico como PRISMA se debe ser consciente de la relevancia de los requisitos no funcionales en la especificación del sistema. Según [Bass98] los requisitos no funcionales son los siguientes:

- **Modificabilidad:** es la facilidad para realizar cambios en el sistema.
- **Portabilidad:** es la flexibilidad del sistema en cuanto a las plataformas sobre las que puede funcionar.
- **Reusabilidad:** es el grado en el que los diferentes componentes del sistema pueden ser reutilizados.
- **Integrabilidad:** es la posibilidad de interacción dentro del sistema de componentes desarrollados por separado.
- **Testeabilidad:** es la facilidad para encontrar fallos en el sistema mediante pruebas.

Así pues, se ve la necesidad de un diseño modular con cierto grado de flexibilidad, para afrontar el desarrollo de los sistemas actuales en base a los requisitos de calidad especificados para los mismos. En este contexto nace PRISMA, comprometiéndose precisamente con la especificación de dichos componentes, y de las interacciones entre los mismos. PRISMA asegura por tanto un modelo suficientemente potente para abordar una descripción derivada de los atributos de calidad anteriormente expuestos.

1.2 Comercio Electrónico y WEB services

El comercio electrónico se puede definir, en un sentido amplio, como cualquier forma de transacción o intercambio de información comercial basada en la transmisión de datos sobre redes de comunicación como Internet ([Vaz00]). En este sentido, el concepto de comercio electrónico no sólo incluye la compra y venta electrónica de bienes, sino también todo tipo de transacciones de información o servicios. Para cubrir las necesidades de mercado respecto a este último punto, Microsoft presenta en su plataforma .NET , junto con otras empresas, la filosofía de los *web services*. Un *web service* es una clase cuya interfaz se hace pública en un servidor WEB y se define con un lenguaje de descripción de interfaces WSDL[1].

Un cliente que accede a un *web service* debe resolver dos cuestiones:

- quien le oferta el servicio: búsqueda del proveedor (por UDDI o DISCO[2])
- cómo invocar la llamada al método: tipo de parámetros y su devolución

Para finalizar este apartado se presenta la estructura de este documento: en el punto 2 se realiza una exposición resumida de PRISMA y su lenguaje, a continuación en la sección 3 el ejemplo que hará de hilo conductor del tema. En el apartado 4 se muestran los conceptos del modelo sobre el caso práctico, en el apartado 5, la sintaxis y la representación gráfica sobre la notación UML. Por último, se presentan en las conclusiones los aportes de nuestro trabajo así como los temas todavía abiertos.

2.-El Modelo de PRISMA

Es una Plataforma Reflexiva ([Caz00]) para modelar arquitecturas basada en una aproximación Orientada a Aspectos. La arquitectura de un sistema queda definida cuando ([Sha96]):

- la funcionalidad global se divide en componentes,
- se describe la topología, es decir, los conectores entre esas componentes, y
- se determina la estrategia de coordinación entre componentes, en concreto, el comportamiento de los conectores.

Teniendo en cuenta estos puntos, PRISMA se apoya en las siguientes bases:

- ***Una aproximación funcional,*** para capturar la semántica de las actividades de un sistema ([Ana99]). Durante la fase de especificación del análisis, resulta más sencillo entender el sistema como un conjunto de servicios que se deben proporcionar que tratar de identificar las entidades del dominio que pueden implementar dichos servicios.
- ***Identificar los conectores*** bajo las características de los ***contratos*** ([And99]) enriquecidos con el concepto de coreografía. En [And99] las interacciones entre

[1] Web Service Description Lenguaje.

[2] Universal Description, Discovery and Integration. DISCOvery. Son protocolos de búsqueda de proveedor de servicios web

participantes deben ser modeladas como entidades de primera clase que capturan esa comunicación para dotarla de funcionalidad y coordinación específica. Por tanto, los conectores, como elementos de la arquitectura, comparten los aspectos funcionales y de distribución, pero su caracterización como conector la establece el aspecto de coordinación, en el que se definen:

  - las políticas de negocio, de manera que cambios en los requerimientos del negocio se traducen en cambios en la coordinación de las interacciones entre componentes, sin alterar sus propiedades individuales,
  - la coreografía que especifica el orden en que los mensajes entre los componentes se envían y reciben ([Mon02]),
  - las sincronizaciones que muestran la compatibilidad entre los tipos de las interfaces de las componentes que conecta.

- ***La aproximación Orientada a Aspectos.*** Tiene como objetivo eliminar el *crosscutting*[3], modularizando correctamente los distintos aspectos presentes en los elementos arquitectónicos. Dichos aspectos también se modelan como entidades de primer orden para asegurar su reutilización y su capacidad para caracterizar a los conectores.
- ***Los conceptos reflexivos*** sobre los que se fundamenta el modelo dinámico de la arquitectura en aquellos aspectos que se reifican. Así, mediante la reificación se genera en el metanivel la metaespecificación de las propiedades del nivel base que se quieren cambiar. La modificación se produce al activar los servicios del metanivel en el aspecto concreto sobre el que ocurre el cambio, que puede ser:
  - sobre la estructura y comportamiento de componentes o conectores
  - sobre la topología del sistema (las conexiones entre componentes)

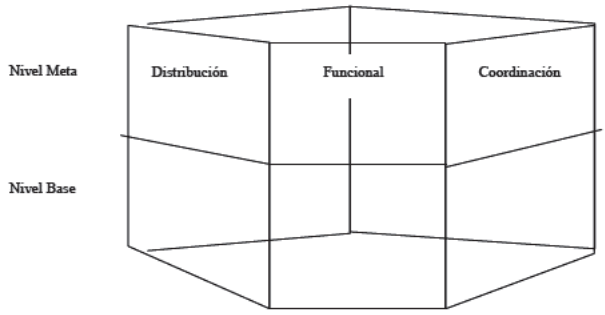


Fig 1: Prisma

Como muestra la Fig 1 se puede visualizar gráficamente cada uno de los elementos arquitectónicos (componentes, conectores y subsistemas) mediante un prisma en el que cada cara representa un aspecto (funcional, distribución, etc) y que a su vez se divide en dos niveles: base y meta. El metanivel recoge la semántica del dinamismo en aquellas caras del prisma que lo presentan.

Bajo PRISMA se define un Lenguaje de Descripción de Arquitectura con dos partes bien diferenciadas:

[3] Dispersión de un aspecto entre diferentes componentes

- E*l lenguaje de definición* de componentes y conectores, que desde la aproximación orientada a aspectos aborda, no solo la especificación de la funcionalidad del sistema, sino aspectos ortogonales a la misma: como la reutilización, la coordinación y especialmente la distribución.
- E*l lenguaje configuración,* que describe en tiempo de ejecución el aspecto emergente de coordinación a través de las sincronizaciones entre componentes y conectores que establecen de forma dinámica el concepto de representación en una arquitectura.

3.- Presentación del Caso de Estudio

En la ontología que maneja el comercio electrónico participan como actores principales las empresas, los consumidores y las administraciones públicas.

Para ilustrar el uso sobre el marco de trabajo de PRISMA se ha elegido un modelo de negocio B2C (Business to Consumer) y se ha desarrollado una aplicación para la gestión de una subasta WEB; un caso real que presenta un marco suficientemente amplio para dejar patente la existencia de una arquitectura en respuesta a:

- *la alta complejidad* en la *funcionalidad* por las intrincadas políticas de negociación entre participantes que deben responder en tiempo real y en la *estructura* con respecto a la organización de sus actividades, participantes y bienes.
- *el fuerte dinamismo* de cambio en las convenciones establecidas para la negociación.
- interesantes *perspectivas de distribución* enfocadas al comercio electrónico.

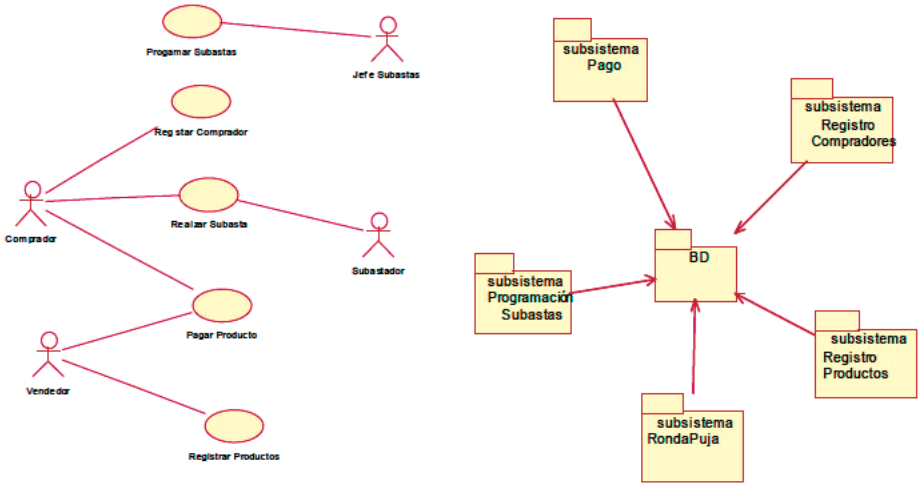
El modelo busca garantizar la equidad en una subasta a la baja, tan vivaz como las de una lonja, además de permitir la participación de compradores no localizados en el recinto en el que se realiza físicamente la subasta mediante la utilización de un sistema software distribuido.

Descripción del problema:

La institución está preparada para realizar las siguientes actividades a cuyo cargo hay un representante de la organización como coordinador de las tareas realizadas. La representación de los casos de uso en la Fig 2 recoge de forma gráfica los requisitos del sistema.

Aunque la Fig 2 recoge la funcionalidad completa del sistema, e incluye como casos de uso la inscripción sobre los formularios de compradores y productos, así como la facturación a clientes y proveedores, el desarrollo del ejemplo sólo se centrará en el tratamiento de la sesión de puja.

La Fig 3 presenta el comportamiento del subsistema RondaPuja. El diagrama de colaboración describe fielmente el proceso que tiene lugar: la sesión da comienzo a la hora fijada por la organización si hay un número mínimo de clientes, y termina al agotarse la mercancía o cuando no hay el número mínimo de participantes establecido.



(a) **(b)**

Fig 2: Modelo UML de una subasta

. La pantalla de los compradores refleja en todo momento el bien subastado y su precio actual así como el alias de todos los compradores participantes. Ante una puja, todos los terminales conectados identifican al pujador y el precio sobre el que se paró la ronda.

. El sistema otorga el género si el pujador figura con crédito suficiente.

. El resto de los participantes pueden también secundar esa misma puja para comprar unidades de ese mismo producto en riguroso orden de participación.

. Se actualizan los datos sobre los terminales, se elimina el producto adquirido y se actualiza el crédito de los compradores.

. Se puede abandonar la sesión en cualquier momento pasando por los formularios de caja cuando se ha adquirido algún bien.

. El proceso continúa animado por nuevas rebajas en el precio.

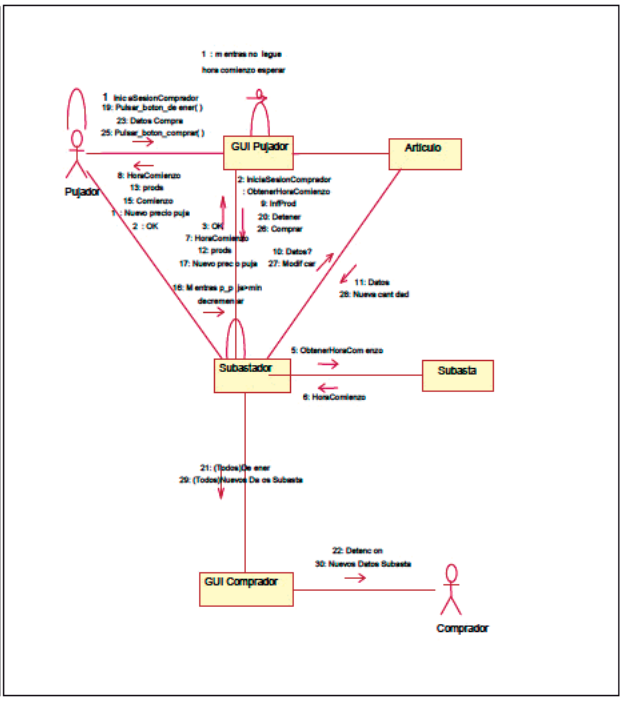


Fig 3: Comportamiento en RondaPuja

4.-Elementos Arquitectónicos en el Ejemplo Propuesto

Se presenta una vista resumida de los elementos propios de las Arquitecturas Software sobre PRISMA. Esta descripción permite establecer la partición de los elementos identificados en el sistema de ejemplo.

4.1 Componentes: Semántica y Composicionalidad

El concepto de componente es acuñado por las metodologías orientadas al comportamiento que modelan el sistema preocupándose por la funcionalidad de las entidades y la colaboración entre las mismas.

Componente: Es difícil llegar a un consenso en la definición de este concepto ([Shaw96],[MDA01]), pero manteniendo la idea marcada por los ADL's, lo definimos como una unidad de composición de aplicaciones software, que posee un conjunto de interfaces. Las interfaces representan el tipo de los puntos de unión entre componentes, o componentes y conectores que se llaman puertos. PRISMA añade a esta descripción dos conceptos orientados a la reutilización:

- la división en aspectos para un tratamiento modular e independiente de cada uno de ellos y,
- las características reflexivas que aseguran las posibilidades de cambio en cada aspecto.

En las fases propias del desarrollo basado en componentes (DBC) ([Iras00]) es importante el proceso de identificación de componentes que provoca decisiones sobre la arquitectura final del sistema. En la subasta identificamos dominios de negocio que representan parte de la infraestructura de la organización, contienen recursos clave y ofrecen servicios. Respecto a la descripción mostrada en la Fig 2b y, por cuestiones de espacio, solamente se profundiza en el subsistema RondaPuja donde se propone inicialmente la existencia de dos componente Participante y SubastalaBaja (ver Fig 4).

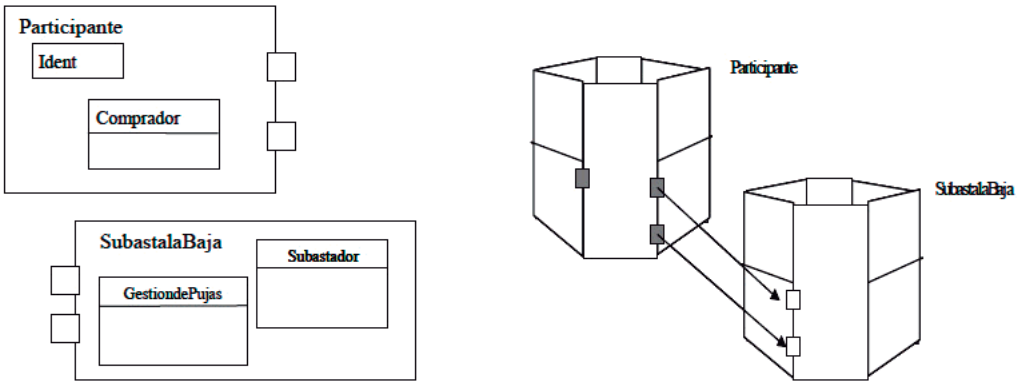


Fig 4: Componentes Participante y SubastalaBaja

Participante encapsula la funcionalidad de las interfaces de usuario (IU) y sobre las que los compradores distribuidos visualizan la situación actual de la ronda. La IU representa la información de los productos bajo subasta así como el precio actual marcado por el subastador en cada momento para su adquisición. Con el botón *comprar* el pujador para la ronda de bajada, evento que determina que la ronda quede detenida. En ese momento, el pujador puede iniciar la adquisición de género.

SubastalaBaja: representa la funcionalidad del *web service*. Su diseño demuestra cómo las decisiones arquitectónicas que se toman durante las primeras etapas del ciclo de vida no fuerzan una implementación concreta, pero le imponen muchas restricciones. Representa los servicios que ofrece la aplicación para gestionar el proceso a la baja de la subasta: determina la hora de la sesión, el precio de salida de los productos, el decremento entre pujas, detiene la subasta ante una puja y gestiona y actualiza las adquisiciones.

Interfaces: Representan el tipo de las conexiones entre componentes o, entre componentes y conectores. Los servicios provistos o requeridos en las componentes (puertos) son canalizados a través de los puntos de entrada y salida de los conectores (roles). Por tanto, los puertos y roles unidos comparten el mismo tipo de interfaz tanto a nivel de signatura como de comportamiento (ver Fig 5).

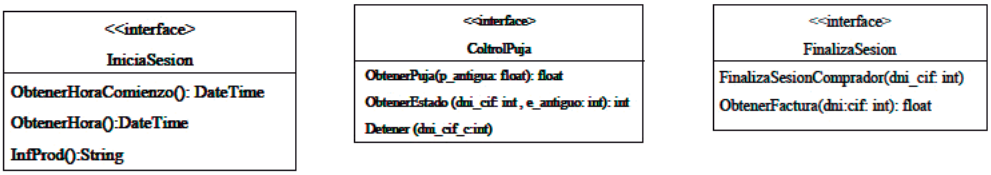


Fig 5: Interfaces entre SubastalaBaja y Participante

4.2 Conectores

Definen las conexiones entre dos o más componentes o entre componentes y subsistemas. Comparten con las componentes su características de ser entidades de primer orden, de agregar aspectos y de tener interfaces que se instancian en sus roles. Sus funciones son claras a un nivel conceptual pero como dice Monge [Mon02] la noción de conector desaparece en el DBC y es aún más difícil diferenciarlo a nivel de implementación. Por ejemplo, su aspecto de distribución se resuelve en los servicios distribuidos del *middleware* que use la tecnología concreta.

A nivel conceptual siempre representaremos los conectores. Estos pueden indicar una simple comunicación entre componentes o bien, propiedades especiales de la interacción que se captura especificando atributos y comportamiento. Con respecto al ejemplo, la unión entre el participante y la sesión de puja tendrá que ser atendida de forma especial cuando el comprador quiera participar desde su terminal en subastas diferentes manteniendo distintas ventanas abiertas por cada una de las sesiones en las que participa. Sin embargo, ante un participante en una única sesión el conector que los comunica realizará una labor mucho más sencilla

4.3 Sistema

Para PRISMA un sistema es un componente formado, al menos, por la composición de dos o más componentes unidos o no mediante conectores.

Recordando la representación dada en la Fig 4, *RondaPuja,* resulta de la composición de Participante y SubastalaBaja en su versión de implementación sencilla, sin conectores (el comprador sólo participa en una subasta). La funcionalidad emergente que surge de la composición queda reflejada en la descripción del subsistema *RondaPuja* con los mismos aspectos (funcional y distribución) que un componente, asociando las interfaces internas entre si y con el subsistema final.

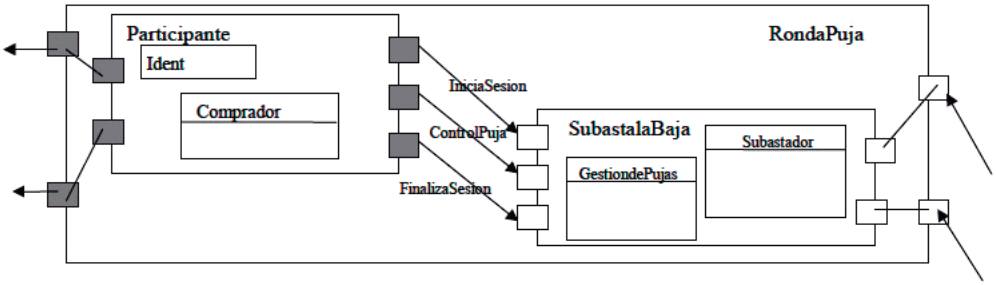


Fig 6: El sistema como componente

Como indica la Fig 6 los puertos de la nueva componente (subsistema) mostrarán los servicios requeridos y provistos para completar la unión con el resto de las componentes y conseguir la funcionalidad completa de la aplicación.

Es decir, la sesión de puja sólo puede comenzar después de haber un control de registro de compradores y una gestión de los productos inscritos por los vendedores (que son los servicios requeridos del subsistema). Por otra parte, cuando los compradores abandonan la ronda, o al término de la misma, el subsistema proporcionará resultados, como por ejemplo la facturación para el cobro de las adquisiciones de compradores y la contabilidad para pagar a los proveedores (servicios provistos).

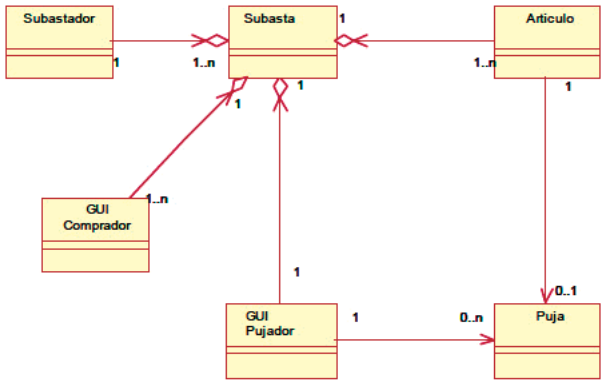


Fig 7: Diagrama de clases de la escena de la ronda.

Si en la figura 6 se puede observar la división del sistema en componentes, y las relaciones entre las mismas. En la figura 7 se pueden ver las clases que aparecen en el mismo. Es este doble nivel de diseño el que nos garantiza la alcanzabilidad del compromiso establecido con los requisitos de calidad. Por ejemplo, mientras la clase Articulo nos garantiza cierto grado de reusabilidad, portabilidad, modificabilidad, e incluso testeabilidad, al ocuparse de la parte de acceso a la base de datos para gestión de productos (altas, bajas...), permitiendo adaptar el sistema a otra base de datos sin alterar el código del resto de clases, y acotando el alcance de los errores derivados de los accesos, la integración de la misma en un componente u otro nos permitirá balancear el diseño. Por otra parte, es a través de esa comunicación entre componentes donde se expresan aspectos del sistema tales como el grado de sincronización, característicos de sistemas con requerimientos temporales como los del ejemplo. El diagrama de la figura 6 evidencia la necesidad de sincronización existente entre Participante y SubastalaBaja, con métodos como ObtenerPuja u ObtenerEstado que cumplen la misión de dar a conocer a los participantes las alteraciones de estos atributos. Es posible, sin embargo, ,

afrontar ese sincronismo de otras formas, como por ejemplo mediante consulta periódica por parte del participante. Evidentemente, la distribución de las clases dentro de uno u otro componente, influye en la complejidad de las interfaces y la densidad de las comunicaciones, otro factor a tener en cuenta a la hora de llevar a cabo el diseño.

5 Especificación PRISMA para el Caso de Estudio

A continuación se presenta la codificación del ejemplo propuesto. Se describen primero las entidades con el lenguaje de definición, cuya sintaxis permite definir componentes, conectores y subsistemas a partir de la especificación de sus aspectos.

Dada la descripción de los aspectos y de los distintos tipos de interfaces que se necesitan para su definición, se pueden ahora describir tipos de componentes y conectores.

Component_Type Participante **Outport** Participant.OutP1:I_IniciaSesion Participant.OutP2: I_ControlPuja Participant.OutP3: I_FinalizaSesion **End Outport** $OutP_i$)+ FAParticipante DAPArticipante **End Component_Type** Participante **Component_Type** *SubastalaBaja* **Inport** SubastalaBaja_InP1: I_IniciSesion SubastalaBaja_InP2: I_ControlPuja SubastalaBaja_InP1: I_FinalizaSesion **EndInport** FAAuctioner **End Component_Type** *SubastalaBaja*	**Connector_Type** *Conect_Participante_SubastalaBaja* **InRol** Conect_Participante_SubastalaBaja_InR1: Conect_I_IniciaSesion Conect_Participante_SubastalaBaja_InR2: Conect_I_ControlPuja Conect_Participante_SubastalaBaja_InR3: Conect_I_FinalizaSesion **EndInRol** **OutRol** Conect_Participante_SubastalaBaja_OutR1: Conect_I_IniciaSesion Conect_Participante_SubastalaBaja_OutR2: Conect_I_ControlPuja Conect_Participante_SubastalaBaja_OutR3: Conect_I_FinalizaSe **End OutRol** $OutR_i$ [Connected Elements_Type Participant, SubastalaBaja [Coi: Coordination Aspect;] [F_i: Functional Aspect;] [D_i: Distribution Aspect;] **End Connector_Type**
/*definición de Interfaces*/ **Funtional Interface** *FI_Participante* *with SubastalaBaja* *I_IniciaSesion* ObtenerHoraComienzo(): DateTime ..... *I_ControlPuja* ..... *I_FinalizaSesion* ..... *with BD* AltaLote(lote: String, proveedor: String) ..... **EndFuntional Interface** *FI_Participante* **Funtional Interface** *FI_SubastalaBaja* **End Funtional Interface** *FI_SubastalaBaja* **Distribution Interface** *DI_Participante* **EndDistribution Interface** *DI_Participante* **Coordinational Interface** *CI_Conect_Participante_SubatalaBaja* Conect_I_IniciaSesion Conect_I_ControlPuja Conect_I_FinalizaSesion **EndCoordinational Interface**	**Functional Aspect** FA_Participante FI_Participante /* Composición paralelaventana Comprador y ventanaIdent */ **End Functional Aspect** FA_Participante **Functional Aspect** FA_SubastalaBaja FI_SubastalaBaja /*composición paralela GestiondePujas y Subastador*/ **End Functional Aspect** FA_SubastalaBaja **Functional Aspect** FA_BD FIBD /* Composición paralela Lote y Comprador **End Functional Aspect** FA_DB **Distribution Aspect** DParticipante DI_Participante **variable attributes** url:URL **events** **End Distribution Aspect** DParticipante

Definir la configuración del sistema RondaPuja supone establecer, en tiempo de ejecución, la estructura de las interacciones entre los componentes que permiten poner en funcionamiento una sesión de puja .

En el primero de los bloques anteriores vemos la especificación de los dos componentes, Participante y SubastalaBaja, detallándose los puertos de entrada y de salida de cada uno. Todo esto lo veíamos gráficamente en la Fig 6. En la implementación resultante mediante Web Services, los componentes esenciales (Ident, Comprador ...), se corresponden con clases, mientras que el componente participante tan sólo sirve para encapsular a los anteriores con objeto de simplificar la especificación , perdiéndose por tanto en la implementación. Del mismo modo, los tres puertos que se detallan, son a su vez agrupaciones de métodos que constituirán la interfaz de la clase, que será publicada en WSDL para las clases que lo necesiten (RondadePuja, Subastador, que son los verdaderos web services). En el segundo bloque se definen los conectores, (en este caso sólo uno) este concepto se traduce en la implementación en visibilidad de métodos entre clases, y por tanto en invocaciones. Finalmente se especifican los aspectos de distribución, coordinación y funcional de a partir de los componentes y conectores anteriores.

6.- Conclusiones

La idea de arquitectura de un sistema complejo no limita la implementación, pero impone muchas restricciones a la misma. Por ello, sus parámetros deben ser definidos y validados en las primeras fases del desarrollo software, y con este propósito es imprescindible la utilización de modelos que permitan representar fielmente las necesidades del software. El uso de notaciones formales aplicadas a las arquitecturas (ADL) ha sido de uso prácticamente exclusivo en ambientes de investigación y, sin embargo, son la única forma de abordar de manera fiable las necesidades que hoy plantean los sistemas.

El objetivo inmediato es suavizar el rigor semántico de estos lenguajes a través de herramientas visuales con las que los desarrolladores están familiarizados y generar automáticamente el código que hoy reclama la programación sobre la red. Con este artículo se abre un camino dirigido a desarrollar una herramienta para la especificación integral de la arquitectura que permitirá su ejecución y el uso de componentes reutilizables. Con el ejemplo expuesto se demuestra que aplicar un modelo de arquitectura adecuado a un problema concreto asegura una implementación de calidad, correcta, eficiente y preparada para ser ejecutada sobre los sistemas distribuidos actuales. Además, el objetivo del diseño se orienta a la utilización de componentes ya creados o, a la generación de estos para ser reutilizados reduciendo los costes de adaptabilidad, evolución y mantenimiento.

Referencias

[Ana99] Anaya R. "Desarrollo y gestión de componentes reutilizables n el marco de OASIS" PhD Tesis: Universidad Politécnica de Valencia. Octubre 1999

[And99] Andrade L., Fiadeiro J. *"Interconnecting Objects via Contracts"*, in UML'99 -Beyond the Standard, R.France and B.Rumpe (eds), LNCS 1723, pp.566-583, Springer Verlag

[Bass98] Len Bass, Paul Clements and Rick Kazman "Software Architecture in Practice". Addison-Wesley 1998

[Car99] Carsi J.A. *OASIS como Marco Conceptual para la Evolución del Software* PhD Tesis: Universidad Politécnica de Valencia. Mar 1999

[Caz00] Cazzola W., Robert J. Stroud. *Mobility, Reflection and Meta-Objects Protocols*. En Proc. of the ECOOP'00 Workshop on Reflective Architecture, June 2000

[Cue00] Cuesta C. E, Fuente P. y Barrio-Solórzano M. *Arquitecturas del Software Dinámicas Basadas en Reflexión.* V Jornadas Ingenieria del Software y Bases de Datos. Pag 203-214. Valladolid, Noviembre 2000

[Gar95] Garlan D.y Perry D. E. *Introduction to the Special Issue on Software Architecture*. IEEE Transactions on Software Engineering, 21(4), Abril 1995.

[Gar02] Garlan D, Chen Shang-Wen, Kompanek *A.. Reconciling the Needs of Architectural Descriptions with Object-Modeling Notation* . To appear in Science of Computer Programing,2002,

[Geo98] Georgis A., Papadopoulos y Farhad Arbab. *Coordination models and languages*. En The Engineering of Large Systems, volume 46 of Advances in Computers. Academic Press, Agosto 1998

[Iras00] Irastorza A., Jaime A., Díaz O.,*"Diseño Basado en Componentes: Alternativas en el paso de Partición"*. Taller de componentes Distribuidas, V Jornadas Ingeniería del Software y Bases de Datos. Valladolid, Noviembre 2000

[Kob00] Kobrym C. *Modeling Components and Frameworks with UML*. Comunications of ACM, 43(10);31-38. Octubre 2000

[Let00] Letelier P.. *Animación Automática de Especificaciones OASIS usando Programación Lógica Concurrente*. PhD Tesis: Universidad Politécnica de Valencia. Marzo 2000

[Loq00] Loques, O., Sztajnberg, A., Leite, J. and Lobosco, M., "On the Integration of Meta-Level Programming and Configuration Programming", In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[McA95] McAffer J,. *Meta-level Programming with Cod A*. In ECOOP'95, pages 190..214, Aug1995.

[Med00] Medvidivic N., Taylor R.N. *A classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transaction on Software Engineering vol26, nº1 Enero2000

[Mon02] Monge R., Alves C. Vallecillo A. *A Graphical Representation of COTS-based Software Architectures*. In Proc IDEAS 2002 La Habana

[Nor99] Noriega P. *Agent mediated auctions: The Fishmarket Metaphor*. Ph.D. Thesis Universitat Autónoma de Barcelona 1999

[Oli98] Oliva A., Garcia I.C., Buzato L.E. *The reflective Architecture of Guaraná.* Technical Report IC-98-14. Instituto de computación de la Universidad de Campiñas. Abril1998

[Per02] Pérez J., Ramos I., Lorenzo A., Letelier P., Jaén J., "PRISMA: Plataforma OASIS para modelos Arquitectónicos". Jornadas de Ingeniería del Software. El Escorial (Madrid) 2002.

[MDA01] OMG. *Model Driven Architecture. A technical Perspective*. OMG document ab/2001-01-01

[Sel01] Selic B. *On Modeling Architectural Structures with UML*. In proc of UML'2001 Workshop on Software Architecture an Requirements Engineering (STRAW'01), Toronto, Canada, May2001

[Sel98] Selic B. Rumbaugh J. *Using UML for Modeling Complex real –Time System.* Disponible en http//rationl.com/media/whitepapers/umlrt.pdf, March 1998

[Sha96] Shaw M., Garlan D. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall 1996

[Sho02] Building XML Web services for the Microsoft .NET platform. Scott Short. Microsoft Press

[Tan96] Tanir O.. *Modelling Complex Computer and Communication Systems: A Domain-Oriented Design Framework*. McGraw-Hill 1996.

[Vaz00] Vázquez E., Berrocal J."Comercio electrónico: Una Visión General". Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid http://greco.dit.upm.es/~enrique/ce/parresum.html**,** 2000

Requirements and Architecture: a marriage for Quality Assurance

Elena Navarro and Isidro Ramos[†]

Department of Computer Science,
University of Castilla La Mancha, Albacete, Spain
enavarro@info ab.uclm.es

[†]Department of Information Systems and Computation
Politechnic University of Valencia, Valencia, Spain
iramos@dsic.upv.es

Abstract. Nowadays, software has quality as a goal over its lifecycle, from its inception to its completion. Several factors, methods and/or processes can be used to cope with this issue. Some of them assessed the quality in terms of customer satisfaction. Such approach highlights how needed the elaboration of high quality requirements specifications is. They should in turn produce systems that are more likely to perform as the stakeholder's expectation.

Architectural specifications act like a bridge between requirements and implementation. They provide systems models with a high abstraction level. Moreover, they allow to reason whether a system satisfies its requirements and therefore to determine the quality shown by the system. But a problem appears when the system is built because new and changed requirements may arise and the system needs to evolve.

This paper sketches our work in progress in this field in which we are concerned about a methodology to guide the reflexive architecture development from the software requirements. This methodology also allows one to identify and define the involved aspects in the final software architecture. The intermediate software artefacts are formalized in order to validate them and achieve a highly automatic process. Additionally, software updates are easily achieved because architectures can be directly derived from the requirements, with a more expectations centric view.

1. Introduction

There is no standard definition of Software Quality. Crosby [5] defines quality as "conformance to requirements", where requirements are customer defined services and constraints. They are the foundation on which the system to be should be implemented. They are the basic resource to achieve customer satisfaction. This view highlights the great significance that requirements have with regard to quality.

Several definitions for Software Architecture (SA) have appeared over time. Perry and Wolf [20] state that "architecture is concerned with the selection of architectural

Navarro, E., Ramos, I. Requirements and Architecture: a marriage for Quality Assurance. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003.

elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design". This statement shows the SA proceeds as a bridge between requirements and implementation. They allow one to reason whether a system meets its requirements and therefore to assess the quality shown by the system. But a problem arises when the requirements evolve. A dissimilarity can appear between the requirements and the architecture.

A methodology ATRIUM (Architecture generaTed from RequIrements applying a Unified Methodology, previously sketched in [16]) and a process are presented in this paper in order to illustrate this synergy between requirements and architecture. It is intended to be flexible enough to deal with concurrently evolving requirements artefacts and architectures, in the way Nuseibeh proposes [17], acquiring and maintaining complex relationships between them.

This work is organized as follows: Motivation and related work is presented in Section 2. Section 3 discusses details of our approach. Conclusions round up the paper in Section 4.

2. Motivation

Quality arises from several key points related to the software development and maintenance process. Several factors, methods and/or process can be used to cope with this issue. Above, we stated that requirements are directly related to the quality in terms of customer satisfaction. Such approach highlights how essential the elaboration of high quality requirements specifications is. They should in turn produce systems that are more likely to perform as the stakeholder's expectation. It is known that formal specifications show a major difference with respect to other approaches in providing increased quality to the requirements specification. In such a way, they can be validated assuring a correct (not ambiguous, not contradictory) specification.

Nowadays, automation is becoming one of the principal means to achieve a greater productivity and a higher product quality. This is due to the trend of increasing acquirer's satisfaction of the developed and delivered software product. The formal specification are not just meaningful to validate the requirements but they are also useful in the automation process. These formal specification can be compiled into models closer to the system to be following the paradigm of automatic programming. It provides an additional advantage in terms of reuse a quality solution.

Another question, related to the quality, entails the known *Non Functional Requirements*, i.e, requirements as modifiability, performance, maintainability and the like which can be demanded for the concrete systems. Approaches as [4] consider that quality attributes can be articulated as non functional requirements. It seems appropriated to include them in the end system, properly encompassed by the system architecture. But they must not be left out of any kind of formal treatment. On the contrary, they ought to be formalized allowing a clear translation into the final system.

Process quality models, such as CMM [18] or ISO/IEC TR 15504 [11], define how the software process should be enacted. They emphasize that quality of a developed product is influenced by the quality of the production process. The definition of formalized processes, with well standardized artefacts, is necessary to achieve an advantage in terms of quality. Moreover, these processes ought to have associated tool support. CASE tools offer potential benefits to supply high quality artefacts, speed up the development process, etc. They provide support for the automatic or semiautomatic techniques involved into the development process.

Therefore, a clear challenge arises from the previous ideas: how a process, sufficiently formalized and flexible, can be defined to acquire and compile both functional and non functional requirements into architectural models, and at the same time catching and managing their concurrent evolution over time. A guided process appears as the best approach to achieve this goal, leading to an increased productivity and a quality solution. This is the main aim of ATRIUM.

2.1 Related work

The relationship between requirements and architectures has recently received increased attention. In this sense, the international workshop STRAW'01 (From SofTware Requirements to Architectures [3]) has emerged and is in its second edition.

Brandozzi and Perry [2] have suggested the use of intermediate descriptions between requirements and architecture known as "architectural prescriptions". These describe the mappings between requirements aspects and architectural descriptions. Nevertheless, it is not clear that the overhead of an additional description is practical. Recent work on software product lines and system families has also examined the relationship between architectures and requirements. The work has focused on identifying core requirements (identified through a process of requirement priorization) and linking them to core architectures (identified by examining the stability of various architectural attributes over time).

Wile [26] has studied the relationships between specific classes of requirements and their equivalent dynamic architectures. They will allow requirements engineers to monitor running systems and their compliance with these requirements. The focus of this work is runtime monitoring, not the more traditional development activity.

Hall et al [9] show an extension to Problem Frames. They define requirements as dependencies among the system and the world outside of the computer. They help the developer to focus on the problem domain, establishing a clear separation between the world and the machine. It faces the problem by establishing a mapping among the structure of the problem domain and the solution domain. This solution is amenable to iterative development due to the interaction among requirements, architecture, and design: each one informing about the others and vice versa.

Grunbaccher et al [8] explore the relationships between software requirements and architectures, and propose an intermediate model, *CBSP* (Component Bus System – Property), to depict the dependencies among the key architectural elements and the stated system requirements. These dependencies are established in a polling process where the involved architects classify each requirement with respect to its

architectural relevance. Based on the CBSP model elements a proto architecture can be derived with the selection of the appropriate architectural style. Therefore, it is a systematic approach, not an automated one, that is only applied in a specific context.

Liu et al [14] propose a similar approach by combining scenarios and goals based models during architectural design. The GRL language is used to define the latter and the UCM notation is used for the former. The main problem that we observe is that both models are loosely coupled resulting eventually in a lack of coherence between them.

Finally, Rashid et al [21] have proposed a model for aspect oriented requirements engineering that identifies the concerns in a early step, jointly with requirements. This model allows to map and establish the influence on later development stages, where they can appear or not as an aspect of the end system. But it is focused on aspects for artefacts at design stages and not architectural aspects.

The main advantage shown by our approach with respect to the others is related to the guided and semi automated process support given from requirements to architectures. Also the coupling of requirements artefacts provides a better assessment of the design alternatives. Moreover, due to the traceability among requirements and architectures, along with the reflexive properties of PRISMA (PlatafoRma oasIs para Modelos Arquitectónicos, OasIs Platform for Architectural Models [19]), the selected model to specify the architectural artefacts, running systems can evolve over time while remaining compliant with their evolving requirements.

3. Our proposal

In order to guide the architects a methodology ATRIUM (Architecture generaTed from RequIrements applying a Unified Methodology), is proposed in this paper. ATRIUM is flexible enough to deal with concurrently evolving requirements and architectural models, as Nuseibeh´s proposal [17], acquiring and maintaining complex relationships between the different requirements artefacts and architectures. It guides throughout the process, from an initial set of requirements enough expressive and formalized as for their automatic compilation (see section 3.1) to the instantiated architecture. PRISMA is the selected model to specify the architecture due to its ability to deal with dynamic software architectures (see section 3.2).

3.1 The emerging Artefacts in the Requirements Engineering phase

In the context of Requirements Engineering, goal driven [6] and scenario based [15] approaches have proven being useful to elicit and define requirements. The former describes the high level objectives that the system to be should achieve; additionally, it provides a way of directly establishing quality attributes as goals of the target system. The latter is proposed for describing and reasoning about large grained behaviour patterns in systems, as well as the coupling of these patterns.

ATRIUM combines these two approaches, by defining a Goals Model (GM) and a Scenarios Model (SM), in order to overcome some of their deficiencies and limitations when used in isolation. Several issues arise in this coupling due to the

overlap and gaps between them and making difficult their interactions [22]. The reason for this union is that it could help us to define not only functional requirements but also non functional requirements. The latter are especially important for the way quality attributes can be directly included in the system to be.

This coupling is similar to the CREWS l'Escritorie approach [22] where scenarios are used to operationalise goals. The difference showed by our approach is the higher abstraction level that we deal with, where scenarios are used to acquire requirements that have an especial meaning for architectures. It provides us maintainability and traceability among both models. The generated models are formalized in order to validate their definition, verify their completeness and provide a specific automation alternative with a significant quality increment.

3.2 PRISMA: a model for dynamic software architecture

PRISMA has been defined as an extension of a formal language called OASIS [13], to provide semantic expressiveness for architectural models. PRISMA is being developed to provide a platform for modeling software architectures using visual languages, hiding the underlying use of formalisms and allowing the automatic generation of code. PRISMA is intended to design a great diversity of complex systems with a highly dynamic nature.

In PRISMA, types specifications include components, connectors, interfaces and aspects, which are first order citizen (Fig. 5). Components are compositional units that provide specific functionality to the system. Ports are their connection points allowing their interaction with the rest of the system and whose types are interfaces. Connectors have connection points called roles and typed by an interface. They are defined in terms of interaction among components and other connectors and so providing a high cohesion and a loose coupling. Moreover, the separation of concerns, together with the difference among components and connectors, facilitates maintenance and reuse of large and complex software systems.

Components and connectors are defined in terms of aspects composition: functional, distribution, coordination, etc, following the Aspect Oriented Software Development [1] approach. Crosscutting is handled because the integration is carried out at a high abstraction level, generating separate code for each aspect. They are defined using a Component Definition Language (CDL) that supports reflexive properties in order to fulfil highly dynamic architectures. It lets to reify any linguistic construction, that could evolve, to the meta level.

Furthermore, a Configuration Language (CL) is introduced to specify an architectural model defining the topology and the instantiated types. This language has also reflexive properties allowing a dynamic reconfiguration at run time. CDL and CL can be compiled to automatically generate code following the Automatic Programming Paradigm.

3.3 ATRIUM: Architecture generaTed from RequIrements applying a Unified Methodology

ATRIUM is defined supporting an iterative and incremental process that entails a set of 5 steps as shown bellow (see Fig. 1). They guide the analyst from an informal set of requirements, the starting point, to a target instantiated PRISMA model. Every step's output artefact is the input for the following step.

Grid Applications support a new model of computing that raises new challenges with respect the heterogeneity of the coordinated processes, the volume and volatility of the shared information and the potentially vast number of processes that require an homogeneous way of communication. Moreover, several works [12] have recognized their needs for dynamic self adaptation to deal with resource variability, system faults, etc. A previous work [12] showed how a set of Grid Computing Nodes exhibit some needs in terms of efficiency as CPU load, number of waiting processes, or available memory. Grid Computing Nodes gather this context information. An efficient task scheduling is needed in order to optimise the available resources by adding new Nodes when they are needed, i.e., self configurable or auto adaptive applications which perform architectural evolution based on contextual information (*ContextAware*). We will exploit this example to illustrate how ATRIUM is applied.

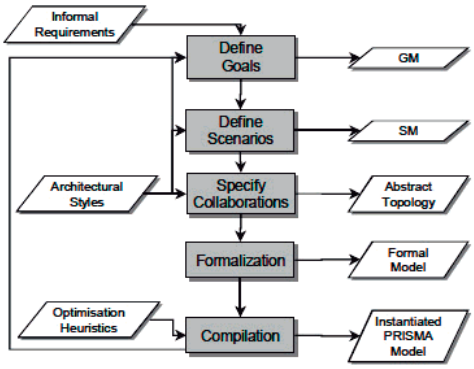


Fig. 1 ATRIUM: a description about its entailed steps

Step 1.- Define the Goals model (GM)

In this step, the GM is defined by the formulation of goals to be accomplished by the software system. They are directly related to both *Functional* and *Non Functional requirements*, such as performance, reusability, interoperability, etc. In this approach, we are specifically highlighting the latter because their meaning in terms of software quality. Moreover, they have a direct impact on the aspects which form the components.

The GM is made up of *goals* and *relationships* between them (Fig. 2). A *goal* is defined as an objective that the system to be should achieve and is stereotyped as Functional or Non Functional according to their type. A *relationship* over goals G1 and G2, is established when G2 is considered as a subgoal, i.e., a goal that has to be (AND relationship) or could be (OR relationship) satisfied in order to achieve G1.

Moreover, a CONFLICT relationship can be established among two goals (see [4] for more details about Goal Models). The GM model is operationalised at the next step by means of the Scenarios Model. Furthermore, the analysis provided by the SM will help us to refine the GM at the next iteration because this task is not as straight forward as to capture all the goals at once.

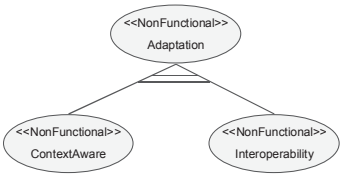


Fig. 2 Goals Model (part of) where ContextAware and Interoperability are two required subgoals (AND relationship) to satisfy the goal *Adaptation*.

Step 2.- Define the Scenarios Model (SM)
The Scenarios Model captures which elements are collaborating to satisfy the established goals in the GM, what their responsibilities are into this task and how their interactions occur to achieve them. Therefore, this model operationalises the GM by setting the different responsibilities of the elements in the system.

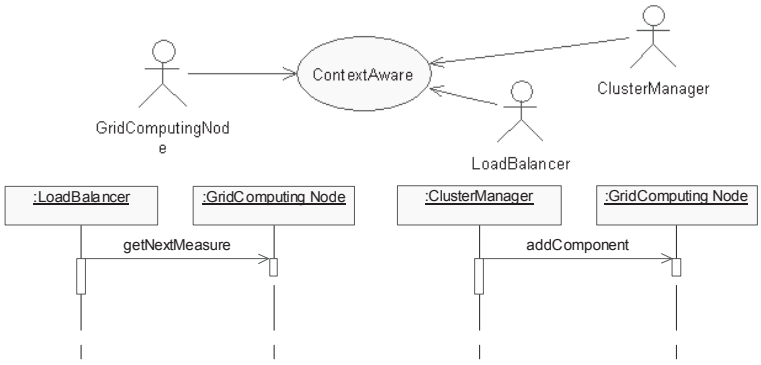


Fig. 3 Scenarios Model (part of) where *GridComputingNode*, *LoadBalancer* and *ClusterManager* are the components which operationalise the goal *ContextAware*. *LoadBalancer* interacts with *GridComputingNode* to get measures in order to optimise the available resources.

SM is made up of two diagrams (see Fig. 3): Use Cases and Message Sequence Charts (MSC) [25] (the latter are being incorporated into UML 2.0). The former provide us with a visual metaphor notation for scenarios. It establishes which goal, from the GM, is going to be operationalised, i.e., each goal in the goal graph is in agreement with an use case in the SM, addressing the traceability problem between the two models. In this case, the components or subsystems, that will appear in the system to be, play the role of the involved actors.

The MSC will represent the temporal sequences of interaction events. When we speak about software architecture [7], we have to identify not only the involved elements but also how they interact. MSCs identify the choreography, i.e. the coordination of the interacting elements, through their construction process.

As was stated above, components (and connectors) are constructed by a gluing of aspects. Therefore, it is necessary to identify what aspects are and what components (and connectors) are cut by them. In this respect, the higher level goals, defined in the GM, establish the concerns of the system to be. This allows one to define which aspect appear on each component by identifying the scenarios where it is involved and, consequently, the operationalised goal. For instance, *ContextAware* will be an aspect for *GridComputingNode* (see Fig. 3).

Step 3.- Define collaborations

The third step identify the collaborations among the identified components and subsystems. The collaborations realize the use cases through collaboration diagrams. The objective is to obtain an abstract topology (see Fig. 4) for the concrete system by using related architectural styles [23] and the architect's ability. Additionally, the connectors are defined according to the components interaction. These first class citizens in PRISMA are required to allow a loose coupling between the components.

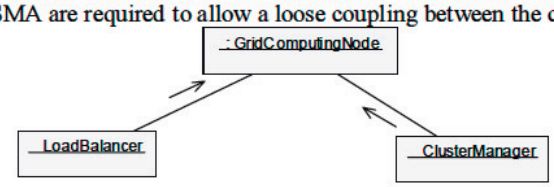


Fig. 4 The abstract topology provides a view of the interacting components.

Step 4.- Formalization

Both models and the abstract topology have to be analysed and semantically checked. It detects whether there are conflicts and what the best alternative is in order to avoid (or minimize) them. The set of artefacts, defined in previous steps, will be specified in the 3APL language [10], whose interpreter will help us to validate them. A set of derivation rules are provided to generate a formal specification from scenarios, goals and collaborations, in order to encourage and speed up the formalization process. This specification is validated and then used for its automatic compilation in the next step.

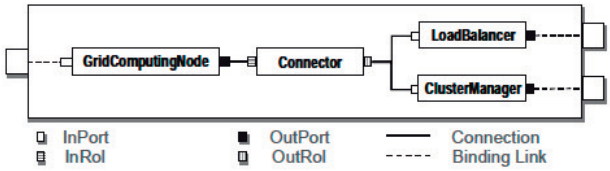


Fig 5. An instantiated PRISMA model for a Grid System.

Step 5.- Compilation

Using the formal model and a set of heuristics, the translation from the requirements model to an instantiated PRISMA model is accomplished (see Fig. 5). This architectural model will be compilable in a concrete target system having reflexive

properties. This step is assisted by the engineer, following an iterative refinement process.

Due to ATRIUM is intended to be iterative and incremental, a feedback will be provided to the Step 1. It allows us to have the overall models up to date, all over the process.

4. Conclusions and Future Works

In this paper we show how to address the iterative development of requirements and architectures during the development of software systems. A methodology, ATRIUM, that guides the analyst from an initial set of requirements to an instantiated architecture has been presented. It uses the strength provided by the coupling of scenarios and goals to systematically guide through the iterative process. Moreover, it allows the traceability among both artefacts to avoid lacks of consistency.

Additionally, requirements and architectures can evolve iteratively and concurrently, in such a way that running systems can dynamically adapt their composition and/or topology to meet their evolving requirements. It is granted by the reflexive properties of PRISMA and by the relationships between requirements and architectures that have been established along the process.

In this work we do not deal with the development of effective environments providing automated support to the process, this will be presented in a later work. A companion CASE tool is being developed to automate the process activities, by using automatic and semiautomatic techniques and providing us with visual notations. It is defined as an extension of a commercial tool and it will be compliant with UML's for the Scenarios Models.

There are questions and problems that remain open: the completeness of each scenario and the completeness of the set of scenarios; the compatibility and consistency of the architectural styles; the creation of the architectural structure or its transformation on the basis of non functional requirements. All these subjects constitute our immediate future work.

References

[1] Aspect Oriented Software Development, http://aosd.net

[2] M. Brandozzi and D. E. Perry, "Architectural Prescriptions for Dependable Systems", Proceedings ICSE'2002, Orlando, May, 2002.

[3] J. Castro and J. Kramer (eds): Proceedings of First International Workshop From Software Requirements to Architectures (STRAW'01), 2001.

[4] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, Non Functional Requirements in Software Engineering, Kluwer Academic Publishing, 2000. 472pp.

[5] P. Crosby: Quality is free, New York, McGraw Hill, Chapter 24 (1979).

[6] A. Dardenne, A. van Lamsweerde, and S. Fickas: "Goal directed Requirements Acquisition". Science of Computer Programming, 20:3 50, 1993.

[7] D. Garlan: "Software Architecture: a Roadmap, in The Future of Software Engineering", A. Finkelstein, Ed.: ACM Press, pp. 93 101, 2000.

[8] P. Grünbacher, A. Egyed and N. Medvidovic: “Reconciling Software Requirements and Architectures: The CBSP Approach”. Proceedings 5th IEEE International Symposium on Requirements Engineering, pages 202 211. IEEE, 2001.
[9] J. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti, “Relating Software Requirements and Architectures using Problem Frames”, Proceedings of IEEE International Requirements Engineering Conference (RE'02), Essen, Germany, 9 13 September 2002.
[10] K.V. Hindriks, F.S. de Boer, W. van der Hoek and J. J.Ch. Meyer, “Agent programming in 3APL”. Autonomous Agents and Multi Agent Systems, 2(4):357 401, 1999.
[11] ISO/IEC DTR 15504 Software Process Assessment. International Organization for Standardization/International Electrotechnical Commission, 1998.
[12] J. Jaén and I. Ramos, “A Conceptual Model for Context Aware Dynamic Architectures”, 3rd International Workshop on Distributed Auto adaptive and Reconfigurable Systems, in conjunction with ICDCS2003 IEEE Conference, May 19 22, 2003 Providence, Rhode Island USA.
[13] P. Letelier, P. Sánchez , I. Ramos, O. Pastor: OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto. Universidad Politécnica de Valencia, SPUPV 98.4011, ISBN 84 7721 663 0, 1998.
[14] L. Liu, E. Yu: “From Requirements to Architectural Design Using Goals and Scenarios, From Software Requirements to Architectures”. Proc. 1st Int. Workshop From Software Requirements to Architectures (STRAW '01) in 23rd Int. Conference on Software Engineering, ICSE 2001.
[15] N.A.M. Maiden. “CREWS SAVRE: Scenarios for Acquiring and Validating Requirements”, Automated Software Engineering, vol. 5, pp. 419 446, 1998.
[16] E. Navarro, I. Ramos and J. Pérez: “Software Requirements for Architectured Systems”. Proc. of the 11t h IEEE International Requirements Engineering Conference (RE'03), September 8 12, 2003 in Monterey, California.
[17] B. Nuseibeh, “Weaving the Software Development Process Between Requirements and Architecture”, STRAW '01 in 23rd ICSE 2001.
[18] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability Maturity Model for Software, Version 1.1", Software Engineering Institute, CMU/SEI 93 TR 24, DTIC Number ADA263403, February 1993.
[19] J. Pérez, I. Ramos, J. Jaén and P. Letelier, E. Navarro: “PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures”, 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 7, 2003 (Accepted).
[20] D.E. Perry and A.L. Wolf: “Foundations for the study of software architecture”. ACM SIGSOFT Software Etzgitzeeritzg Notes, 17(4):40 52, October 1992.
[21] A. Rashid, P. Sawer, A. Moreira, J. Araújo. “Early Aspects: a Model for Aspect Oriented Requirements Engineering”. International Conference on Requirements Engineering (RE 2002), University of Essen, Germany, September 9 13, 2002.
[22] C. Rolland, G. Grosz and R. Kla: “Experience with Goal Scenario Coupling in Requirements Engineering”, IEEE International Symposium on Requirements Engineering, June 07 11, 1999, Limerick, Ireland.
[23] M. Shaw and P. Clements, “A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems”, Proc. COMPSAC'97, Washington, DC, Aug. 1997, pp. 6 13.
[25] The next major revision of the Unified Modeling Language™, http://www.u2 partners.org/
[26] D. Wile, “Residual Requirements and Architectural Residues”, Proceedings of the 5th International Symposium on Requirements Engineering (RE’01), Toronto, Canada, pp.194 201, IEEE CS Press, 2001.

El aspecto de distribución de PRISMA

Josep Silva, Nour Hussein, José Á. Carsí, Isidro Ramos

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46021 Valencia – España
{jsilva, nourali, pcarsi, iramos}@dsic.upv.es

Resumen. Las arquitecturas software permiten el desarrollo de software basado en componentes; uno de sus objetivos es la reusabilidad, que se consigue a través del desarrollo de artefactos software que puedan funcionar en distintos sistemas sin necesidad de ser reespecificados o reimplementados para ser usados en el nuevo contexto. Por otra parte, la combinación del desarrollo orientado a aspectos con el desarrollo orientado a componentes aborda la construcción de componentes software desde puntos de vista diferentes que realizan una funcionalidad específica y que pueden ser desarrollados de manera independiente. Este artículo presenta un modelo de desarrollo de software que combina ambas aproximaciones para la especificación de aspectos en la arquitectura PRISMA. Se propone una plantilla de especificación de aspectos que permite la definición de patrones de comportamiento de una manera totalmente independiente del componente. Esto permite la total reutilización no sólo de componentes sino también de aspectos en la arquitectura. El artículo presenta las plantillas de especificación y configuración de aspectos en la arquitectura PRISMA e incluye un ejemplo completo para el caso del aspecto de distribución.

Palabras Clave: Desarrollo orientado a aspectos, Desarrollo orientado a componentes, Arquitecturas software, Distribución.

1 Introducción

Muchos sistemas y procesos frecuentemente necesitan ser distribuidos o redistribuidos total o parcialmente por necesidades de la organización. Estos cambios en la distribución usualmente se realizan con la finalidad de mejorar la velocidad, la tolerancia a fallos o la interoperación entre componentes, para equilibrar la carga, o también, para satisfacer nuevos requisitos del sistema. Existen muchas propuestas para la especificación de sistemas distribuidos, algunas de ellas con la propiedad de ser independientes del sistema, como CORBA [1] y OSF DCE [2], mientras que otras específicamente diseñadas para un lenguaje o sistema operativo, como Java RMI [3] o Microsoft DCOM [4]. El principal inconveniente de estas propuestas es que la especificación de la distribución interfiere con la especificación de la funcionalidad. Desde nuestro punto de vista, la especificación de la funcionalidad de un sistema es

Silva, J., Ali, N., Carsí, J. A.,Ramos, I. El aspecto de distribución de PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003.

una tarea suficientemente compleja, y no debería ser mezclada con la especificación de la distribución de sus componentes. El analista/programador debería pensar en *qué* hacen los componentes y *cómo* lo hacen, sin preocuparse de *dónde* serán ejecutados esos componentes.

Existen propuestas [5,6,7] que incluyen lenguajes específicos para la especificación de los aspectos de distribución de un sistema. Estas propuestas usualmente tratan de separar la especificación de la funcionalidad de la especificación de la distribución. En muchas de estas propuestas, uno de los principales inconvenientes que emergen cuando se desarrolla un sistema distribuido es que los módulos distribuidos (o componentes) rompen la uniformidad del modelo no solamente por el hecho de estar especificados en un lenguaje distinto, sino porque a menudo interfieren en las tareas del resto de módulos del sistema de una manera poco transparente. Esto particularmente ocurre en el paradigma orientado a objetos, donde los objetos tienen que conocer la dirección de los servidores de nombres, y además deben de informarles si cambian de ubicación (esto significa además que tienen que estar asociados a un servidor de nombres). También deben registrar sus servicios, etc., con la pérdida de transparencia y reusabilidad que esto implica.

En la programación orientada a aspectos este problema ha sido parcialmente resuelto [8,9,10] gracias a la separación del aspecto de distribución del aspecto funcional. Sin embargo, ambos aspectos deben ser sincronizados para compartir atributos, eventos y otras propiedades. Esta sincronización limita en gran medida la reusabilidad del código. La arquitectura PRISMA [11] combina técnicas de reusabilidad de arquitecturas basadas en componentes con técnicas de reusabilidad de la programación orientada a aspectos. En este artículo, se propone un modelo de especificación del aspecto de distribución para la arquitectura PRISMA que permite la especificación de patrones de comportamiento de componentes distribuidos de una manera totalmente independiente del componente dónde ese comportamiento será utilizado. Este hecho permite una reusabilidad total de la especificación en otros componentes. Además, se introduce la particularidad de utilizar herencia entre aspectos en el modelo. El modelo será presentado con un ejemplo completo a lo largo del artículo. Se ha tomado como ejemplo de especificación el aspecto de distribución, sin que ello limite la aplicabilidad de las ideas aquí propuestas a cualquier otro aspecto de la arquitectura.

El artículo ha sido dividido en 5 secciones: El apartado 2 provee una breve revisión del estado del arte en el desarrollo de sistemas distribuidos, particularmente en la programación orientada a aspectos. En el apartado 3, se presenta una introducción a la arquitectura PRISMA y a su modelo de aspectos. En el apartado 4, presentamos las plantillas de definición y configuración del aspecto de distribución considerando las extensiones necesarias que deben ser realizadas en el lenguaje PRISMA. Finalmente, las conclusiones del artículo son presentadas en el apartado 5.

2 Estado del arte

La distribución y la intercomunicación de los componentes de un sistema, muchas veces es una cuestión que no afecta a la funcionalidad de los mismos. Aproximaciones como CORBA y Java RMI [1,3], hacen la distribución parcialmente transparente para el desarrollador proveyendo a las aplicaciones con una capa intermedia o 'middleware' (i.e. el Object Request Broker (ORB)) que oculta la distribución, pero los objetos deben ser registrados en el inicio a través de llamadas al ORB como parte de la funcionalidad de los mismos. Por otra parte, los programadores deben saber qué objetos estarán distribuidos y desarrollar interfaces para generar *stubs* e integrarlos en el ORB. En estas aproximaciones se consigue que el programador pueda definir sus componentes sin preocuparse de problemas propios de la distribución como el desarrollo de servidores de nombres. No obstante, no se aborda el problema de poder especificar el comportamiento de un objeto ante eventos ocurridos en el ORB (i.e. indicar que un componente debe clonarse al recibir cierta carga de mensajes en una unidad de tiempo). Problemas de este tipo deben ser resueltos por el programador.

El grupo CLIP propuso una extensión para el lenguaje Ciao Prolog para especificar la distribución de sus módulos [12]. En ese sistema, todos los módulos son especificados como si fueran locales; y la distribución es configurada con una nueva especificación utilizando el mismo lenguaje (con la extensión propuesta). Después de ello, el compilador sabe qué módulos deben ser distribuidos y estos son compilados de un manera particular. Es importante remarcar que el programador no mezcla funcionalidad con distribución, y solamente utiliza un lenguaje.

La programación orientada a aspectos tiene como principal objetivo la división del código fuente de las aplicaciones en diferentes módulos dedicados a resolver una parte concreta del problema tratado, la cual es llamada aspecto (i.e. distribución, evolución, coordinación, etc.). Esta división se realiza de tal forma que cada uno de los aspectos puede ser desarrollado de manera independiente al resto del código, lo cual beneficia la modularidad y reusabilidad del mismo. Muchas de las aproximaciones existentes hoy en día [9,10] son aproximaciones que tratan de aplicar esta disciplina a un lenguaje en particular como Java, C o Visual Basic. Otras aproximaciones [8,11,13], por el contrario abordan el problema desde un nivel de abstracción mayor en un lenguaje de análisis o de diseño y a partir del mismo realizan una compilación a otro de implementación como pueda ser Java. En este último caso se habla de análisis o diseño orientado a aspectos. La principal ventaja de estas aproximaciones radica en que la separación en aspectos se realiza en una etapa más temprana del ciclo de vida, pudiéndose obtener ventajas como la reutilización de aspectos en los diagramas de análisis.

En este artículo, se propone un modelo orientado a aspectos donde la especificación de la funcionalidad y la distribución se realizan separadamente pero en un mismo lenguaje. Nuestro modelo permite especificar los patrones de distribución de manera independiente de la funcionalidad de los componentes, pero además permite que esa especificación de la distribución pueda ser reutilizada en otros componentes. En particular, se divide el aspecto de distribución en dos secciones, una

sección totalmente reutilizable que especifica la distribución del componente, y otra parte dedicada a coordinar el aspecto de distribución con el resto de aspectos del componente y con el propio componente. De esta manera, se consigue poder reutilizar totalmente la especificación del aspecto de distribución pudiendo llegar incluso a utilizar mecanismos de reutilización entre aspectos como la herencia.

3 La arquitectura orientada a aspectos PRISMA

PRISMA [11] es un modelo arquitectónico para definir sistemas de información complejos a partir de la definición y configuración de artefactos software. Los artefactos PRISMA se definen a partir de la composición de distintos aspectos: funcional[1], calidad, distribución, coordinación, etc. Estos pueden variar en función del tipo de artefacto y sistema de información que se pretenda definir. Cualquier artefacto (componente, conector, sistema) del modelo arquitectónico de PRISMA puede ser visto como un prisma (véase figura 1). Cada cara del prisma define un aspecto del artefacto; cada aspecto tiene a su vez un nivel base y puede tener un nivel meta para dar cuenta de su propia evolución.

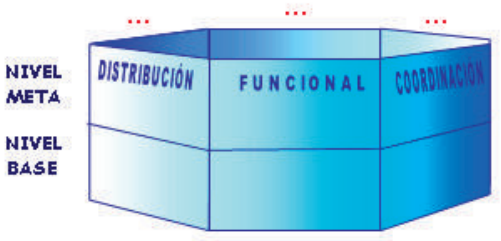


Figura 1: *Un artefacto PRISMA es un prisma compuesto a partir de sus aspectos.*

Un artefacto PRISMA puede ser de tres tipos:

Un Componente: es un tipo de artefacto que encapsula funcionalidad y que no actúa como coordinador entre otros artefactos. Consecuentemente, no puede tener el aspecto de coordinación, pero necesariamente debe contener un aspecto funcional.

Un Conector: es un tipo de artefacto que actúa como coordinador entre otros artefactos. Su función es enlazar y sincronizar la comunicación entre componentes, sistemas y otros conectores. Consecuentemente debe contener un aspecto de coordinación, pero no un aspecto funcional.

Un Sistema: es un artefacto que agrega un conjunto de conectores, componentes y otros sistemas conectados adecuadamente entre sí. Cada sistema tiene sus propios puertos y por lo tanto puede ser tratado como un componente. Adicionalmente, un sistema define enlaces entre sus propios puertos y los de sus componentes agregados.

[1] En PRISMA la funcionalidad de un artefacto es vista como un aspecto independiente.

Cada artefacto PRISMA se compone de un conjunto de aspectos que especifican su comportamiento global, y de puertos de entrada y salida para poder conectar el artefacto a otros artefactos y a través de ellos establecer la comunicación. Cada puerto implementa una interfaz específica que provee un conjunto de atributos y servicios. A través de los puertos de entrada, el artefacto recibe peticiones del entorno externo (comportamiento servidor). En contraste, a través de los puertos de salida, el artefacto realiza peticiones al entorno externo (comportamiento cliente). En ambos casos el componente no necesita saber con quién se comunica, sino que interactúa con la interfaz a través de sus puertos pudiendo estar conectado a varios componentes al mismo tiempo sin que este hecho pueda condicionar su comportamiento.

Entre los aspectos más importantes que un artefacto puede obtener en función de su tipo destacan:

El Aspecto Funcional: Captura la semántica del sistema de información definiendo su estructura y comportamiento.

El Aspecto de Coordinación: Establece las reglas de negocio y la sincronización durante la comunicación entre artefactos. Además, especifica la coreografía y los protocolos de comunicación, extendiendo el concepto de "*contrato*" [14].

El Aspecto de Distribución: Especifica las causas y consecuencias de la ubicación dinámica de las instancias de los artefactos.

En PRISMA existen dos lenguajes –ambos extensiones de OASIS 3.0 [15]– para especificar modelos arquitectónicos: El lenguaje de definición de componentes y el lenguaje de configuración de sistemas. El primero define los tipos de artefactos de la arquitectura; el segundo especifica como interconectarlos estableciendo una topología determinada. Ambos lenguajes establecen como constructores de primer orden interfaces, aspectos, componentes y conectores; de tal manera que cualquier interfaz puede ser reusada por diferentes aspectos, y cualquier aspecto puede ser reusado por diferentes artefactos.

Como resumen, los componentes son ignorantes de a quién están conectados en la arquitectura. Simplemente envían y reciben peticiones mediante sus puertos sin conocer el origen o destino de las mismas. La conexión de distintos componentes se realiza a través de un conector que sí es consciente de los componentes que coordina. Teniendo en cuenta esta característica, en el aspecto de distribución, serán los conectores los únicos artefactos que conocen la ubicación de los artefactos a los que se conectan, permaneciendo esta información oculta para los componentes.

4 El aspecto de distribución de PRISMA

En este apartado se va a presentar el aspecto de distribución de la arquitectura PRISMA. En su definición se han preservado las características del modelo PRISMA permitiendo que la especificación del aspecto sea independiente de la arquitectura del sistema concreto donde ese aspecto funcionará (i.e. en la definición del aspecto no es preciso conocer con qué otros artefactos será interconectado). Además, la definición

del aspecto es independiente del propio artefacto donde este será insertado (i.e. en la definición del aspecto no es preciso conocer la definición del propio artefacto ni del resto de aspectos que este incluirá). De esta manera se consigue maximizar la reutilización de las especificaciones realizadas en otros componentes.

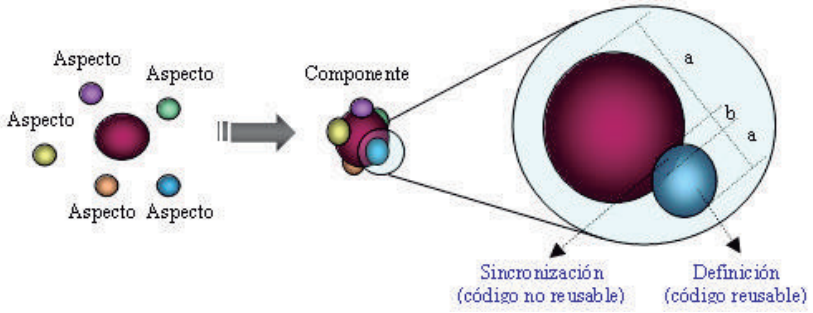


Figura 2: *Reutilización de aspectos a partir de su sincronización en componentes*

La especificación del aspecto de distribución de PRISMA está dividida en dos partes: la primera parte especifica *qué comportamiento* tiene este aspecto en el sistema distribuido (i.e. qué patrones de distribución satisface este aspecto); la segunda parte especifica *cómo sincronizar* el comportamiento del aspecto de distribución con el comportamiento del resto de aspectos. Esta división permite que la primera parte del aspecto sea totalmente independiente del componente en el que ese aspecto va a ser incluido, lo cual permite la reutilización de aspectos.

En la figura 2 se muestra como un conjunto de aspectos ya definidos pueden ser reutilizados incluyéndolos en un componente de tal manera que para acoplar el aspecto en el componente solamente es necesario especificar la sincronización entre el aspecto y el componente (*b*). El resto de la especificación es independiente del componente y consecuentemente reutilizable. En consecuencia, una especificación completa consta de dos partes, la parte *definición* la cual es independiente del componente y la parte *sincronización* la cual está totalmente ligada al componente. Las letras *a* y *b* también indican el orden de desarrollo de las distintas partes del componente. Concretamente, no es posible desarrollar la parte de sincronización de un aspecto sin tener previamente desarrolladas las especificaciones del componente base y del propio aspecto. Es evidente que este mecanismo puede ser incorporado en todos los aspectos de la arquitectura, sin embargo este trabajo está orientado a la especificación del aspecto de distribución.

Como ya se ha comentado en el apartado 3, existen modelos [13] que permiten componer un componente complejo a partir de aspectos parcialmente independientes; ahora bien, si el arquitecto deseara reutilizar un aspecto de distribución concreto para definir otro aspecto más complejo no podría hacerlo. Esto se debe a que en la mayoría de las aproximaciones no existe la herencia entre aspectos, puesto que el nivel de granularidad en la reutilización es el aspecto completo. En nuestra propuesta estas limitaciones se ven solventadas gracias a la separación introducida entre especificación y sincronización que hace que la primera parte de la especificación sea totalmente independiente del componente.

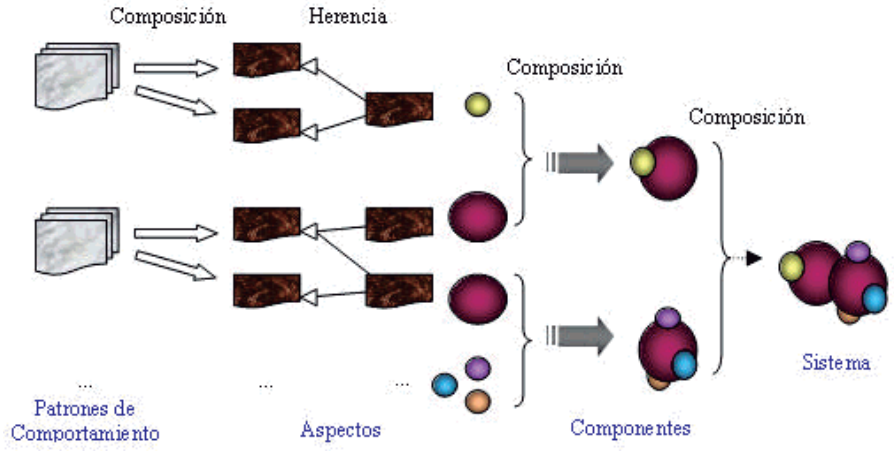


Figura 3: *Proceso de desarrollo de sistemas en PRISMA*

El mecanismo de herencia obliga a incluir en la plantilla de definición del aspecto de distribución un mecanismo de identificación propio y del resto de aspectos para poder determinar de cual de ellos se heredan propiedades. Sin embargo, la herencia también permite poder realizar una composición de aspectos incremental a partir de patrones de comportamiento ya definidos, permitiendo así que la reutilización no solo se realice en el momento de crear componentes, sino también en el momento de componer aspectos. De esta manera, el arquitecto dispondría, para el caso del aspecto de distribución, de una colección de patrones para componer aspectos de distribución y de una colección de aspectos para componer componentes (véase figura 3).

Cada patrón de comportamiento consiste en una especificación de aspecto que captura la semántica de un patrón elemental que se repite con frecuencia. Gracias al mecanismo de herencia la composición de aspectos puede realizarse a partir de librerías de patrones ya especificados reutilizándolos en cada especificación. La figura 4 presenta las plantillas de definición y configuración de aspectos para el caso del aspecto de distribución. Posteriormente, se presenta un ejemplo completo de especificación usando ambas plantillas en el que se aprecia el mecanismo de herencia.

Las plantillas de la figura 4 dividen la especificación del aspecto de distribución en dos secciones. La primera sección define el comportamiento del aspecto a partir de la semántica del lenguaje OASIS 3.0 [15]; a partir de ella, dicha definición establece el comportamiento cliente-servidor autodefiniendo todos los atributos y servicios sobre los que se quiera establecer una observación. La segunda plantilla asocia estos servicios y atributos con los correspondientes de los aspectos del componente en el que el aspecto de distribución se inserte. De esta manera, la primera especificación define el comportamiento independientemente del componente, y la segunda parte lo configura y sincroniza con el resto de aspectos. Adicionalmente, permite especificar una URL donde el aspecto comenzará su ejecución.

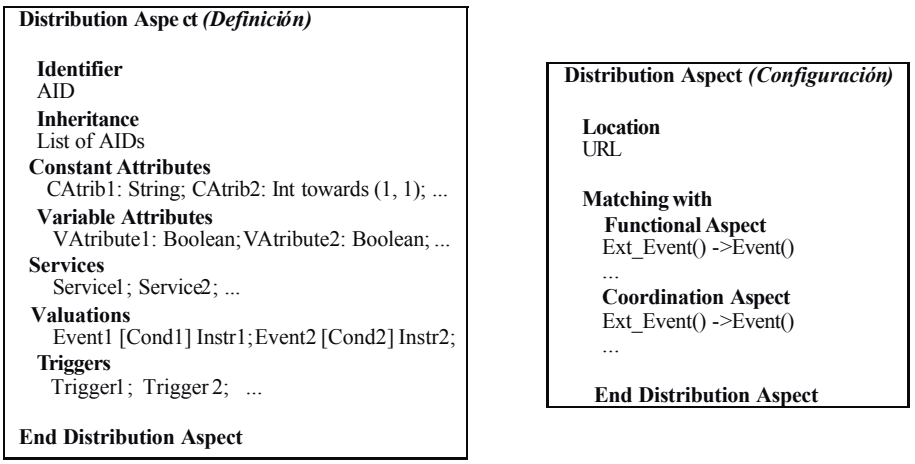


Figura 4: *Plantillas de definición y configuración del aspecto de distribución*

La figura 5 muestra la especificación de dos patrones de comportamiento AID1 y AID2 del aspecto de distribución. El patrón AID1 ofrece el servicio '*move*' al resto de componentes del sistema, permitiendo así que otros componentes puedan mover[2] al componente que posee este patrón en su plantilla. El patrón AID2 ofrece el servicio '*clone*' que crea una copia del componente que posee este patrón en la URL especificada. Adicionalmente mantiene una lista de las copias creadas. Reutilizando los patrones AID1 y AID2 a partir de una relación de herencia se ha definido el aspecto de distribución AID3, el cual hereda el comportamiento especificado en AID1 y AID2 y además extiende dicho comportamiento especificando que cuando el servicio '*Ext_event*' sea invocado 100 veces entonces el aspecto de distribución supondrá que existe un elevado tráfico en la red producido por invocaciones a ese servicio y consecuentemente creará una copia de ese componente a través del servicio '*clone*'; de esta manera la carga puede repartirse entre ambos componentes. Como puede apreciarse, la definición de los aspectos que se ha realizado es totalmente independiente del componente en el que ese aspecto vaya a ser insertado. Cada definición de aspecto únicamente especifica un conjunto de servicios que el aspecto ofrece, y la manera en que el aspecto debe comportarse ante una llamada a los mismos. La interacción entre un aspecto particular y el resto de aspectos de un componente se realiza en la plantilla de configuración del aspecto. Para el caso del ejemplo, la plantilla de configuración que se ha especificado puede apreciarse en la figura 5. Concretamente, en dicha plantilla se especifica, además de la dirección inicial de ejecución del componente, que el servicio '*Ext_event*' sobre el que se desea realizar una observación se corresponde con el servicio '*Copyfile*' del aspecto funcional. De esta manera, el acoplamiento entre la especificación del comportamiento y el resto de aspectos del componente se realiza en una especificación independiente permitiendo su total reutilización.

[2] La instrucción System.move(URL) es una llamada al middleware PRISMA que básicamente crea una instancia en la URL especificada igual al componente invocador a través del servicio '*create*', y a continuación invoca al servicio '*destroy*' de ese mismo componente.

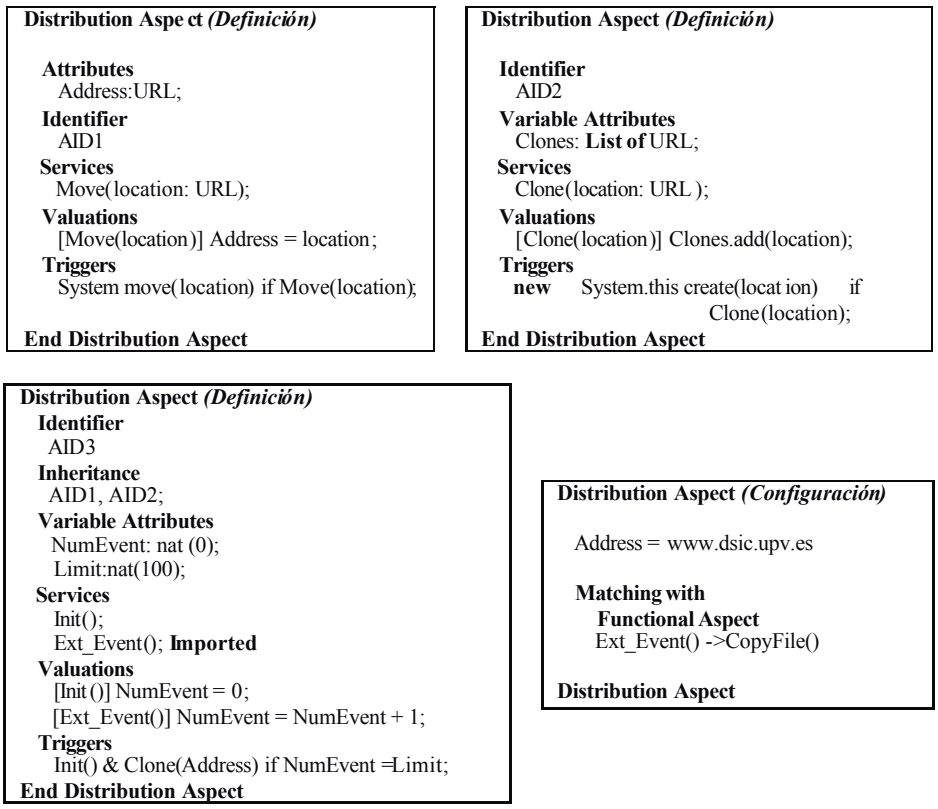


Figura 5: *Ejemplo de definición y configuración del aspecto de distribución*

5. Conclusiones

La especificación de la distribución constituye una importante tarea en el desarrollo de sistemas de información. En la actualidad, existen trabajos en los cuales dicha tarea se realiza de una manera independiente de la especificación de la funcionalidad. En particular, esto ocurre en el paradigma orientado a aspectos donde la distribución y la funcionalidad son especificadas en aspectos separados.

En este trabajo se presenta una propuesta que combina arquitecturas software con desarrollo orientado a aspectos, elevando de esta forma el nivel de abstracción en la especificación de la distribución. Se han introducido las plantillas de especificación y configuración del aspecto de distribución de la arquitectura PRISMA. En la plantilla de especificación se indican qué patrones de comportamiento de distribución satisface el aspecto. Dicha especificación es independiente del componente y del resto de aspectos del mismo; permitiendo ser totalmente reutilizada en otros componentes. La plantilla de configuración es la encargada de sincronizar la especificación con el resto

de aspectos. La separación introducida entre definición y configuración ha permitido utilizar mecanismos como la herencia en la plantilla de especificación de aspectos.

Aunque este trabajo ha sido realizado para la plantilla de especificación sobre un aspecto concreto (distribución) de una arquitectura concreta (PRISMA), las ideas expuestas sobre división de las especificaciones para su posterior reutilización pueden ser aplicadas a cualquier aspecto y arquitectura.

Referencias

1. CORBA Official Web Site of the OMG Group: http://www.corba.org/
2. DCE documentation of Open Software Foundation (OSF): http://www.transarc.ibm.com/Library/documentation/dce/1.1/index html
3. JAVA Remote Method Invocation (RMI) Specification of SUN Microsistems: http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC html
4. Distributed Component Object Model (DCOM) at Microsoft Web site: http://www microsoft.com/com/tech/dcom.asp
5. Kermarrec, Y., Nana, L., Pautet, L., and Tardieu, S.; "*GNATDIST : a configuration language for distributed Ada 95 applications*" In ACM, editor, ACM Tri Ada conference, Philadelphia, Pages 63-72, 1996.
6. Haridi, Seif, Van Roy, Peter; Smolka, Gert. "*An Overview of the Design of Distributed Oz*". Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97), p.176-187, Maui, Hawaii, USA. July 1997.
7. Álvaro López Ortega; "*Programación distribuida con ADA95 bajo GNU/Linux (I)*"; Editorial Hispalinux, Valencia, España. 2001.
8. Herrero J. L., "*Propuesta de una Plataforma, Lenguaje y Diseño, para el Desarrollo de Aplicaciones Orientadas a Aspectos*". Tesis Doctoral, Extremadura, Mayo 2003.
9. Samtani Sergio Soares and Paulo Borba. "*PaDA: A Pattern for Distribution Aspects*". In Second Latin American Conference on Pattern Languages Programming — SugarLoafPLoP, Itaipava, Rio de Janeiro, Brazil, August 2002.
10. Sérgio Soares, Eduardo Laureano and Paulo Borba. "*Implementing Distribution and Persistence Aspects with AspectJ*". Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02, páginas 174-190, ACM Press. 2002, Seattle, WA, EUA. ACM SIGPLAN Notices 37(11).
11. Pérez J., Ramos I.,Lorenzo A., Letelier P., Jaén J., "*PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos*", Actas VII Jornadas de Ingeniería del Software y Bases de Datos, JISBD, Escorial (Madrid), Noviembre 2002.
12. J. Correas, F. Bueno. "*A Configuration Framework to Develop and Deploy Distributed Logic Applications*". ICLP01 Colloquium on Implementation of Constraint and Logic Programming Systems, November 2001.
13. M. Pinto, L. Fuentes, M. E. Fayad, J. M. Troya, *Separation of coordination in a dynamic aspect oriented framework*, Proceedings of the 1st international conference on Aspect-oriented software development, Enschede, The Netherlands, Pages: 134 – 140. 2002.
14. L.Andrade and J.Fiadeiro; "*Interconnecting Objects via Contracts*", in UML'99 -Beyond the Standard, R.France and B.Rumpe (eds), LNCS 1723, pp. 566-583, Springer Verlag 1999.
15. P.Letelier, P.Sánchez, I.Ramos, O.Pastor. "*OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto*", Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721-663-0, 1998.

MOBILITY AND REPLICABILITY PATTERNS IN ASPECT-ORIENTED COMPONENT-BASED SOFTWARE ARCHITECTURES

Nour Hussein Ali, Josep Silva, Javier Jaen, Isidro Ramos, Jose A. Carsí, Jennifer Peréz
Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n
E-46071 Valencia – Spain
{nourali | jsilva | fjaen | iramos | pcarsi | jeperez} @dsic.upv.es

Abstract

The development process of software systems has become extremely complicated due to the complexity of the structures and the continuous evolution of the requirements. Software architectures are backbones for developing complex, dynamic and distributed software systems. Recently, two approaches have emerged to construct architectural models: the Component-Based and the Aspect-Oriented Software Development. In this work, we present distribution patterns for software architectures. These patterns describe cases in which elements should react to issues and adapt to changes in requirements to optimize the architecture by relocating or replicating. These patterns are applied to an aspect-oriented component-based software architecture called PRISMA to specify its distribution aspect using its ADL (Architecture Descriptive Language).

Key Words

Reuse, Patterns, Migration, Replication, Aspects, Components.

1. Introduction

Nowadays most systems are complex, distributed, dynamic and have to continuously adapt to new organizational needs. The development of these distributed systems can be done via the specification of their architectural models.

Dealing with systems at early stages of development is worthy due to the costs and time reduction that result at later stages. Despite of the improvements provided by the object-oriented approach at early stages of software systems' development, it has been proven that it does not satisfy all the requirements needed nowadays to develop complex systems. The poor capability of the model for reutilization, evolution and adaptation to changes implied the rise of new approaches as the Component-Based Software Development (CBSD) [1] and the Aspect-Oriented Software Development (AOSD) [2]. In CBSD, the parts of the system can be reused at high levels of abstraction, reducing the time needed to develop a system. The AOSD techniques allow separation of concerns by modularizing crosscutting concerns in a separate entity: the aspect. The encapsulation of the aspect allows a high abstraction level, a higher reutilization and the possibility of manipulating the aspects at execution time.

In our work we deal with the distribution of software architectures at a high abstraction level enabling the validation of the architectures at early stages of development. The elements are migrated and replicated for improving speed, interoperability and fault tolerance, satisfying load balance and adapting to changes without interfering with the functional specification. These interactions can be done at compilation, in a static way, or dynamically at execution time.

Taking into account the benefits of the previously described approaches, our proposal combines the AOSD and CBSD approaches to define distributed software architectures. The CBSD approach allows us to reuse architectural elements independently of their location. Moreover, the AOSD allows the independence of the distribution behaviour of these elements from their functional behaviour. This separation is achieved by encapsulating all the distribution concerns and properties in an independent entity –the distribution aspect– allowing its manipulation, maintenance and evolution without affecting the non-distribution entities of the elements. In addition, the system can use different middlewares at the same time e.g. two clients accessing the system one using JAVA RMI [3] and the other CORBA [4], dynamically change the middleware and facilitate functional tests locally. In addition, these distributed properties and behaviours can be reutilized by different elements due to the separation and modularization.

In this paper, we introduce distribution patterns that can be used as guidelines to develop distributed software architectures with an aspect oriented approach. The patterns describe causes for the mobility and the replicability of elements when faced with distribution issues such as saturation, unbalance load, latency, etc and

Ali, N., Silva, J., Jaén, J., Ramos, I., Carsí, J. A., Pérez, J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proc. of the 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2003). 2003.

changes in properties of the system. We apply the patterns for the mobility and replication of elements in an aspect-oriented component-based software architectural model called PRISMA [5]. These elements are described via aspects. The patterns are used for the specification of the properties and behaviours of a distribution aspect. PRISMA describes the architecture using two languages: the component definition language and the configuration language. These languages are extensions of a formal object-oriented specification language called OASIS 3.0 [6].

The sections of the paper are structured as follows: section 2 presents some related works on distribution. Section 3 presents the current state of the PRISMA architectural model. In section 4 the mobility and replication patterns are presented introducing how to use them for the PRISMA architectural model and language. Finally, section 5 draws some conclusions and highlights some future work.

2. Related Works

Sometimes the distribution and intercommunication of the components of a system do not affect their functionality. Approaches as CORBA [4] and JAVA RMI [3] make distribution partly transparent for the developer. CORBA provides the applications with an intermediate layer or "middleware" (e.g. the Object Request Broker (ORB)) that hides distribution, whereas the objects have to be registered initially by calls to the ORB. On the other hand, programmers have to know which objects will be distributed and develop interfaces to generate stubs and include them in the ORB. These approaches enable the programmer to define its components without taking into account distribution details as name servers. However, the problem of enabling the specification of an object's behaviour against events occurred in the ORB (e.g. indicating that a component has to be cloned when it receives a certain load of messages in a unity of time) is not supported. This type of problem has to be resolved by the programmer.

Other proposals try to make the development of the functionality totally independent from the configuration of the distribution. One of these proposals is GNATDIST [7]. This tool allows communication and distribution of applications' partitions developed with GNAT-Glade, which is a compiler of the Ada95 language. In this model two similar languages exist: one for the specification of the functionality and another one for the specification of the distribution.

The CLIP group of the Madrid University of Technology [8] proposed an extension to the Ciao Prolog language for specifying the distribution of its modules. In this system, all modules are specified as if they were local and the distribution is configured with a new specification in the same language (with the extension proposed). Next, the compiler knows which modules should be distributed and these are compiled in a particular manner. It is important to take into account that the programmer separates the functionality from the distribution and only uses a single language.

Work has been done in the conceptual modelling of mobile object systems using language constructs which can clearly distinguish between the mobile and stationary parts of a complex system. There is an extension of the TROLL language [9] in order to specify these mobile object systems. However, the work does not incorporate the flexibility of evolving the stationary parts into mobile parts. Many software architectures at conceptual levels have been developed using an ADL (Architecture Description Language) such as LEDA [10] and Darwin [11, 12]. However, these ADLs are still not mature enough to provide mobility, replication and fault tolerance specifications.

The rise of aspect-oriented programming languages introduced the encapsulation of the distribution code in the distribution aspect separating it from the rest of the code. This is done by extending known programming languages as AspectJ [13, 14] in the case of JAVA. Whilst, an inconvenient of these low level languages is the definition of the pointcuts inside the aspects decreasing reusability. Therefore, to solve these issues the aspect-oriented software development approach emerged. It incorporates aspect concepts at early stages of development, as in the design phase, and supports code generation to AspectJ. In works as in [15] distribution issues as load balancing, caching, migration are not contemplated. Moreover, it does not use a general purpose language (i.e. the language used to specify the distribution behaviour is different from the one used to specify the functional behaviour of the architecture).In fact, when describing the functional entity (the object) Java is used, whereas for defining the distribution aspect another language is used and for composing the aspect with the object a third language is used.

In our conceptual aspect-oriented component-based architectural model PRISMA [5] the manipulation of distribution issues and properties are separated from the rest of the properties of the system. The ADL used is a general purpose language allowing the manipulation of the distribution behaviour, even at execution time, and the total reuse of distribution aspects for different elements with the same distribution behaviours.

Patterns that provide a structure for implementing distribution using AOP (Aspect Oriented Programming) applied to AspectJ have been presented in [13] to achieve better separation of concerns. The objective of our work is to specify patterns for aspect-oriented component based architectural elements that can change their location and replicate to face issues such as caching, fault tolerance,

unbalanced load, latency and adaptation to non-distribution properties. These can be used as guidelines when developing configurable distributed software architectures. These patterns will be used to describe the distribution aspect behaviour in PRISMA.

3 The PRISMA Architectural Model.

PRISMA [5] is a model to define architectures of complex software systems. The PRISMA elements are defined by composing different aspects: functional, quality [16], distribution, coordination, context-awareness [17], etc. These may vary depending on the type of architectural elements and the type of information systems taken into consideration. Any element of the PRISMA architectural model can be seen as a prism (see figure 1). Each side of the prism is an aspect which is divided into a base level and can have a metalevel to achieve evolution of this specific aspect.

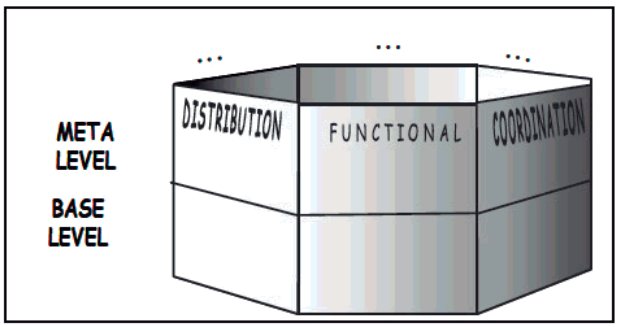


Fig. 1. A PRISMA element as a prism.

The PRISMA elements types are:

Component: A component is an element of the architectural model that captures system functionality and that does not act as a coordinator between other elements. Therefore, it cannot have the coordination aspect but must have the functional aspect.

Connector: A connector is an element which acts as a coordinator between elements. It links and synchronises components, systems and other connectors by means of their roles. Therefore, it must be composed of a coordination aspect and cannot have a functional aspect.

System: A system is an element that includes a set of connectors and components and other systems connected with each other. A system has its own ports (inports and outports) and can be treated as a component. Moreover, a system defines binding links between its ports and the ports of the included components. The system is defined by the configuration language.

Each PRISMA element is composed from a set of aspects, in-ports and outports (in the case of connectors they are called inroles and outroles, respectively). Each inport / inrol or outport /inrol has a specific interface that makes public a set of services. An element receives petitions from the external environment by means of the inports / inroles (server behaviour). In addition, the element requests petitions from the external environment by means of the outports / outroles (client behaviour).

The most common aspects an element can contain are the following:

Functional Aspect: The functional aspect captures the semantics of the information system defining its structure and behaviour.

Coordination Aspect: The Coordination aspect defines the business rules and the synchronisation between elements during their communication. Moreover, the coordination aspect includes the choreography extending the "contract" concept [18].

Distribution Aspect: The distribution aspect specifies the features that define dynamic location of the instances.

In PRISMA there are two languages, which are extensions to OASIS 3.0 [6], to specify architectural models: the component definition language and the configuration language. The component definition language defines the elements of the architecture. This language has as first-class constructs: interfaces, aspects, components and connectors in such a way that any interface may be used by several aspects and aspects may be used by several elements including the distribution aspect. This allows designers to reuse not only components but also interfaces and aspects.

```
/* interfaces definition*/
Interface I1
  Attributes…
    Services …
End Interface I1

Interface I2
  Attributes…
  Services…
End Interface I2
…

/*aspects definition*/

Distribution Aspect D specifies I1, I2
  Attributes …
  Services…
  Valuations…
  Triggers…
  Protocol …
End Distribution Aspect D;
…
/*components  definition*/

Component_type C
  Inports Inport1:I1; End Inports;
  Outports Oport1:I2; End Outports;
  Functional Aspect F;
  Distributuion Aspect D;
  Weaving
     Aspect1 with Aspect2
     Op-Service() ->Service()
     …
End Component_type C;
```

Fig. 2. The template of the component definition language.

The component definition language (see fig. 2) specifies the distributed interfaces. Each **interface** has a

set of attributes (consulting services that do not modify the element state) and services. These interfaces can be used by any distribution aspect that needs the interfaces' services. Moreover, the distribution aspect specifies its **attributes** and the semantics of its internal **services** and the services of its interfaces. The **valuations** specify how invocations to services affect attributes. The **triggers**, as the name indicates, specify the triggers that occur to the distribution aspect. The component type is described specifying the **inports** and **outports** typed with interfaces, the aspects it is composed of and the pointcuts (**weaving**) between its composed aspects. The Op defines the sequence of execution of the two aspects whether it is before, after or around.

As previously noticed, the component definition language defines the architectural elements at a high abstraction level which can be easily reused without specifying any location of the elements. Whereas, the configuration language imports the types of elements, defined previously with the component definition language, and defines instances and the topology of a specific architectural model. The types of elements are imported, instanced and finally the topology of the model is expressed by interconnecting the instances. Such interconnection is achieved by creating concrete instances of the connector's synchronizations, attachments between connectors and other elements, and system's bindings to obtain a system. At this level of abstraction, the distribution aspect defines the location of the elements in the architectural model. The distribution aspect looks as follows:

Distribution Aspect ***(Configuration)***
Attributes
Location:URL(*value*);
End Distribution Aspect

As a summary, the components do not know whom they are connected to in the architecture. They only send and receive petitions without realizing to whom or from whom these petitions are. These components are called context independent [12]. The elements bind with each other by means of connectors. Whereas, the connector knows the locations of the elements it links.

4. Mobility and Replicability Patterns for PRISMA Elements.

In this section we present patterns that describe a specific case in which a distribution behaviour, such as relocation or replication, should be performed in order to react at issues such as unbalanced load, saturation, latency or for adaptation for non-distributional changes. These patterns can be served as guidelines for any software architecture with mobile and replicable elements that separate the distribution properties from the functionality of the system. The patterns we present are for conceptual architectural models.

The patterns specified will be used to show their appliance on the PRISMA architectural model. These patterns are intended to be used for the specification of the services and attributes a distribution aspect can have. Although, these patterns are used for the PRISMA architectural model they also can be used for other architectures.

The use of patterns saves time and costs in the development of systems. The developer of the system at the time of specifying the architecture only needs to choose the most adequate patterns for fulfilling the systems requirements. These patterns can easily compose aspects in an aspect-oriented architecture as shown in Figure 3. Each pattern defines the case in which an element can be replicated or moved. The architect can combine these patterns into different distributional aspects. In figure 3, the small balls number one and three represent different distribution aspects composed from a library of distribution patterns, and the small ball number two represents a non-distributional aspect. The combination of the different patterns depends on the needed distribution behaviour of the architecture. Designers have to combine them to form a complex distribution aspect, and then, that distribution aspect is weaved with other aspects to create an element. Afterwards, the elements are connected to form systems. In the example of the figure, notice that the two elements are created from different distributional aspects but have the same non-distributional aspect.

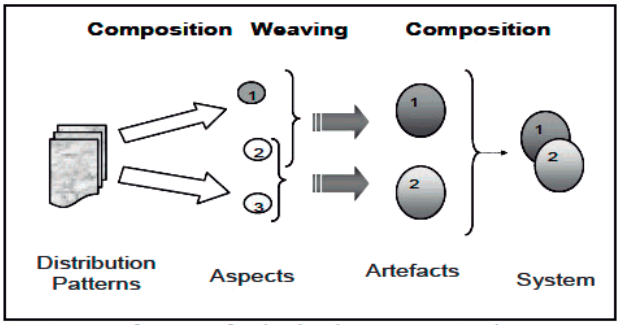


Fig. 3. Use of patterns for the development process in PRISMA.

4.1. An Example

The patterns are going to be explained using a concrete example based on a banking system: A bank chain has a number of bank offices in different locations. Each customer opens an account (the account can be shared by different people) at a specific bank office. This bank office will attend all the services of the customer. A customer can deposit and withdraw money from any ATM. Each bank system has a database for the customer records and the database is updated by a programme that administrates the database.

4.2. Distribution Patterns

In this section we introduce the proposed patterns with some explanations of each of them and a representative example based on the previously described bank system. These patterns belong to a library, where they can be identified by a pattern number (**P-number**).

P.01. Service invocation
When a service is invoked a change of state in the element occurs, the service can belong to the distribution aspect or to any other aspect of the element. This pattern controls these state changes and produces a distributed behaviour related to them, as for example a change of location or replication of the own element. By means of these distributed behaviours this pattern can also be used in order to improve the efficiency in communications. For example, the architect can specify that the invocation of a concrete functional service will cause a drastic increment in the requests to an element. As a result, the element can replicate itself in order to be able to resolve the requests efficiently.

This pattern can be applied to the example of the bank system in the case a customer changes its address. Therefore, the principle bank office that attends the customers' requests and services will have to change to the bank office nearer to his/her new address. In this situation the record of the customer and account will move to the new bank office or in other cases they can be replicated.

P.02. Service invocation in a certain number of times
This pattern is essential to avoid saturation and errors that can produce breakdowns due to the high petition of a concrete service. Therefore, to optimize the network the server can migrate or replicate to a neighbourhood closer to the clients. Take notice that this pattern can include the situation in which the same service is invoked a certain number of times within a timeinterval or just during the whole lifecycle of a specified element.

This pattern can be applied to our bank system example as follows: let us say that at night (a specific hour of the night) all transactions of ATM machines should backup to the database. Then, at this time all ATM machines should invoke the service backup of the database administrator programme. The load of the petitions will be enormous; as well the same programme will not cope to serve all the ATM machines. To optimize the system, the programme can be replicated minimizing saturation and balancing load.

P.03. Excess in the invocation of services of an element
When an element receives a determined quantity of calls to its services, a saturation of serving events can occur. In order to avoid this situation, the architect must specify the limit of calls that this element should be able to accept and, when that limit is exceeded, the actions that the element should perform to solve the situation. The most probable is that the element needs to make a replicate of itself in order to balance the load. This pattern can also include the case that the total number of services invoked reaches a certain quantity in a unit of time.

In addition, we present some special cases of this pattern for PRISMA or any architecture that its elements' ports have a set of services instead of one:

I. **Total number of messages that interact with a port exceeds a certain quantity.**

Ports are the doors to an element. They are the means elements interact with the external environment. As an element can be saturated, a port can be as well. Consequently, migration can be performed to the location of the connected connector or to the neighbourhood of the clients that request petitions to a determined port. In addition, the total number of events that interact with a port in a unit of time can be taken into consideration.

II. **Percentage of messages of a specified port exceeds a certain percentage with respect to the total number of events of an element**

When this pattern is satisfied, it informs that the load of a port is heavier than the other ports of the element. In this way the element is moved to the location of the connector that extremely interacts with this element. In addition, it can move to the neighbourhood of the clients of this element reducing the communication traffic in the network. As a special case of the pattern time can be considered.

The example of the bank system can apply this pattern in the case that in a unit of time, usually in the morning, when the services of the bank's database are heavily accessed. Therefore, to avoid saturation on the database, the administrator programme is replicated to the locations of the bank offices.

P.04. Excess of the percentage of messages received to an element by a source with respect to the total number of sources
In the cases in which two elements communicate very frequently with each other with respect to the other elements of the architecture, the communication channel between these two elements should be considerably optimized with respect of the other channels. The distribution aspect has to evaluate if the communication with that element is much greater, and in the case it is, an element has to change its location in order to favour that communication. The mobility of the element closer to the source can minimize the response time and optimize the communication between the client and the server. As a result, the errors that may occur due to the distance of the communication path can be reduced.

The example of the bank system can apply this pattern in the case that an account is shared by two clients that consult the account from different offices or ATMs. If one

of the clients sharing the account accesses much more than the other, the element 'account' should be able to optimize the communication by moving itself to the office of the client which accesses it more. Moreover, it should be able to detect the situation in the case that the other client starts to access the account.

P.05. Excess of the volume of data interchanged with a source

The volume of data interchanged in a network can generate saturation in the network, especially, in cases where the target of the data can not process such amount of information. Many times the bandwidth of the communication channel can't be increased and it is necessary to cancel the communication in order to avoid saturation. This kind of situation can be solved if the amount of data passing through an element's ports is controlled by moving an element to the host of the other and perform the communication locally.

In our example, there are elements that calculate statistics from the daily operations of each ATM every night. In this situation, it would be a bad idea to send all the operations from all the ATMs via the net at the same time interval, because it could probably cause a collapse. The element could consider the amount of data to be sent and move itself to the ATMs in order to get all the data locally, without affecting the current time response of the whole system or replicate itself.

P.06. Excess of the latency time of a message

The latency of a requested service exceeds a specific time means that problems are occurring in the network. If a certain element detects that a latency of its message has exceeded, then the element has to try to accommodate his situation as easily as possible so as to continue his work. To accomplish this it can replicate to a neighbourhood nearer to the elements it interacts with.
In the bank system if an ATM requests a service from the programme and no answer is received, the ATM will ask the programme to replicate.

It can be noticed that these patterns just introduced are simple but should essentially be considered in developing distributed software architectures. These patterns can be joined to form complex distributed software architectures by combing them in different ways. The purpose of identifying these patterns is for defining the distribution aspect of our architectural distributed model PRISMA. In this way, the services for the mobility and replicability of the PRISMA elements will be defined saving time for the architect to start defining them from zero.

These patterns can be easily expressed in the PRISMA languages. For example let's take the examples of the first and fourth pattern, the situation in which a customer changes its address therefore the customer record needs to be ubicated to a new location of his/her bank office and the case the customer record replicates to the location of the ATM. The skeleton of the component definition language of the client record will look as follows:

```
Interface Mblty
 Services
   move;
End Interface
Interface Rplctn
 Services
   clone;
End Interface
Interface chngdrs
 Services
  Change_address;
End Interface

Distribution Aspect D specifies Mblty,Rplctn
 Attributes
  Location:URL;
  Num_Servs_ATM:nat(0);
  ATM_ID:URI;
  Limit:nat;
 Services
 move(Bank_off_loc:URL);
 clone(ATM_loc:URL);
 init();
 Valuations
  [move(Bank_off_loc)]Location:= Bank_off_loc;
  [allPorts.Events() &ATM_ID]Num_Servs_ATM=
Num_Servs_ATM + 1;
  [init()]Num_Servs_ATM =0;
 Triggers
 Clone(ATM_loc)&init()when Num_Servs_ATM = Limit;
 move(Bank_off_loc)when ExtEvent();
  new System.this.create(ATM_loc)when clone(ATM_loc);
End Distribution Aspect

Component_type C
 Inports  Inport1:Mblty, Inport2:Rplctn, Inport3:chngdrs
End Inports;
Outports Oport1: chngdrs; End Outports;
Functional Aspect F;
 Distribution Aspect D;
Weave
     Functional Aspect with Distribution Aspect
     After-Change_address ():Ext_Event()
     ...
End Component_type C;
```

Fig. 4: The specification in PRISMA of the customer record that moves and replicates.

In figure 4, three interfaces are defined. Two are distribution interfaces and a functional interface. The distribution interfaces define the distribution services that can be invoked by external elements. The functional interface defines the functional service, *change_address*, to specify the act of changing the address of a client. The distribution aspect specifies the distribution interfaces describing the distributional behaviour of the services. As it can be noticed in the services section there is a service *init* that is not part of any distribution interface, this is due to that this service is an internal service that does not serve the external environment. This service only initializes the *NumEvent* attribute which counts the number of services invoked in the component by a specific ATM, when the element is cloned.

In addition, the *change_address* service affects the location of the element after it is invoked. This is specified in the weaving part of the component by the **After** defined term. Moreover, it can be noticed that the *change_address* service is not assigned in the distribution aspect. The aspect specifies that the execution of an external event will cause a change of location of the component. In this way the distribution aspect is reused and easily adapted to new changes.

5. Conclusions and Future Work

The component-based approach allows the separation of a system into smaller parts (elements) and the aspect-oriented approach which aims at separating the concerns in different entities, which were tangled in the object oriented approach, have shown improvements in dealing with distribution issues and behaviours. The PRISMA conceptual distributed aspect-oriented component-based software architectural model was introduced in this work showing all the contributions it has with its languages in describing and dealing with the development of distributed software systems. We have presented in this paper conceptual patterns for mobile and replicable elements to be used for complex, distributed and evolving architectures. These patterns can be served as guidelines for developing distributed software architectures. The patterns described were used to describe distribution issues in the distribution aspect of the PRISMA elements.

The patterns described in this work will be incorporated in the development of a case tool for the automatic generation of the PRISMA specification languages. These patterns will be options that a user using this tool chooses for the description of the distribution aspect in developing distributed, complex and evolving, software systems. Automatic code generation to programming languages will be done from our ADL to generate distributed applications.

References

[1] C. Szyperski, *Component software: beyond object-oriented programming*, (New York, USA:ACM Press and Addison Wesley, 1998).

[2] Aspect-Oriented Software Development, http://aosd.net

[3] JAVA Remote Method Invocation (RMI) Specification of SUN Microsistems: http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html

[4] CORBA Official Web Site of the OMG Group: http://www.corba.org/

[5] J. Pérez, I. Ramos, A. Lorenzo, P. Letelier and J. Jaén, PRISMA: OASIS Platform for Architectural Models(In Spanish),*Actas VII Jornadas de Ingeniería del Software y Bases de Datos, JISBD*, Escorial (Madrid), Spain, 2002,349-360.

[6] P. Letelier, P. Sánchez, I. Ramos and O. Pastor, *OASIS 3.0, A formal focus for the object oriented conceptual modelling(In Spanish)*(Universidad Politécnica de Valencia, SPUPV-98.4011, ISBN 84-7721-663-0, 1998).

[7] L. Nana, Y. Kermarrec and L. Pautet, GNATDIST: a configuration language for distributed Ada 95 applications: *ACM Tri Ada conference*, Philadelphia, USA, 1996, 63-72.

[8] J. Correas and F. Bueno, A Configuration Framework to Develop and Deploy Distributed Logic Applications: *ICLP01 Colloquium on Implementation of Constraint and Logic Programming Systems*, Cyprus, 2001.

[9] P. Ahlbrecht, S. Eckstein and K. Neumann, Language Constructs for Conceptual Modelling of Mobile Object Systems*: In Proc. 4th Int. Symp. on Collaborative Technologies and Systems, Simulation Series*, Orlando, USA,2003, 121-126.

[10] C. Canal, *A Language for the Specification and Validation of Software Architectures*, (In Spanish) P.H.D Thesis, Malaga, Spain, 2000.

[11] J. Kramer, J. Magee and M. Sloman, Configuring distributed systems: *In Proceedings of the 5th ACM SIGOPS European Workshop*, Mont St Michel,September 1992.

[12] J. Magee, N. Dulay, S. Eisenbach and J. Krammer, Specifying Distributed Software Architectures: *Proc. of the 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, 1995, 137-153.

[13] S. Soares and P. Borba, PaDA: A Pattern for Distribution Aspects: *In Second Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, 2002, 87-99.

[14] S. Soares, E. Laureano and P. Borba, Implementing Distribution and Persistence Aspects with AspectJ: *Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02*, Seattle, WA, USA,2002, 174-190

[15] J.L. Herrero, *Proposal of a Platform, Language and Design, for the Development of Aspect Oriented Applications*.(In Spanish)P.H.D. Thesis, Extremadura, Spain, 2003.

[16] J. Pérez, I. Ramos, J. Jaén, P. Letelier, E. Navarro, PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures: *3rd IEEE International Conference on Quality Software (QSIC 2003)*, Dallas, Texas, USA, 2003 (Accepted: Pending for Publication in Nov 2003).

[17] J. Jaén and I. Ramos, A Conceptual Model for Context-Aware Dynamic Architectures: *23rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with ICDCS2003*, Providence, Rhode Island, USA, 2003,138-146.

[18] L. Andrade and J. Fiadeiro, *Interconnecting Objects via Contrats*, OOPSLA´99.Tutorial 56.

ADVICE:

A JOURNAL ON ASPECT ORIENTATION

An Online journal serving the aspect orientation practice communities
www.onlinejao.org. Published by Houman Younessi, RISE ©ADVICE, 2004
ISSN: 1548-3851

Vol. 1, No. 1, June-August 2004

Designing Software Architectures with an Aspect-Oriented Language

Pérez, J; Ali, N; Carsí, J.A.; Ramos I.
Polytechnic University of Valencia,
Valencia, Spain.

Navarro, E.
University of Castilla-La Mancha
Albacete, Spain.

Abstract

Aspect Oriented Programming (AOP) has emerged in the last years as a new paradigm of sofware development. AOP provides techiques to improve the structure and reusability of the code. Moreover, this paradigm allows the reduction of the costs of maintainability and development of the final code by means of the separation of concerns. This is due to the encapsulation of the seperated code in a new reusable entity called *Aspect*. A great deal of aspect-oriented programming languages have emerged and have proved the advantages that this new paradigm of development offers. For this reason, the aspect-oriented approach is being introduced into the early phases of the life cycle of software development, at analysis and design. In this way, the development of software applications is improved from the beginning until the end of the process. In this work, we present in detail an aspect-oriented language to specify software architectures of complex, dynamic and distributed information systems. This aspect-oriented Architecture Description Language (ADL) follows the PRISMA architecture modeling approach, which integrates the advantages of Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD). Our ADL defines architectural elements at two levels of abstraction: the types definition level and the configuration level. The types definition level defines the PRISMA types (interfaces, aspects, components and connectors), and the configuration level specifies the topology for a specific architecture by instantiating types and connecting them. We demonstrate how it is possible to specify software

Citation: Pérez et al.; "Designing Software Architectures with an Aspect-Oriented Language", in *ADVICE:Journal of Aspect Orientation*, Vol. 1, No. 1, June-August 2004, pp. .....

Pérez J., Ali, N. Carsí, J. A., Ramos, I. Navarro, E. Designing Software Architectures with an Aspect-Oriented Language. ADVICE: Journal of Aspect Orientation, Vol. 1, No. 1, 2004.

architectures using aspects and how the development and maintainability processes of architectural models that are defined using PRISMA are improved.

1 INTRODUCTION

Nowadays, information systems are becoming more and more difficult to develop due to their complex structures, non-functional requirements and distributed and dynamic nature. Two approaches of software development have emerged to overcome these needs: Component-Based Software Development (CBSD) (Szyperski, 1998) and Aspect-Oriented Software Development (AOSD) (AOSD, 2004).

On the one hand, AOSD allows the separation of concerns by modularizing crosscutting concerns in a separate entity, the aspect. The encapsulation of the aspect allows the reusability of the same aspect in different objects and the evolution of an aspect without affecting the rest of objects and aspects. The main effort in this approach has been done at the implementation level by strongly emerging the Aspect Oriented Programming (AOP) as a new paradigm of sofware development. A great deal of aspect-oriented programming languages have been proposed and have been proven in order to demonstrate that AOP improves the structure and reusability of the code.

On the other hand, CBSD decomposes the system into reusable entities called components that provide specific services to the rest of the system. Finally, software architectures have emerged as a solution for design and development processes of complex information systems. There is a wide variety of architectural models, but among them there is no consensus about the different concepts and approaches.

PRISMA is an approach to develop complex information systems that has been designed taking into account different solutions of interest: architectures, CBSD and AOSD. This approach provides a model and an Architecture Description Language (ADL). The PRISMA model defines software architectures by integrating aspect-oriented software development (AOSD) and component-based software development (CBSD). This integration is directly reflected on its ADL. In this way, PRISMA ADL is presented as an aspect-oriented language to design complex, distributed, dynamic and reusable architectures. In this work, we specially focus on the demonstration of the improvement of the reusability, the development and the maintainability of PRISMA architectures using the PRISMA ADL.

Our research group has developed OASIS (Letelier et al., 1998). OASIS is a formal language to define conceptual models of object-oriented information systems that allows the verification, the validation and the automatic code generation from object-oriented conceptual models.

However, OASIS does not allow the specification of software architectures and the support of the non-functional requirements of complex information systems. The background of OASIS compiler allows us to reuse our compilation knowledge in the PRISMA Model. Moreover, the PRISMA language is based on OASIS to define the semantics of the architectural models in a formal way and to preserve the main advantages of OASIS language. In this way, we can validate and verify PRISMA architectural models and we will be able to automatically generate aspect-oriented code form the PRISMA ADL. Now, we are working on the first version of the PRISMA compiler to automatically generate C# code from the PRISMA ADL using .NET

Remoting technology. It is important to keep in mind that PRISMA language is independent of the technology and this will allow us to choose among different technologies at the time of generating the code. For this reason, we will be able to compile the same PRISMA architectural model into different aspect-oriented languages and technologies. In this way, we achieve to reduce the development time and to preserve the traceability between the architectural model and the application code.
The PRISMA ADL defines architectural elements at two levels of abstraction, the types definition level and the configuration level. The type definition level defines architectural types (interfaces, aspects, components and connectors) with a high level of abstraction, its main advantages are software reuse and complexity reduction by integrating components and aspects. In this way, we are able to show components definition as well as aspects description.
The configuration level designs the architecture of software systems by creating and interconnecting instances of the defined architectural elements. In other words, the architectural configuration of a specific software system is specified in this level.
The structure of the paper is the following: In section 2, a brief summary of related work is given. In section 3, the PRISMA architectural model is presented. In section 4, we introduce an example to illustrate the PRISMA architecture description language. In section 5, the PRISMA ADL is explained following the example in section 4. Finally, conclusions and further work round up this work.

2 RELATED WORK

A great deal of work has been done with architectural models, components, architecture description languages and other features related to the implementation of complex systems. Part of the work is focused on finding a consensus about the different concepts and approaches; however, due to the lack of a semantic commitment in the field, an important formalization effort is required to achieve a reasonable level of integration. We take into account different solutions of interest for our model: Component-Based Design and Aspect-Oriented Approach. The smooth integration of the considered solutions and the high abstraction level are the main advantages of PRISMA model.
The work developed by Garlan (Garlan and Kompanek, 2000) has been of great inspiration in our effort to define the first order citizens of our model. The work by Andrade & Fiadeiro (Andrade et al., 2002) on the notion of contract has been considered in our work incorporating some of the differences from (Monge et al., 2002) and (Bracciali et al., 2002) to create our connector type.
Our motivation is similar to the Model-Driven Architecture (MDA) approach proposed by the OMG group (OMG, 2004) for system interoperability and integration. MDA encourages modelling using UML extensions which are independent from the final implementation platforms. They propose separation of concerns and different abstraction levels in the modelling process. Transformation techniques and inter-level relationships are used in order to produce the final implementation.
A wide variety of models based on separation of concerns have been proposed ((Rashid, 2001), (Tarr et al., 1999), etc). However, the most extended is the Aspect-Oriented

Paradigm (AOP). Aspect-Oriented models can be classified into two different categories: static models and dynamic models. The static models are not able to change aspects and their weaving at run-time, whereas the dynamic ones permit this advantage. Examples of dynamic models are Mask Model (Sánchez, 1999) and Dynamic Aspect Oriented Platform (DAOP) (Pinto et al., 2003). The first one is based on objects and aspects. The second one combines components and aspects, but it does not provide an architectonical view of its models due to the fact that it does not define the connector architectural element. As Shaw present in her work (Shaw, 1994), the specification of information system with complex coordination protocols is so difficult without the connector. This is due to the connector provides the separation of the components interaction achieving a higher level of abstraction, modularity and architectonical view of the system.

A well accepted aspect-oriented programming language is AspectJ (Kiczales et al., 2001) which is an extension to Java where the code is separated into aspect and non-aspect code (objects). However, based on experiments implementing different aspects like persistence and distribution (Soares et al., 2002), this model has many drawbacks as aspects can be hardly reused and the model performs static weaving where adaptability at run-time is impossible. Therefore, work has been done in (Popovici et al., 2002) to provide dynamic weaving based on using the Java Virtual Machine Debugger Interface (JVMDI). Investigation has been done in (Pichler et al, 2003) to discover requirements needed to integrate Enterprise JavaBeans and AspectJ. In addition, the JAsCo language (Suvee et al., 2003) integrates AOSD and CBSD at the implementation level by extending the JavaBeans and introducing connectors to perform dynamic aspect weaving. Other aspect-oriented technologies that are important to mention are: AspectC#, Loom.Net, SetPoint and Aspect.NET. AspectC# (Kim, 2002) is an aspect-oriented extension of C# language; however, it weaves aspects statically and it is not prepared to develop complex applications. Loom.Net (Schult and Tröger, 2003) technology has two different versions: Loom.Net and Rapier-Loom.Net. The Loom.Net has a static weaving and the Rapier-Loom.Net is the improved version of the first one allowing a dynamic weaving. SetPoint (SETPoint, 2004) is a different technology in the sense that it uses semantics descriptors to build the weaving between aspects. Finally, it is important to mention the Aspect.NET (Safonov, 2003) project that is a Microsoft® project that it is modifying the CLR of VisualStudio.NET in order to provide support for aspect-oriented technology.

Mainly aspects have been carried out on the design level by extending UML meta-model as in (Suzuki and Yamamoto, 1999). Works like (Navasa, 2003) present guidelines on how separation of concerns can be handled by ADLs but no real work has been done applying these requirements.

In (Constantinides and Elrad, 2000) the main requirements for developing aspect-oriented software architectures are presented. They are the linguistic support for weaving aspects, the dynamic adaptability and the reusability. PRISMA and its ADL are presented as a good solution to specify aspect-oriented software architectures because they satisfy these requirements.

A wide variety of ADLs have been proposed and each one has advantages and disadvantages. An interesting comparison between these languages is done in the work by (Medvidovic and Taylor, 2000). However, these languages do not support an aspect-oriented specification of their architectures. As a result, PRISMA provides both the advantages of ADL's and the aspect-oriented techniques. Other ADLs also integrate

AOSD and CBSD such as TranSAT (Barais et al. 2004) and DAOP-ADL (Pinto et al., 2003). However they do not provide an architectonical view of its models due to the fact that they do not define the connector concept, they only consider components. Moreover, the aspect is not part of the components of TranSAT model, this language includes aspects refining the original components specification and generating another one that includes them. For this reason, this language only considers technical concerns.
Finally, it is important to keep in mind that PRISMA does not specify simple architectural system of the academia such as: pipelines, filters, blackboard, etc. PRISMA ADL is being used to specify industrial projects where the information systems are complex, open and active as the EFTCoR (Pérez et al., 2003b). EFTCoR is a robot that cleans the hulls of ships. Now, we are designing the EFTCoR teleoperation system usign the PRISMA ADL.

3 ARCHITECTURAL MODEL

The PRISMA architectural model allows the definition of architectures of complex software systems. One of its main contributions is the integration of the AOSD and the CBSD (D'Souza, 1999). In this way, PRISMA specifies different characteristics (distribution, security, context-aware, coordination, etc.) of an architectural element using aspects.
Most architectural models define their primitive architectural elements and specify their syntax and semantics. However, the PRISMA model, does not only describe the syntax and semantics of its architectural elements, but also defines the needed aspects to overcome the requirements of a software architecture.
A PRISMA architectural element can be analyzed from two different views, internal and external. The internal view (see Figure 1) shows an architectural element as a prism being an aspect of this architectural element each side of the prism. In this way, we represent that an architectural element of PRISMA is formed by a set of aspects; whereas, the external view (see Figure 2) is an architectural element which encapsulates its functionality as a black box and publishes a set of services that offers to the rest of architectural elements.

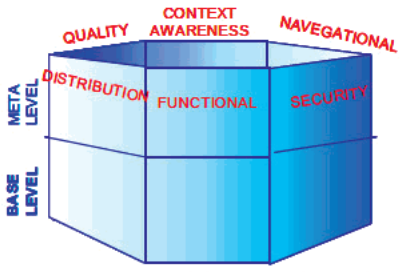


Figure. 1: Internal View of an architectural element



Figure. 2: External View of an architectural element

3.1. THE INTERNAL VIEW OF PRISMA

The Aspect-Oriented technique was first introduced in (Kizcales et al. 2001) by defining aspects as properties of a system that crosscut the basic functionality of the system, achieving a reduction in tangling code.

The concept of an aspect at the implementation level is very clear. However, not much work has been done concerning aspects at an architectural level. In our work, an aspect represents a specific view of the software architecture. For this reason, a PRISMA architectural element is formed by a set of aspects that describe it from the different views of the architecture. For example, the internal view in Figure 2 could represent an architectural element that belongs to a web-distributed information system which takes into account context awareness (Jaén and Ramos, 2003), quality (Pérez et al., 2003a), distribution (Ali et al., 2003), navigation and security concerns. However, in other information systems, their components will have other aspects.

A lot of aspects can be used to describe architectural elements that vary in different application domains. In this paper, we are going to focus on the following aspects:

- **Functional Aspect**: The functional aspect captures the semantics of the information system by defining its attributes and behaviour.
- **Coordination Aspect:** The coordination aspect defines the synchronisation among elements during their communication. Moreover, the choreography or protocol is defined, extending the "contract" concept of Andrade & Fiadeiro (Andrade et al., 2002).
- **Distribution Aspect:** The distribution aspect specifies the features that define the dynamic location of the instances (Ali et al., 2004). It specifies services to obtain metric values in order to implement complex distribution strategies as the ones shown in ((Cheng et al., 2002), (Zinky et al., 1997)). It defines the distribution policies and strategies in order to optimize a distribution topology of the architecture through applying mobility patterns (Ali et al., 2003).

Simply defining an aspect (crosscutting concerns) is not enough. The methods indicating how aspects are weaved (connected) with the rest of the system have to be determined. In most aspect-oriented programming languages, the weaving information is specified in the aspect by the pointcuts and the methods of the advice. These lack the reusability of the aspects in different objects due to the fact that aspects are specifically defined for the determined objects.

In PRISMA, the weaving indicates that the execution of an aspect service can generate the invocation of services in other aspects. Nevertheless, in order to preserve the independence of the aspect specification from the rest of aspects and weavings, the weaving is specified externally.

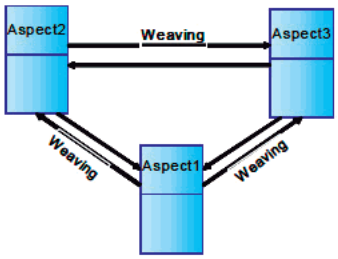


Figure. 3: Aspect weaving

As a PRISMA architectural element is formed by a set of aspects (functional, distribution, etc.), the weaving is part of the architectural elements specification. In this way, we achieve the reusability of the aspects not only in the same architectural model but also in different architectural models. Figure 3, shows that aspects are specified independently from the architectural elements they will form and that the weaving is the glue of the aspects forming a prism (Internal View). As a result, we achieve that PRISMA aspects are completely reusable and the maintainability of PRISMA architectural models is improved. This last advantage is due to a change in a specific concern (e.g: distribution) only affects to the aspect that implements the concern. However, without applying AOSD, the specification of this concern will be tangled with the application code (distribution and functional attributes and methods are mixed) and it will be very difficult to find all the needed updates.

The weaving methods are operations that describe the temporal order of the weaving process. For simplicity, we present the **after**, **before** and **instead** methods which are typical of the AOP whose semantics are as follows:

- **after**: aspect1.service is executed after aspect2.service
- **before**: aspect1.service is executed before aspect2.service
- **instead**: aspect1.service is executed in place of aspect2.service

3.2. THE EXTERNAL VIEW OF PRISMA

PRISMA has two architectural elements that are first citizens of the language: components and connectors. In this section, the PRISMA architectural elements and their connection relationship are briefly presented in order to focus on the aspect orientation of the language.

3.2.1. Components

A PRISMA component is an architectural element which captures the functionality of the information system and does not act as a coordinator between other architectural elements. It is formed by an identifier, a set of aspects, their weaving relationships and the inports and outports whose type is a specific interface (a set of services).

An inport specifies services of an interface that a component *offers to* the rest of the architectural elements (server behaviour). An outport specifies services of an interface that the component *requires from* the rest of architectural elements (client behaviour).

3.2.2. Connectors

A PRISMA connector is an architectural element which acts as a coordinator between other architectural elements. It is formed by an identifier, a set of aspects, their weaving relationships and one or more inroles and outroles, whose type is a specific interface. These roles represent interaction points among components.

3.2.3. Attachments

A PRISMA connection relationship among the architectural elements is called *attachment.* The attachment relationship establishes the connection among ports of components and roles of connectors. It is important to keep in mind that an attachment always connects outports and inroles as well as outroles and inports and the connection is established by the >> symbol, which indicates the direction of the data sent. The syntax is the following:

```
Attachments
        <name_of_component>.<name_of_(in |out) Port> ( >> | <<)
        <name_of_connector>.<name_of_(in | out) Role>;
End_Attachments;
```

4 EXAMPLE: A BANK SYSTEM

We are going to illustrate the architecture description language of PRISMA with a very simple bank system. We want to define a system which contains a client-server architecture between two components. In this simple bank system, we are going to present the coordination, distribution and functional aspects.
There are two components that are involved in the banking system and are connected between them: an *ATM* and an *Account*. The ATM is the client component that requires *withdrawal* and *balance* services from the *Account*. The *Account* is the server component which offers the required services of the *ATM.* The *ATM* stores an ATM number which identifies it (*numberId)*, the amount of available money in the ATM (*money*) and where the ATM is located (*address).* The *Account* stores an account number which identifies it (*numberId),* the amount of money (*money)* and the postal address of its owner (*address)*. Moreover the system has a connector *ATMAccount* that controls the communication process between both components.
We are going to assume that the information is stored in the main memory in order to simplify the example without having to consider the persistence aspect.

5 AN ASPECT-ORIENTED LANGUAGE TO DESIGN SOFTWARE ARCHITECTURES

The PRISMA ADL defines the architectural elements at different levels of abstraction: the types definition level and the configuration level. The types definition level defines

architectural types (interfaces, aspects, components and connectors) with a high abstraction level. Its main advantages are software reuse and complexity reduction by integrating components and aspects. In this way, we show both component definition and aspect description. The PRISMA types defined at this level are stored in a PRISMA library so that they can be reused by other types or specific architectures.

The configuration level designs the architecture of software systems by creating and interconnecting instances of the defined architectural elements in the previous level. In other words, at this level we specify the topology of a specific software system. The language at both levels of abstraction is presented in detail following the example presented in section 4.

5.1. THE TYPES DEFINITION LEVEL

The types definition level of PRISMA defines the first-class citizens of the language: interfaces, aspects, components and connectors. They are stored in the PRISMA library in order to be reused. The fact that interfaces and aspects are first-class citizens of the language increases the reusability because an interface may be used by several aspects and an aspect may be used by several architectural elements. In addition, components and connectors can be reused in different architectural models.

5.1.1. Interfaces

Interfaces publish services without taking into account the ports, roles and aspects that are going to use them. Next, we present some interfaces of the banking system.

```
Interface ICreditCardTransactions
   withdrawal(Quantity: currency): currency;
   balance():Currency;
   changeAddress(NewAdd: string);
End_Interface ICreditCardTransactions;


Interface IMobility
   move(NewLoc:loc);
End_Interface IMobility;
```

5.1.2. Aspects

A common syntax of aspects has been defined. This syntax is based on OASIS and, for this reason, it is constituted by the sections that are presented bellow. This common syntax is going to be used to present the functional aspect.

- **Functional Aspect**

  The head of an aspect specifies its name and the kind of characteristic it defines: *functional, distribution, coordination,* etc. Moreover, interfaces whose semantic is defined by the aspect are detailed next to the *using* reserved word. We are going to define a functional aspect which specifies the semantics of the *ICreditCardTransaction* interface (see Table 1, section 1).

  Attributes are specified inside an aspect. These attributes are necessary to store

information about the characteristics of the aspect. Attributes are preceeded by the *Attributes* reserved word and they have a name and a type (see Table 2, nº 2). This type defines the kind of values that the attribute can store.
An aspect defines the semantics of services. The set of services specified in an aspect must contain the *begin* service, the *end* service and the interface services that this aspect uses (see Table 2, nº 3). *Begin* and *end* services do not mean that it is possible to instantiate an aspect, they make a reference to the new and destroy services of the architectural element that the aspect will belong to. The semantics of services is defined by means of preconditions and valuations using dynamic logic (Harel, 1984). *Preconditions* establish the condition that must be satisfied to execute a specific service (see Table 2, nº 5). *Valuations* specify the changes in the attribute values by the execution of services (see Table 2, nº 4).
With regard to services, it is important to take into account that the same service can have two different behaviors: client and server. The client behavior is when a service is invoked to another architectural element. The server behavior is when a service is offered and processed by an architectural element. Sometimes, it is necessary to distinguish between these two behaviors. The syntactical difference between client or server service is the reserved word *in* and *out.*
Finally, we want to emphasize that the language assumes that each service has its corresponding response service. A response service is a service which returns the result of a specific service. We do not specify them explicitly in the aspects and interfaces; however, it is possible to specify whenever it is necessary.
The aspect example shown above the precondition to execute the *withdrawal* service and the valuation of the *money* attribute when this service is executed. The precondition ensures that there is enough money to be able to withdraw the required quantity and the valuation update quantity of available money. Services, preconditions and valuations are preceeded by the reserved words *Services, Preconditions* and *Valuations,* respectively.
In addition, it is possible to specify triggers and operations. A *trigger* generates the invocation of a service if its condition is satisfied. The syntax to specify a trigger is the following:

```
triggers
      name_of_the_invoked_service when { boolean_condition}
```

An *operation* is a process that establishes a sequence of actions that must happen. An operation is composed by more than one service and it could be executed in a transactional way (atomic = all or nothing) or not (it is possible to execute part of the operation without finishing the execution of every service that composes the operation). If you want an operation to be executed in a transactional way you must add at its specification the *transactional* reserved word. The syntax to specify an operation is the following:

```
operations [transactional]
 transaction_name:
  name_of_current_state =
               name_of_the_service.name_of_the_following_service,
...;
```

Table 1: *BankInteraction* Functional Aspect

```
1 Functional Aspect BankInteraction using ICreditCardTransaction
2   Attributes
       numberId: number;
       customer: string;
       address: string
       money: currency;
3   Services
       in/out withdrawal(Quantity: currency):currency;
4              Valuations
                   [in withdrawal(Quantity]
                     money = money - Quantity;
       in/out balance():Currency;
       int/out changeAddress(NewAdd: string);
4              Valuations
                  [in changeAddress(NewAdd] address = NewAdd;
5   Preconditions
       in withdrawal(Quantity)
           if Quantity <= money;
6   Protocols
       BANKINTERACTION = begin.TRANSACTION;
       TRANSACTION = in/out withdrawal(Quantity).TRANSACTION +
                     in/out balance.TRANSACTION +
                     in/out changeAdress(NewAddress).TRANSACTION
                     end;
End Functional Aspect BankInteraction;
```

Finally, it is necessary to specify the protocol to describe the order and the state in which a service could be executed (see Table1, nº6). The protocol is a textual specification of a transition state machine. The graphical representation of the protocol example is the following:

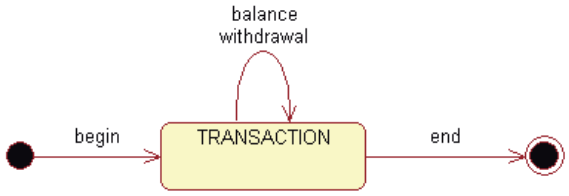


Figure. 4: Graphical specification of protocol

- **Coordination Aspect**
  This aspect introduces two sections that are unique to the coordination aspect, they are the *Synchronizations* and the *Choreography*.
  Synchronizations coordinate service executions. These services belong to an interface that the coordination aspect specifies. Synchronizations are specified using poliadic π-calculus (Milner, 1991). This language allows us to describe architectural models whose services can be executed concurrently. The poliadic π-calculus is a generalized version of the monadic π-calculus, which allows us to send a tuple of parameters through the link instead of only one parameter as in monadic π-calculus. We show the client-server coordination by means of the

services *withdrawal* and *balance* (see Table 2, nº 3). It is important to emphasize that response services are specified explicitly in the synchronizations adding *response* to the name of the service.

Table 2: *BankCoordination* coordination aspect

```
1 Coordination Aspect BankCoordination using ICreditCardTransaction
2   Services
      withdrawal(Quantity: currency): currency;
      balance():string;
3  Synchronizations
     CLIENT=(withdrawal(Quantity)?.ICreditCardTransaction →
             responseWithdrawal(Quantity)!.
             ICreditCardTransaction)
            ||
            (balance()?.ICreditCardTransaction →
             responseBalance()!.ICreditCardTransaction)
             ||
             … …;
     SERVER= (withdrawal(Quantity)!.ICreditCardTransaction →
              responseWithdrawal(Quantity)?.
              ICreditCardTransaction)
              ||
              (balance()!.ICreditCardTransaction →
              responseBalance()?.ICreditCardTransaction);
               ||
               … …;
4   Choreography
    (withdrawal(Quantity)?.ICreditCardTransaction →
     withdrawal(Quantity)!.ICreditCardTransaction →
     responseWithdrawal(Quantity)?.ICreditCardTransaction →
     responseWithdrawal(Quantity)!.ICreditCardTransaction)
    ||
    (balance()?.ICreditCardTransaction →
     balance()!.ICreditCardTransaction →
     responseBalance()?.ICreditCardTransaction)→
     responseBalance()!.ICreditCardTransaction)
    ||
  … …);
End_Coordination Aspect BankCoordination;
```

The synchronization is a subprocess that defines the behaviour of a component from the point of view of the connector. There must be the same number of synchronizations as interfaces used by the coordination aspect. This is due to the fact that each interface will be associated to an inrole or/and outrole of the connector that will be formed by the coordination aspect that we have defined. In addition, each inrole and outrole will be attached with a component. As a result, we are able to synchronize the ports of the components attached to the roles of the connector by synchronizing their interfaces.

For instance, the CLIENT synchronization specifies the role of the *ATM* component from the point of view of the *ATMAccount* connector and the SERVER synchronization specifies the role of the *Account* component from the point of view of the *ATMAccount* connector.

The *Choreography* specifies the coordination process among components, the glue of synchronizations. For example, it could be the parallel composition between the management of two different services (*withdrawal and balance)* and

the sequence of messages associated to a specific service (e.g: the client request the *withdrawal* service, the connector invokes to the server the *withdrawal* service, the server returns the result and finally, the connector sends the result of the *withdrawal* service to the client) (see Table 2, nº 4).

- **Distribution Aspect**
  This aspect introduce the same sections of a functional aspect with some predefined attributes and services.
  The distribution aspect specifies the locations of the instances. However, if the location is specified in the distribution aspect, the same distribution aspect cannot be reused in different architectural elements if instances are distributed in different locations. This problem is solved using the PRISMA ADL due to that it is separated into the Type Definition Language and Configuration Language. Thus, the Type Definition Language specifies that a type has a location without assigning it a value. The value of the location is assigned when types are instantiated in the Configuration Language.

Table 3: *ExtMbile* distribution aspect

```
Distribution Aspect ExtMbile usign IMobility;
   Attributes
      id: nat;
      location: LOC NOT NULL;
   Services
     move(NewLoc:LOC);
      Valuations
          [move(NewLoc)] location:= NewLoc;

Protocols
         EXTMBILE ≡ begin.EXTMBILE;
         MOVEMENT ≡ move.EXTMBILE +
                    end;
End Distribution Aspect ExtMbile;
```

The distribution aspect presented in Table 3 specifies the behaviour of a mobile architectural element enabling the change of the location attribute by the move service *valuation*. The attribute *location* has as an abstract data type called *LOC*. This data type hides the different mechanisms of locations of an architectural element at a physical level, e.g. it can be a URL, an IP, etc. The *location* attribute should have a value when an architectural element is instantiated, this is indicated by the *NOT NULL*.

5.1.3. Components, Connectors and Attachments

A simple architectural element is specified with the set of ports/roles whose type is an interface, the aspects it is formed of, and the aspects' weaving. The aspects are reused by importing them from the PRISMA library (types defined in the types definition level). The aspect weaving is specified by the aspects that participate in the weaving, the services of the aspects where they are weaved and the weaving methods (after, before and around). In order to better understand how to specify components and connectors, we

show their syntax by means of the specification of the *ATM* and *Account* components and the connector that connects them.

The specified aspects in this paper are going to be reused to define the architectural elements of the example: the *ExtMbile* distribution aspect (see Table 3) to define the three architectural elements, the *BankInteraction* functional aspect (see Table 1) to define the *Account* and *ATM* components (see Table 4) and finally, the *BankCoordination* coordination aspect (see Table 2) to define the connector *ATMAccount* (see Table 5). However, it is important to take into account that these aspects could be reused to create other architectural elements. We present the synchronization between aspects in the *Account* component; for example, when the customer address changes (*ChangeAddress*), the account of this customer must be moved to another place nearer to his/her new address (see weaving section in *Account* component of Table 4).

Table 4: Definition of components

```
Component_type Account
  Inport
    ATMTrans: ICreditCardTransactions
  End_Inport;
  Outport
    AccountResult: ICreditCardTransactions
  End_Outport;
  Functional Aspect Import BankInteraction;
    Weaving
      Distribution Aspect
         changeAddress(NewAdd: string) before move(NewAdd: string);
    End_Weaving;
  Distribution Aspect Import ExtMbile;
End_Component_Type Account;
Component_type ATM
  Inport
    VISACreditCard: ICreditCardTransactions;
  End_Inport;
  Outport
    AccounTrans: ICreditCardTransactions
  End_Outport;
  Functional Aspect Import BankInteraction;
   Distribution Aspect Import ExtMbile;
End_Component_Type ATM;
```

Table 5: Definition of connectors

```
Connector_type ATMAccount
   Inport
      ATM: CreditCard_Transactions
   End_Inport;
   Outport
      Account: CreditCard_Transactions
   End_Outport;
   Coordination Aspect
    Import BankCoordination:Coordination_Aspect;
   Distribution Aspect
     Import ExtMbile: Distribution_Aspect;
End_Connector_Type ATMAccount;
```

5.2. THE CONFIGURATION LEVEL

The configuration level is used to define instances and the topology of the architectural model. In order to do this all needed connectors, components and attachment types must be imported. Then, all the necessary instances of the imported types can be defined and, finally, the topology of the model is expressed by interconnecting such instances. Such an interconnection process is achieved by creating instances of attachments.

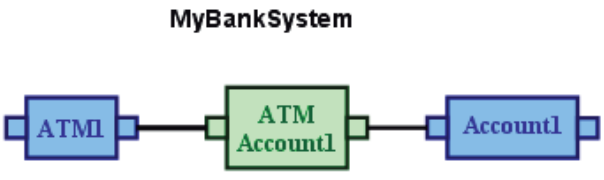
Table 8: Architectural model of a simple bank system

```
Architectural Model  SimpleBankSystem
   Inport
      SystemInput: ICreditCardTransactions;
   In_Inport;
   Outport
      SystemResponse: ICreditCardTransactions;
   End_Outport;
   Import_Types
       Components ATM, Account, Administrator;
       Connectors ATMAccount, AdministratorATM;
   End_Import_Types;
   Instances
       ATM1: ATM;
       Account1: Account;
       ATMAccount1: ATMAccount;
   End_Instances;
   Attachments  ATMAccountAttach
    ATM1.AccounTrans >> ATMAccount1.ATM;
    ATMAccount1.Account >> Account1.ATMTrans;
    Account1.AccountResult >> ATMAccount1.ATM;
    ATMAccount1.Account >> ATM1.VisaCreditCard;
   End_Attachments;
End Arquitectural Model SimpleBankSystem;
```

We must define the connection relationships among the architectural elements' instances of the bank information system (attachments). The attachment relationship establishes the connection among ports of components' instances (ATM, Account) and roles of connectors' instances (ATMAccount) (see attachments section in Table 5). It is important to keep in mind that an attachment always connects inports and outroles as well as outports and inroles (see Table 6).

Figure 5 illustrates the specific architecture of this Bank system. As Table 8 shows, the components and the connector that it includes and the relationships among them are instantiated.

Figure. 5: Graphical Architectural of the bank system



6 CONCLUSIONS AND FURTHER WORKS

In this paper, a innovate Aspect-Oriented Architecture Description Language (AOADL) to specify complex, dynamic and distributed information systems has been presented. This language allows us to define PRISMA architectural models. The structure, design and maintainability of PRISMA architectures are improved by reusing elements at different levels of granularity (interfaces, aspects, components and connectors). This reusability is achieved by means of the division of the language into two levels of abstraction and the integration of AOSD and CBSD into the language.

The types definition level of the ADL allows us to store interfaces, aspects, components, and connectors in a PRISMA library in order to enable their reusability. Next, all types stored in the library can be reused by the configuration level to define a software architecture.

We are currently working on the *PRISMA: Model Compiler of Aspect-Oriented Component-Based Software Architectures* project of Microsoft® Research Cambridge. This project consists in the specification of industrial systems such as the EFTCoR teleoperation system (Pérez et al., 2003b). EFTCoR is a robotic platform to clean the hulls of ships that has strong requirements in terms of adaptability to different devices, safety for operators, reponse time, etc. We want to provide reusable and dynamic architectures for systems of this kind using aspect-oriented techniques. Currently, we are designing the EFTCoR teleoperation system usign the PRISMA ADL and developing the compiler which will allow us to generate automatically C# code from the PRISMA ADL using the technology .NET Remoting.

As further works, we are going to extend the ADL in order to support evolution in run-time and provide its graphical notation to desing architectures in a friendy way.

The incorparation of evolution features into the language will permit to reconfigure dynamically the PRISMA architectures without stopping the application execution. With regard to the graphical notation, we have selected UML and we are studying the correspondence between our types and its UML representations. We have available a first version of a UML (v 1.5) profile for PRISMA (Pérez and Ramos, 2003) and in the near future, we are going to adapt this profile to UML 2.0 and to present our graphical notation.

ACKNOWLEDGMENTS

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, DYNAMICA project TIC2003-07776-C02-02. Moreover, its is development is founded by the Microsoft Research Cambridge, PRISMA: Model Compiler of Aspect-oriented component-based software architectures Project.

BIBLIOGRAPHY

Ali N., Pérez J., Ramos I. (2004); “High Level Specification of Distributed and Mobile Information Systems”; Second International Symposium on Innovation in Information & Communication Technology (ISSICT), 21-22 Abril, Amman, Jordania

Ali N.H., Silva J., Jaen J.,Ramos I., Carsi J.A., Perez J. (2003): "Mobility and Replicability Patterns in Aspect-oriented Component-Based Software Architectures"; *Proceedings of 15th IASTED , Parrallel and Distributed Systems, Acta Press,* Marina del Rey, C.A.,USA, November, ISBN: 0-88986-392-X, ISSN:1027-2658, pp. 820-826.

Andrade L. F., Fiadeiro J. L., Gouveia J., Koutsoukos G., Wermelinger M. (2002); "Coordination for Orchestration"; *5th International Conference on Coordination Models and Languages COORDINATION*, Lecture Notes in Computer Science 2315 Springer, York, UK, April 8-11, ISBN 3-540-43410-0, pp. 5-13.

AOSD (2004); *Aspect-Oriented Software Development*, http://aosd.net

Barais O., Cariou E., Duchien L., Pessemier N., Seinturier L. (2004); "Transat: A framework for the specification of software architecture evolution". *In* ECOOP First *International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04)*, Oslo, June. http://wcat04.unex.es/. bib

Bracciali A., Brogi A., Canal C. (2002); "Sistematic Component Adaptation"; *In Proc IDEAS*, La Habana, Cuba.

Cheng S., Garlan D., Schmerl B., Steenkiste P., Hu N. (2002); "Software Architecture-based Adaptation for Grid Computing" , *IEEE Conference on High Performance Distributed Computing*, *IEEE Computer Press*, Scotland, July.

Constantinides C.A., Elrad T. (2000); "On the Requirements for Concurrent Software Architectures to Support Advanced Separation of Concerns"; *In Proceedings of OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. Available at: http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/constantinides.pdf

D'Souza D., Wills A. (1999); "Objects, Components and Frameworks with UML: The Catalysis approach"; *Addison-Wesley*.

Garlan D., Kompanek A. (2000); "Reconciling the Needs of Architectural Description with Object-Modeling Notations"; *Proceedings «UML» 2000*, Springer, York, UK, October 2-6, LNCS 1939, pp.498-512.

Harel D. (1984); "Dynamic Logic"; *in Handbook of Philosphical Logic II,* editors D.M. Gabbay, F. Guenthner; pp.497-694, Reidel.

Jaén J., Ramos I. (2003); "A Conceptual Model for Context-Aware Dynamic Architectures"; *3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with ICDCS*, May 19-22, Providence, Rhode Island USA.

Kiczales G., Hilsdale E., Huguin J., Kersten M., Palm J., Griswold W.G. (2001); "An Overview of AspectJ", *In proceedings of the European Conference on Object-Oriented Programming*, Springer-Verlag.

Kim H. (2002); "AspectC#: An AOSD implementation for C#", *Msc. Thesis,* Trinity College, Dublin, November.

Letelier P., Sánchez P., Ramos I., Pastor O. (1998); "OASIS 3.0, A formal focus for the object oriented conceptual modelling"; *Polytechnic University of Valencia, SPUPV-98.4011*, ISBN 84-7721-663-0 (In Spanish).

Medvidovic N., Taylor R. N. (2000); "A classification and Comparison Framework for Software Architecture Description Languages"*; IEEE Transactions of SW Engineering*, Vol. 26, nº 1, January.

Milner R. (1991); "π- Cálculo Poliadico: A tutorial".

Monge R., Alves C., Vallecillo A. 2002; "A Graphical Representation of COTS-based Software Architectures"; *In Proc IDEAS*, La Habana, Cuba.

Navasa, A., Perez, M.A., Murilllo J.M., Hernandez, J. (2003); "Aspect Oriented Software Architecture: a structural perspective"; *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop collocated to the 1st International Conference on Aspect-Oriented Software Development*, Enschede , The Netherlands, 22 April

OMG (2004); Object Management Group (OMG), http://www.omg.org

Perez J., Ramos I., Jaén J., Letelier P., Navarro E. (2003a); "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures";. In proceedings of 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November, © IEEE Computer Society Press ISBN 0-7695-2015-4, pp. 59-66.

Pérez J., Ali N., Ramos I., Pastor J.A., Sánchez P., Álvarez B. (2003b); "Tele-operated Systems Development using the PRISMA approach", VIII conference on Software Engineering and Databases, Alicante, Spain (in spanish).

Pérez J., Ramos I. (2003); "Oasis: Formal Support to Define Hypermedia, Dynamic and Distributed Models"*,* Technical Report DSIC-II/22/03, Polytechnic Universtiy of Valencia, octubre

Pichler R., Ostermann K., Mezini M. 2003; "On Aspectualizing Component Models"; *In Software Practice and Experience,* Volume 33, Issue 10, Wiley Publishers, 957-974.

Pinto M., Fuentes L., Troya J.M. 2003; "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development", *Generative Programming and Component Engineering: Second International Conference, GPCE*

2003, Erfurt, Germany, September 22-25, Springer Verlag Computer Science, ISSN: 0302-9743,

Popovici A., Gross T., Alonso G. 2002; "Dynamic Weaving for Aspect-Oriented Programming"; *In proceedings of the 1st international conference on Aspect-oriented software development,* Enschede, The Netherlands, April.

Rashid A. 2001; "A Hybrid Approach to Separation of Concerns: The Story of SADESK"; *Proc. Reflection conf.*, Springer-Verlang, LNCS, 2192, pp. 231-249

Safonov V. O. 2003; "Aspect.NET - a new approach to aspect oriented programming"; *.NET Developers Journal*, April.

Sánchez F. 1999; "Mask Model: Towards adaptability of synchronization restrictions in LCOO", *PhD. dissertation, Extremadura University,* Spain.

Schult W., Tröger P. 2003; "Loom.NET - an Aspect Weaving Tool"; *In Workshop on Aspect-Oriented Programming, ECOOP'03*, Darmstadt.

SETPoint 2004; *SETPoint,* http://www.dc.uba.ar/people/proyinv/setpoint/

Shaw M. 1994; "Procedure Calls Are the Assembly Language of Software Interconection: Connectors Deserve First-Class Status", *Proceedings of Workshop on Studies of Software Design*, Enero.

Soares S., Laureano E., Borba P. 2002; "Implementing Distribution and Persistence Aspects with AspectJ"; *Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02*, Seattle, WA, USA, 174-190.

Suvee D., Vanderperren W., Jonckers V. 2003; "JasCo: an Aspect-Oriented approach tailored for Component Based Software Development"; *In proceedings of the 2nd international conference on Aspect-oriented software development*; Boston Massachusetts, March.

Suzuki J., Yamamoto Y. 1999; "Extending UML with Aspects: Aspect Support in the design phase"; *3rd Workshop on Aspect-oriented Programming en European Conference on Object Oriented Programming(ECOOP)*, 299-300

Szyperski C. (1998); "Component software: beyond object-oriented programming", *ACM Press and Addison Wesley*, New York, USA.

Tarr P., Ossher H., Harrison W.H., Sutton S. M. 1999; "N Degrees pf Separation: Multi-Dimensional Separation of Concerns"; *Internacional Conference on Software Engineering (ICSE),* ACM, pp. 1907-119.

Zinky J., Bakken D., Schantz R. (1997); “Architectural Support for Quality of Service for CORBA Objects”; *Theory and Practice of Object Systems 3(1)*.

ABOUT THE AUTHOR(S)



Jennifer Pérez Benedí was graduated in Computer Science of Valencia University of Technology in 2001. She has a research fellowship founded by the Spain government and she is the project manager of PRISMA: Model Compiler of Aspect-Oriented Component-Based Software Archictures Project of Microsof Research Cambridge. Her research interests are software architectures, Component-Based Software Devolpment, Aspect-Oriented Software Development, software evolution and model compilers. Her e-mail address is jeperez@dsic.upv.es



Nour Ali was graduated in Computer Science of Bir-Zeit University in Ramallah in 2002. She is a second year Phd student in the Department of Information Systems and Computation of of the Polytecnic University of Valencia. Her current interests are distributed systems, software architectures, aspect-oriented software development, automatic code generation and software engineering. Her email is nourali@dsic.upv.es



José A. Carsí Cubel received his PhD degree in computer science from the Polytechnic University of Valencia, in 1999. He is currently an associate professor of computer science at Technical University of Valencia. His research interest are component models, ADLs, model management and software evolution. He can be reached at pcarsi@dsic.upv.es



Isidro Ramos Salavert is a full professor in Computer Science at the Polytechnic University of Valencia. His research interest are CBSD, AOSD, Software Architectures, model management and software evolution. His e-mail address iramos@dsic.upv.es



Elena Navarro is an assistant professor in the Computer Science Department in the University of Castilla-La Mancha. Previously, she was working as a researcher in the Informatics Lab in the Agricultural University of Athens contributing to the European Project CHOROCHRONOS. In addition, she was working as staff in the Regional Government of Murcia. Her e-mail address enavarro@info-ab.uclm.es.

Goals and Quality Characteristics: Separating Concerns*

Elena Navarro
Computer Science Department
University of Castilla-La Mancha
Avda. España S/N, Albacete, Spain
enavarro@info-ab.uclm.es

Patricio Letelier, Isidro Ramos
Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n, Valencia, Spain
{letelier, iramos}@dsic.upv.es

Abstract

Software Requirements Specification (SRS) organization for complex and/or large systems have to do with several not faced challenges until the moment. This organization is a key factor to facilitate the quality assurance of the SRS, regarding features as: correctness, completeness, consistency and modifiability. Organization is also crucial for an effective exploitation of the SRS when elaborating other related or derived artefacts. Although there is a consensus about SRS content, this is not applicable to the organization. Nevertheless, it is evident that depending on the system, their stakeholders and the activities to perform with the SRS, the relevant criteria for SRS organization and presentation can be different. Additionally, another of the main problems to be solved is related to the crosscutting of requirements that produces tangled specifications. This work faces these issues: the organization of SRS by applying Aspect Oriented techniques to properly manage the crosscutting. A Goal Oriented approach for requirements allows us to establish traceability from software goals to specific requirements and from the latter to operationalizations that are realized as software components. In this work, we present an integration of aspect and Goal Oriented approaches, to properly manage the SRS organization and presentation. Furthermore, our proposal uses the standard ISO/IEC 9126 as the starting point to organize goals and requirements. ATRIUM, a methodology to concurrently define requirements and software architecture, provides the setting for our proposal.

1. Introduction

IEEE 830-1998 [11] recommendations are the milestone concerning the contents which are mandatory in a Software Requirements Specification (SRS). This standard suggests several possible organizations for specific software requirements. Nevertheless, when the amount of requirements and/or the complexity are considerable, those suggestions are not enough. Requirements organization and presentation are crucial to facilitate their maintenance and ensure other desirable features such as: correction, completeness, consistency and traceability. Along the requirements elicitation and specification process, several stakeholders are involved (both from the customer and technical side) every one with their own interests and views of the system, which ought to be rightly represented and reconciled in the SRS.

A requirement is a capacity that software should exhibit or a condition that should be met. This capacity or condition can be expressed at different abstraction or detail levels. Concretely, we distinguish two levels: *goals* and *requirements*. Goals allow one to establish, by means of refinement and composition processes, a derivation graph from software interests or goals until specific requirements. On the other hand, *requirements* are detailed enough as to be assigned to a software component and lately be verified. *Goal Oriented Requirements Engineering* [13] employs this strategy to perform this refinement from goals until requirements and their subsequent operationalization in software elements.

In the literature, different paradigms cope with the *separation of concerns* of a system in order to provide support for evolution adaptability, comprehensibility, etc. These *concerns* can range from non-functional features, as security or fault tolerance, to functional features. Aspect Oriented Software Development (AOSD) [1] is one of these approaches. AOSD provides a set of techniques that allow managing those interests that appear scattered along the system and crosscut several elements. AOSD identifies these *concerns* realizing them as *aspects* and managing them explicitly.

In the AOSD context, several proposals have been introduced at different abstraction levels, both during implementation [12] and design [21] that define an aspect as an additional constructor of the language. It is associated to the constructor "class" to manage the crosscutting that can appear in the methods specification for a class. Several works have also been proposed at the requirements level as Aspect Oriented Requirements Engineering (AORE) [20]. In this proposal, an aspect is a requirement which is related to a set of other requirements, but that is separately specified. By means of

* This work has been funded by the Spanish CICYT project DYNAMICA TIC2003-07776-C02-02

Navarro, E., Letelier, P., Ramos, I. Goals and Quality Characteristics: Separating Concerns. Proc. of Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004. 2004.

a technique known as *weaving*, it manages this crosscutting.

The aim of this work is to present our proposal to organize software requirements, by integrating Aspect Oriented techniques within a Goal Oriented approach for requirements, as we previously sketched in [17]. Furthermore, the ISO/IEC 9126 [10] standard is used as a starting point to define the concerns. ATRIUM [16], a methodology to concurrently define requirements and software architecture, provides the setting for our proposal.

The reminder of this work is organized as follow. Section 2 briefly describes the ATRIUM methodology. Section 3 introduces the ISO/IEC 9126 standard and how it is used in our proposal. Section 4 presents in detail the set of tasks that drive the goals model definition. The way we apply the set of tasks to a case study is shown in section 5. Some related works are described in section 6. Eventually, the achieved conclusions and future works round up the work.

2. ATRIUM: Requirements and Software Architectures

ATRIUM is a methodology oriented to the concurrent definition of Software Architectures (SA) and Requirements. In ATRIUM, decisions at architectural level are made to satisfy specific software requirements. With this aim, ATRIUM provides the analyst with guidance, along an iterative process, from an initial set of user/system needs until the instantiation of the architecture, specified by means of a PRISMA model [18]. PRISMA is an architecture description language that allows us to define dynamic architectures.

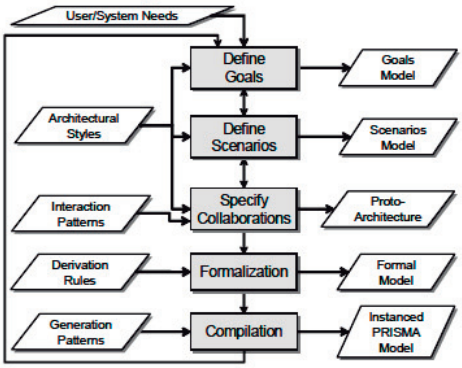


Figure 1 ATRIUM: Activities and Artifacts

ATRIUM entails five activities (Figure 1) to be iterated on in order to define and refine the different artifacts and allow the analyst to reason about partial views, both of requirements and of architecture. The *Define Goal* activity (see section 4) allows identifying the different concerns of the software and the crosscutting between them. These concerns are candidate to be classified as aspects, in the PRISMA specification, and realize them through aspects integrated into components and/or connectors.

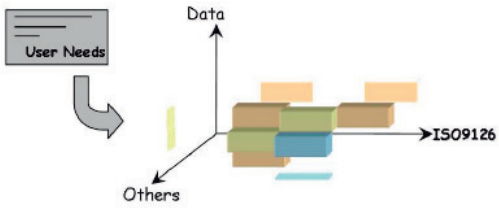


Figure 0 Unfolding a Software Specification

3. ISO/IEC 9126: Selecting and Identifying Concerns

Quality criteria, used for the software products assessment, are highly related to the requirements specified in their SRS. This means a practical binding with the global organization of the SRS in order to facilitate the subsequent evaluation of the software quality. In this sense, the ISO/IEC 9126 standard is an important reference as software quality model by defining a set of features that can be required of quality software. This reason makes the ISO/IEC 9126 especially suitable as taxonomy for concerns. It provides an initial framework to elicit and organize goals and requirements. In this way, as the informal software needs are elicited, they can be analyzed, broken down and organized. This allows managing the specification crosscutting, by reducing or removing other drawbacks such as: redundancy, inconsistencies, etc. At the same time, weaving relationships can be defined in order to re-establish a tangled representation whenever it is needed.

ISO/IEC 9126 determines three software quality aspects: process quality, product quality and product in use quality. The main aim is just the latter, i.e., to notice the product quality through the effect that its use causes. The quality in use depends on or is influenced by the internal and external characteristics of the software product. These characteristics are affected by the software construction process. With regard to the specific requirements, described in the SRS, we are interested in the characteristics defined for the software product quality and for the quality in use. These characteristics are listed in Table 1.

We have to notice that from the software requirements perspective this taxonomy goes beyond the traditional classification of functional and non-functional requirements which is not a meaningful contribution to the requirements organization. In fact, most of the typical functional requirements can be set with the *suitability* sub-characteristic. On the other hand, the software product capacity to satisfy the standards, conventions or regulations are broken down as sub-characteristics of type

"compliance" below each quality software product sub-characteristic. Although the ISO/IEC 9126 provides us with a wide set of concerns, this set can be extended if needed. In this case, several alternatives arise when we apply our proposal:

a) Considering a new dimension for organizing goals/requirements, additional to the taxonomy proposed by ISO/IEC 9126. This option could be of interest whenever the additional characteristics are as relevant as those already considered and whether the crosscutting with them could be high.
b) Including a new characteristic/sub-characteristic for extending the taxonomy. This alternative would be recommended when the aspects which are dealt with are not as relevant as those considered and/or it is not expected that the crosscutting could be so high.
c) Dealing with this element as an attribute of the goal/requirement. This option is suggested when it is coped with aspects which are not so relevant (they are neither exactly goals nor requirements) or do not involve an important crosscutting. However, they are particularly interesting to group and present goals/requirements.

Table 1 Quality Characteristics of the ISO/IEC 9126

Quality Type	Characteristic	Sub- Characteristic
Software Product Quality	functionality	suitability
		accuracy
		nteroperability
		security
		compliance
	reliability	maturity
		ault tolerance
		ecoverability
		compliance
	usability	understandability
		earneability
		operability
		attractiveness
		compliance
	efficiency	ime behaviour
		esourse utilisation
		compliance
	maintainability	analysability
		changeability
		stability
		esteability
		compliance
	portability	adaptability
		nstallability
		co-existence
		eplaceability
		compliance
Quality in use	effectiveness	
	productivity	
	safety	
	satisfaction	

The IEEE 830-1998 offers several criteria and guidelines to organize specific requirements. It recognizes that there is no an optimum organization to be applicable to every system. Between the mentioned organization criteria are: operation system mode, user type, problem entities, system services, stimulus, answer and/or functions hierarchies. Therefore, as it is recommended by the *c)* alternative, those elements can be dealt as attributes of goals/requirements instead of extending the taxonomy. In this way, it is possible to offer a view related to the joins based on these elements, although they are not elements of the initial taxonomy.

For instance, we could deal with the section "logic requirement of the database" (included in the IEEE 830-1998) as an aditional dimension (called *data*). Figure 2 illustrates the framework extension with new dimensions.

4. Goals Model and *Concerns* of the system

Both identification and specification of the different *concerns*, and their possible *crosscutting*, are addressed, in this work, with the definition of the tasks described in Figure 3, which integrate the *activity 1*. The output of this activity is the Goals Model, which was previously described in [15]. Its conception was influenced by the NFR Framework [3] and the KAOS proposal [4], although it integrates both functional and non-functional requirements. Furthermore, it plays an important role for the aspect identification as we previously stated in [17]. The Goals Model allows one to specify not only *goals*, *requirements* and *operationalizations* but also the *crosscutting* which can appear. For the Goals Model construction, the ISO/IEC 9126 quality model, described above, is used as an instantiable framework, providing the analyst with an initial description of the set of *concerns* of the system.

4.1. Defining the Goals Model

Figure 3 establishes the workflow for the Goals Model elaboration along with the input artifacts needed for its realization. Although Figure 3 shows only a sequential flow to apply the tasks, in practice, its application is iterative. Involved tasks, in the Goals Model, are run with every new identified or selected goal.

The first task deals with the *Identification/Selection of Goals*. As we can observe in Figure 3 the standard ISO/IEC 9126 is an input for this task. This model provides an initial view of the *concerns* that could be meaningful for the system. In this way, the analyst can iteratively select what he/she considers proper and initiate its specification and refinement. As we elicit the requirements, it could be convenient to incorporate new *concerns* to properly include additional goals/requirements.

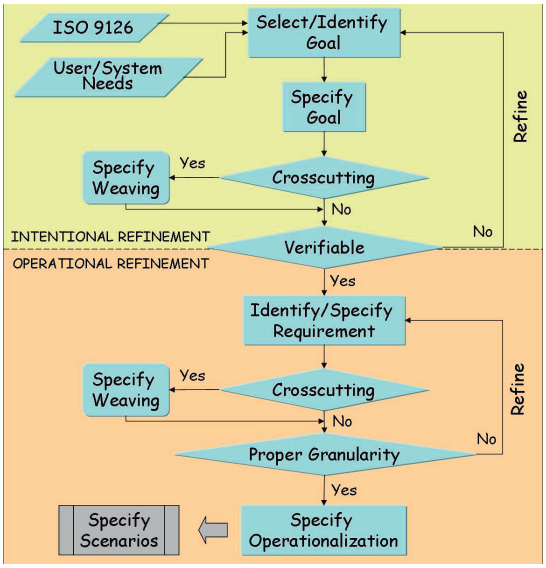


Figure 3 Workflow to specify goals and requirements

User/System needs are other input for this task. They provide us with information to identify new goals/requirements. Every goal, established by means of this process, is aligned with the specification of *concerns* determined by the ISO/IEC 9126 model, and acts as a node for the graph definition. This will provide us with a twofold advantage; on one hand to facilitate the understanding of the specification, and, on the other hand, to drive the elicitation and analysis process.

In the task *Specify Goal*, the attributes, that constitute it, are not only established, such as its name, priority, etc., but also composition relationships (AND/OR/XOR) among goals ([15] offers a full description of the goals model).

When crosscutting of goals/requirements is identified, task *Specify Weaving* is performed in order to establish weaving relationships. These relationships can be stereotyped according to the traditional AOSD weaving mechanisms, i.e., *before* and *after*. This allows us to express how a piece of goal/requirement specification (from the aspect point of view) is incorporated inside some goal/requirement specification. Other more specific weaving relationships could be used (like in [19]), but we suggest to do this refinement in the specific domain context of the system.

This refinement process of goals goes on until the goal is assignable to a system agent. At this moment, we change from an intentional refinement to an operational refinement, and consequently to the specification of a requirement. Another difference between a goal and a requirement stems from the latter ones have to be verifiable. In this way, *Identify/Specify Requirements* follows a similar process to that observed for goals, through the definition of attributes and composition relationships. Additionally, weaving relationships can also be established.

After the requirement specification, the next task is *Specify Operationalization*, that is, the definition of the agent or set of agents that collaborate in its realization. During the Goals Model construction, the operationalization is only a description of the proposed solution for the realization of a requirement, working this description as an input for the ATRIUM activity *Define Scenarios*. The latter cope with the whole definition of the solution through the relevant scenario specification. It is introduced in the Goals Model in order to allow us to describe the relationships between that solution and the already defined requirements in the Model. In this way, we can denote how a solution can contribute to positively realize a requirement and negatively to others. Thanks to these relationships, we achieve a more exhaustive analysis of the set of possible solutions.

5. Case Study

This section illustrates how we have applied our proposal in the context of a real system. This work was developed thanks to the collaboration with a group involved in the European Project Environmental Friendly and cost-effective Technology for Coating Removal (EFTCoR) [5]. The main scenario of this project is the hull maintenance operations. Mainly, it addresses operations of coating removal, washing and re-painting of hull of ships by using a family of robots, that either performs different operations or the same operation but in a different way.

These maintenance operations have a high impact both economical and environmental. The former is related to the time that the ship must go into the dry dock and to the costs derived of its maintenance. The later is due to the generated residues along the operations. Furthermore, these processes are very hazardous for operators.

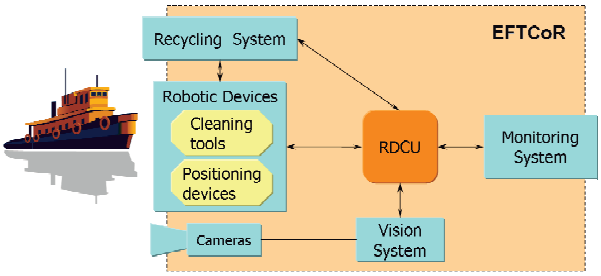


Figure 4 Teleoperation Robotic System for Hull Maintenance Operations



The identified robotic teleoperation platform [5] is integrated by the next subsystems (illustrated in Figure 4):

a) *Monitoring System*: encompasses the functionality concerning to the informational and managerial needs related to ship maintenance operation that is going to be accomplished.
b) *Vision System*: allows the hull inspection of the working areas and provides information for automatically moving the robotic devices along the hull.
c) *Recycling System*: retrieves the residues from the working areas and recycles them.
d) *Robotic Devices Control Unit*: interacts with the others robotic devices with the aim of getting the needed information to control the different devices (positioning systems and cleaning tools) to be used in the maintenance tasks. It is accomplished according to the commands introduced by the operator.

Our case study focuses on the Robotic Devices Control Unit (RDCU, Figure 4). Its architectural definition is highly relevant because the fact that several constraints have to be satisfied in order to allow a dynamic behavior of the system. This dynamism allows the EFTCoR to replace, at run time, each cleaning tools and positioning devices. Either change or operation has to be secure, providing a means to stop it if any damage can be produced to the equipment, the environment or the operator. Moreover, every operation has to be scheduled to accomplish hard deadlines.

5.1. Specifying the Goals Model

Next, some sentences about the established needs for RDCU, extracted from [5], are presented:

Positioning systems and tool can work simultaneously. The RDCU is responsible of co-ordinating their actions according to their operational states, the mission parameters, and the current state of the environment.
(1) The operational commands for positioning systems and activating the tool can be easily and efficiently issued to the system. *(2)The system should react to such commands efficiently. In some cases, the execution time of the commands should be smaller than a given deadline.*
(1)The possibility of using different coating removal technologies (blasting, water pressure). Though the chosen technology for the EFTCoR project is blasting, the RDCU should be open to incorporate cleaning tools based on other technologies. *(2) The possibility of using the system for different maintenance tasks, including at least the fresh water washing before blasting and the painting after blasting.* *(3) The possibility of using different positioning systems and different combinations of primary and secondary positioning systems.* *(4) The possibility of using different tools for the same or different processes (already considered in the first point).*

From the previous statements we can observe how the crosscutting appears in the specification, as for instance the goals related to *Efficiency* and *Adaptability*. Both goals are applied to other goals as *ControlPositioning* or *ControlTools*. When the task *Identification/Selection of Goals* was applied by using the previous sentences as input, it generated as a result the next goals list that

appears on Table 2 (more details about the applicable attributes can be found in [15])

The graph on Figure 4 shows (part of) the Goals Model where the refinement of goals from Table 2 is reviewed. In this way, we observe how *Portability*, *Functionality* and *Efficiency* are some of the selected characteristics to become concerns for the EFTCoR system. Furthermore, this figure shows us some of the relationships of refinement that were established. For instance, the AND relationship for the goals *AdaptabilityWorking-Environment* and *AdaptabilityHullMaintenaceOperation* that was introduced to satisfy *Adaptability*. On the other hand, a weaving relationship has been established between *AdaptabilityWorkingEnvironment* and *Control-Possitioning*.

Table 2 (Partial) Description of Goals for EFTCoR

GOAL	DESCRIPTION
Functionality	The system has to provide functions which meet stated and implied needs
Suitability	The system has to provide an appropriate set of functions for specified tasks and user objectives
CoordinateDevices	The RDCU has to coordinate robotic devices according to the current mission procedure and global system state
ControlPossitioning	The RDCU has to control the positioning devices
ControlTools	The RDCU has to control the tools attached to positioning devices blasting head, painting tool, etc)
Efficiency	The system should use efficiently their esources
TimeBehaviour	The system should respond with appropriate speed
PerformanceOperation	The system should respond with appropriate speed to the operation commands
Portability	The system should be able to be ransferred from one environment to another
Replaceability	The system should be able to use different software product for the same purpose
Adaptability	The systems has to be adaptable for different specified environments
AdaptabilityWorkingEnvironments	The system operation has to be adaptable to different working environments
AdaptabilityHullMaintenanceOperation	The system operation has to be adaptable to different hull maintenance operations

6. Related works

There are many works related to the management of aspects at different abstraction levels, at implementation as well as design level. At the requirements level, several proposals have focused on how to identify and specify concerns of the system. They also focus on how to determine which concerns will be realized as aspect, in other derived artifacts, in such a way that the closure property [6] is satisfied.

Grundy [9] has defined a proposal called Aspect-Oriented Component Engineering (AOCE), in order to define and develop software components from requirements until design, implementation and deployment. In this proposal, components are integrated by aspects, but there is no explicit specification about how these aspects are a realization of the aspects at the requirements level.

Rashid et al [20] have proposed a model for requirements engineering that identifies the concerns at an early stage of development, along with the requirements. This model allows relating and establishing the impact of these early aspects on later stages of the development. Nevertheless, this proposal does not allow recreate the specification to offer a better comprehension for the validation end user.

Brito et al [2] proposal is the closest one to our approach. It introduces a goals model, the NFR Framework [3] concretely, to specify non-functional requirements along with another technique, such as use cases or viewpoints, to specify functional requirements. In this way it extends the framework with a new type of relationship called *required concerns* in order to determine a potential crosscutting with other requirements. However, our proposal provides the analyst with a unique artefact to specify both kinds of requirements. It facilitates both its use and comprehensibility.

On the other hand, no stated proposals provide the analyst with an initial framework which helps him/her to identify the concerns of the system. Finally, another significant difference is related to the uniform management of goals/requirements, with regard to the applicable categories for its classification. Nevertheless, other proposals explicitly focus on the distinction between functional and non-functional, by employing different techniques to describe both kinds of goals/requirements. This entails a high cost in terms of legibility and maintainability.

Additionally, most of the Aspect Oriented requirements proposals, such as [19, 14], apply separation of concerns techniques after identifying the crosscutting in the specification. In this way, they offer a solution to improve the original specification. In our proposal, thanks to the use of a Goal Oriented approach, that allows us to begin the software requirement specification at a level of concerns, we can deal with the goals and requirements relationships detecting and managing the crosscutting among them by using weaving relationships.

7. Conclusions and future works

In this work, we have presented a proposal to elicit and analyse requirements that integrates Goal Oriented and Aspects Oriented approaches. Owing to the Goal Oriented approach, an explicit traceability is preserved from goals towards requirements, and from the latter to the operationalizations that realize the software components. On the other hand, the Aspect Oriented approach allows a

smart and effective management of the crosscutting that can appear in the SRS when goals and requirements are tangled. By defining weaving relationships between goals/requirements elements an optimal representation is achieved, that can allow to recreate the original representation whether it is needed. A workflow has been defined that integrates both approaches, detailing a set of tasks. It provides a guide for the elaboration and organization of the requirements.

Another advantage that offers our proposal is the use of the ISO/IEC 9126 as a starting point to establish the possible concerns. Additionally, it is possible to tailor, in terms of content, the SRS to the IEEE 830-1998 but with meaningful advantages for elaboration and organization of the requirement specification.

Several challenges are to be faced. One of them is related to the development of a tool that provides our proposal with a proper support. The flexibility for the requirements representation, regarding several criteria, is one of main factor in this development. With this aim, we are exploiting the structure of characteristics as well as using attributes which are defined for goals and requirements.

Another point of interest, for future works, is related to the weaving relationships. Concretely, we are concerned with its suitability to the set of offered relationships along with their semantic. According to this topic, the emerging issues from the case study will facilitate its verification and validation. Specially when, following the ATRIUM stated process, the established weaving relationships and the identified concerns and requirements will have traceability to aspects in architectural artefacts.

References

[1] Aspect-Oriented Software Development, http://www.aosd.net

[2] I. Brito, A. Moreira, "Integrating the NFR framework in a RE model", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, March 22, 2004 - Lancaster, UK in conjunction with 3rd Aspect-Oriented Software Development Conference.

[3] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, Non-Functional Requirements in Software Engineering, Kluwer Academic Publishing, 2000.

[4] A. Dardenne, A. van Lamsweerde, and S. Fickas: "Goal-directed Requirements Acquisition". Science of Computer Programming, 20:3-50, 1993.

[5] EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. European Project within the 5th Framework Program (GROWTH G3RD-CT-00794), 2003.

[6] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, H. Ossher, "Discussing aspects of AOP", Communications of the ACM, Vol. 44 , Issue 10, pp. 29 32, October, 2001.

[7] T. Elrad, R. E. Filman and A. Bader, "Aspect-oriented programming: Introduction", Communications of the ACM, Vol. 44 , Issue 10, pp. 29 32, October, 2001.

[8] C. Fernández, J.A. Pastor, P. Sánchez, B. Álvarez, A. Iborra: "Co-operative Robots for Hull Blasting in European Shiprepair Industry. Robotics and Automation Magazine (RAM)", special issue on Industrial Robotics Applications & Industry-Academia Cooperation in Europe. New Trends and Perspectives. (To appear in September 2004)

[9] J. Grundy, "Aspect-Oriented Requirements Engineering for Component-Based Software Systems", In Proceedings of the IEEE International Symposium on Requirements Engineering, June 07 - 11, 1999, Limerick, Ireland.

[10] ISO/IEC Standard 9126-1 Software Engineering- Product Quality-Part1: Quality Model, ISO Copyright Office, Geneva, June 2001

[11] IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications, In Volume 4: Resource and Technique Standards, The Institute of Electrical and Electronics Engineers, Inc. IEEE Software Engineering Standards Collection.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier and J. Irwin, "Aspect-Oriented Programming". In the Proceedings of the European Conference on Object-Oriented Programming, Finland. Springer-Verlag LNCS 1241. pp. 220-242, June 1997.

[13] A. van Lamsweerde "Goal-Oriented Requirements Engineering: A Guided Tour", Invited Paper for 5th IEEE International Symposium on Requirements Engineering, Toronto, August, 2001, pp. 249-263.

[14] A. M. D. Moreira, J Araújo, I. Sofia Brito, "Crosscutting quality attributes for requirements engineering". In Proceedings of the 14th international conference on Software Engineering and Knowledge Engineering, pp. 167-174, July 15-19, 2002, Ischia, Italy.

[15] E. Navarro, I. Ramos and J. Pérez: "Goals Model-Driving Software Architecture", 2nd International Conference on Software Engineering Research, Management and Applications, May 5-8, 2004, Los Angeles, CA, USA.

[16] E. Navarro, I. Ramos and J. Pérez: "Software Requirements for Architectured Systems". Proc. 11th IEEE International Conference on Requirements Engineering, pp. 365-366, September 8-12, 2003, Monterey, CA, USA.

[17] E. Navarro and I. Ramos, "Requirements and Architecture: a marriage for Quality Assurance", VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante, 12-14 Noviembre 2003

[18] J. Pérez, I. Ramos, J. Jaén, P. Letelier and E. Navarro, "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", In Proceedings of the 3rd IEEE International Conference on Quality Software, Dallas, Texas, USA, pp. 59-66, November 6 - 7, 2003.

[19] A. Rashid, A. Moreira, J. Araújo, "Modularisation and composition of aspectual requirements", In the Proceedings of the 2nd international conference on Aspect-oriented software development, pp. 11-20, March 17 - 21, 2003, Boston, Massachusetts.

[20] A. Rashid, P. Sawer, A. Moreira and J. Araújo, "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", In the Proceedings of the International Conference on Requirements Engineering, pp. 199-202, September 9-13, 2002, Essen, Germany.

[21] J. Suzuki and Y. Yamamoto, "Extending UML with Aspects: Aspect Support in the Design Phase", AOP Workshop at ECOOP'99, Lisbon, Portugal, 1999.

UML Visualization for an Aspect and Goal-Oriented Approach

Elena Navarro
Department of Computer Science, UCLM
Avda. España S/N
Albacete, Spain
+34 967 59 92 00 ext. 2461
enavarro@info-ab.uclm.es

Patricio Letelier
Department of Information Systems and Computation, UPV
Camino de Vera s/n
Valencia, Spain
+34 96 387 7007 ext. 73589
letelier@dsic.upv.es

Isidro Ramos
Department of Information Systems and Computation, UPV
Camino de Vera s/n
Valencia, Spain
+ 34 96 387 7350
iramos@dsic.upv.es

ABSTRACT

The Goal-Oriented requirement engineering approach offers important advantages for a deeper study of software requirements. Some of them are supported for reasoning about design alternatives and traceability between requirements and software architecture. However, in complex systems, requirements specifications suffer from crosscutting, which affects elaboration, readability and maintainability of the specification, even when using a Goal-Oriented approach. Separation of concerns, included in Aspect-Oriented Requirement Engineering provides an elegant and effective solution to cope with this problem. In this work we present a model for requirement specification which integrates Goal-Oriented and Aspect-Oriented approaches. This model is included in ATRIUM, a methodology for concurrent definition of requirements and software architecture. Using a UML profile we give graphical notation to our model allowing its support in most CASE tools based on UML.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: *Requirements Specification – Elicitation methods, Languages and Methodologies.*

General Terms

Algorithms, Documentation, Design, Reliability, Standardization, Languages, Verification.

Keywords

Aspect Oriented, Goal Oriented, Software Requirements, Software Architecture.

1. INTRODUCTION

The requirements specification involves a number of challenges related to quality characteristics that must be achieved, like those described in IEEE 830-1998 [10] standard as: correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, and traceable. The Goal-Oriented approach [12] has aroused interest in researchers because of its facilities for elaboration and deeper analysis of requirements specification. In this approach, detailed requirements are obtained by stepwise refinement starting from general system goals (or concerns). This refinement continues until requirements are assigned to system agents which are able to operationalize them. Thus, the Goal-Oriented paradigm has two advantages that make it especially suitable to guide the selection among several architectural design alternatives:

- Its ability to specify and manage positive and negative interactions between goals [6] allows the analyst to reason about design alternatives.
- Its capability to trace low-level details back to high-level concerns [7] is very appropriate to bridge the gap between architectural models and requirements.

However, like in other approaches for requirements specification, when dealing with complex and/or large systems, crosscutting of elements usually appears in the specification. This crosscutting manifests itself by affecting negatively readability and maintainability of the specification. The Aspect-Oriented Requirement Engineering (AORE) [20] identifies and manages the crosscutting in an elegant and effective way, based on separation of concerns.

ATRIUM [17] (Architecture generated from RequIrements applying a Unified Methodology) is a methodology that emphasizes the concurrent definition of requirements and software architecture. In the ATRIUM context, the Goals Model plays an essential role conducting the software architecture generation and validation process. This Goals Model integrates Goal-Oriented and Aspect-Oriented approaches, offering the advantages of each of them.

On the other hand, one of the main concerns in any modern modelling approach is to include a notation that allows a visual representation of the produced specifications. UML seems to be the appropriated candidate to be used as a language for software modelling. Thanks to its extension mechanisms around the concept of UML profile, UML can be extended and adjusted to the particular needs of our Goals Model. Furthermore, most CASE tools provide support for UML and allow us, in different

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
5th Aspect-Oriented Modeling Workshop'04, October 11, 2004, Lisbon, Portugal.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

Navarro, E., Letelier, P., Ramos, I. UML Visualization for an aspect and goal oriented approach. Proc. of the 5th Aspect-Oriented Modeling Workshop (AOM'04) at UML 2004 Conference. 2004.

levels, to work with these kinds of extensions based on UML profiles.

In Requirements Engineering, there are some works defining notations based on UML in the Goal-Oriented [9] approach as well as in the Aspect-Oriented one [4]. However, none of them integrates both perspectives in the way required by our Goal Model.

The aim of this work is to present the Goal Model of ATRIUM and the definition of a corresponding UML profile. This work is structured as follow: next section is giving a brief review of ATRIUM, its intention and activities. Section 3 shows how the profile has been described for our proposal. Eventually, section 4 and 5 describe the related works and the reached conclusions.

2. ATRIUM: REQUIREMENTS AND SOFTWARE ARCHITECTURES

ATRIUM is a methodology oriented to the concurrent definition of Software Architecture (SA) and Requirements. In ATRIUM, decisions at architectural level are made to satisfy specific software requirements. With this aim, ATRIUM provides the analyst with guidance, along an iterative process, from an initial set of user/system needs until the instantiation of the architecture, specified by means of PRISMA model [19]. PRISMA is an architecture description language that allows us to define dynamic architectures.

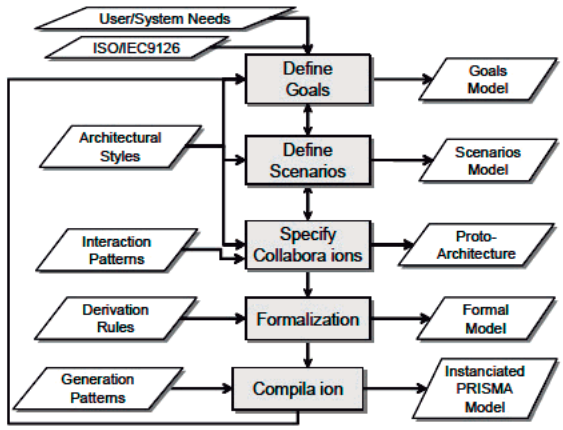


Figure 1 ATRIUM: activities and artefacts

ATRIUM entails five activities (Figure 1) to be iterated on in order to define and refine the different artefacts and allow the analyst to reason about partial views, both of requirements and of architecture. The *Define Goal* activity allows one to identify the different concerns of the software as well as the crosscutting between them. The main idea is to determine which concerns are candidate to be classified as aspects, in the PRISMA specification, and realize them through aspects integrated into components and/or connectors.

The Goals Model, one of the artefacts generated by applying ATRIUM and previously presented at [16], was inspired by the NFR Framework [6] and KAOS [7]. Goals, requirements, contribution, etc, are also elements in its construction. Nevertheless, the main difference, that this proposal exhibits, is that only one model is introduced for functional and non-functional requirements definition. It is due to both types of requirements are highly relevant in the architectural definition.

Finally, our proposal introduces concepts from the Aspect Oriented Software Development (AOSD [3]) approach. Terms as *weaving relationships*, *crosscutting*, etc, have been brought in our approach, as we already stated in a previous work [18]. In our approach, the Goals Model allows us to identify and manage the involved concerns, in such a way that every identified goal, along the process, is considered initially as a concern.

2.1 Building blocks

The Goals Model provides a number of abstractions in terms of which constraints on the software system have to be defined. A key element introduced in its construction is a *goal*. It is defined as an objective that the system-to-be should achieve [13], i.e., a constraint or obligation that the system should meet. In its definition it is characterized as *Functional* or *Non-Functional*, according to the type of need or expectation it refers to:

- *Functional goals or requirements* describe services that the software provides, i.e., the transformations the system performs on the inputs.
- *Non-Functional goals or requirements* describe conditions or constraints that the software must satisfy; they refer to how the services are provided, for instance, in terms of performance, adaptation, security, etc. We are highlighting them because they are especially meaningful in terms of the architecture of the end system.

Additionally, other characteristics have to be stated when a goal is defined. For instance, each goal has to be classified according to its *priority*, from *very high* to *very low*, for the system-to-be. This classification helps the analyst to focus on the important issues. These priorities can arise from several factors: organizational ones when they are critical to the success of the development, constraints on the development resources, etc.

Moreover, a set of *preconditions* and *postconditions* should be identified. Preconditions establish which situations must hold before some operation is performed. Postconditions define the conditions that have to be satisfied after some operation is performed. Their evaluations help us to determine the best design alternatives among those that satisfy the postconditions for the established goals. For their description a variant of dynamic logic [14], which includes deontic operators for expressing permission and obligation, is used but to go into more details on this topic is out of the scope of this paper.

Similarly to goals, another used element in the Goals Model construction is known as *requirement*. They also specify a need or constraint on the end system, although its main difference regarding goals is its capability to be assigned to and realized by a set of agents. Additionally, also due to its capability it can be verified at the end system. Its textual notation is similar to that defined for goals. However, in this case, both postconditions and preconditions should be defined for each requirement.

Aside from goals and requirements, another building block for the Goals Model is the *operationalization*. When an analyst has refined the initial set of goals, he/she must offer a set of solutions that allow the system to achieve the established goals. An operationalization is a solution that provides the target system with architectural design choices which meet the users' needs and

expectations. They are called *operationalizations* because they describe the operation of the system, i.e., the system behaviour, to meet functional and non-functional requirements.

It can be noticed that the alternative solutions to satisfy a given requirement are not described on each operationalization. On the contrary, it is in the Scenarios Model where these solutions are expressed. However, operationalizations are introduced in the Goals Model to conceptually represent each solution so that relationships among the different alternatives can be established within the Goals Model. Operationalizations establish a coupling between the Scenarios Model and the Goals Model, establishing the traceability between operationalizations and a specific view of the Scenarios Model.

Additionally, we want to notice that operationalizations are not functionally or non-functionally characterized like goals and requirements. This is because the same solution can be associated to different goals, both functional and non-functional.

2.2 Relationships for the Refinement Process

The stated building blocks, goals, requirements and operationalizations, are inter-related by means of a set of *relationships*. They are in charge of gluing the different elements to complete the model and enhance its cohesion. Moreover, their relevance is not only restricted to this gluing but also they allow the analyst to introduce the rationale for the system design. The decomposition of goals or how an operationalization positively or negatively contributes to a goal can be defined via relationships.

They are applied via a stepwise refinement process which takes an informal set of user/system needs, usually stated in natural language, as well as the framework for an initial selection of concerns provided by the ISO/IEC 9126 [11]. Both act as inputs to begin with the model definition, and using some of the proposals to identify goals [2]. In such a way, every refinement step generates new goals/requirements and/or operationalizations in the model. Therefore, the analyst has to deal with a reduced set of building blocks at each step.

There are two types of refinements that can be applied: intentional and operational. The former describes how a goal can be reduced into a set of subgoals/requirements via *AND/OR/XOR* relationships. The latter depicts how a set of solutions address a requirement by means of *AND OPERATIONALIZE* / *OR OPERATIONALIZE* / *XOR OPERATIONALIZE* relationships. Both building blocks and relationships are structured as an acyclic graph, where the refinement is achieved along the structure, from the higher to the lower level, by applying intentional and operational refinements.

Every goal, which is too coarse-grained, is refined in a set of subgoals which are a decomposition of the original one. An *AND* relationship between a goal *GoalX* and a set of sub-goals G_1, …, G_N or requirements R_1, …, R_N is established if the whole set of sub-goals and/or requirements has to be satisfied in order to satisfy *GoalX*. An OR relationship is established if *GoalX* is satisfied if at least a sub-goal or requirement is satisfied. Finally, a XOR relationship is introduced if *GoalX* is satisfied when only a sub-goal or a requirement from this set can be satisfied.

An operational refinement deals with requirements and operationalizations. The alternative solutions for each goal are established by means of this decomposition. There can be a large number of valid operationalization methods that are applicable to a requirement. In such a case, it is up to the analyst to examine the impact of such methods on other requirements and decide on what and how many operationalizing methods must be applied via *AND OPERATIONALIZE/ OR OPERATIONALIZE/ XOR OPERATIONALIZE* relationships. An *AND OPERATIONALIZE* relationship relates the set of mandatory solutions for a requirement. On the other hand, whenever several alternative solutions can be provided for a requirement, the analyst can introduce them in terms of the *OR OPERATIONALIZE* relationship. Finally, whenever several alternative solutions can be provided for a requirement, but only one can be selected for the end system, the analyst can introduce them by using *XOR OPERATIONALIZE* relationship. OPERATIONALIZE relationships do not only relate operationalizations to requirements but also to other solutions. It is used to refine the operationalizations down to other simpler ones, i.e., to describe how a solution can be expressed in terms of a set of simpler solutions.

A set of symbols [++|+|#|-|--] are used to characterize the way an operationalization contributes to achieve a requirement. Symbols ++ and + describe a strong positive and positive contribution, i.e., it provides a sufficient or partially sufficient solution, respectively, to satisfy the related requirement. On the other hand, symbols -- and - describe a strong negative or negative contribution, i.e., the operationalization prevents or partially prevents, respectively, the satisfaction of the related requirement. The # symbol is introduced to specify operationalizations whose impact (positive or negative) is neutral at the moment. This is the default value.

The intentional refinement is iteratively applied to the set of goals ending up when every sub-goal can be operationalized, i.e., when a requirement can be defined. Similarly, the operational refinement is iteratively applied to operationalizations until the corresponding scenarios are simple enough.

Another type of relationship that can be introduced is *conflict*. It can be set up among two goals/requirements if an incompatibility appears between them, in other words, whenever the satisfaction of a goal/requirement prevents the satisfaction of another goal/requirement.

Finally, other relationship that can appear between goals/requirements is called *weaving*. When a goal/requirement crosscut other goals/requirements, a *weaving* relationship is established. These relationships can be charecterized according to some of the traditional AOSD weaving mechanisms, like *before* and *after*. This allows us to express how a piece of goal/requirement specification (from the aspect point of view) is incorporated inside some other goal/requirement specification. Other more specific weaving relationships could be used (like in [19]), but we suggest to do this refinement in the specific domain context of the system.

3. DEFINING A UML PROFILE

Along the process of definition of the ATRIUM profile, we were faced with several challenges. One of them was related to the satisfaction of the requirements [1] that Aldawud et al stated for defining a UML profile for AOSD:

(1) *The Profile shall enable specifying, visualizing, and documenting the artifacts of software systems based on*

Aspect-Orientation. This requirement has been satisfied by means of the stereotypes and their visual representation as it is described below.

(2) *The Profile shall be supported by UML (avoid "Heavy-weight" extension mechanisms), this allows a smooth integrating of existing CASE tools that support UML*. This requirement is also satisfied due to our UML profile has been defined according to the established construction rules in the UML specification [22].

(3) *The Profile shall support the modular representation of crosscutting concern.* The separation of concerns provided by the goal oriented approach, along with the defined weaving relationship, allows us to identify and manage crosscutting in an early stage.

(4) *The Profile shall not impose any behavioural implementation for AOSD, however it shall provide a complete set of model elements (or Stereotypes) that enable representing the semantics of the system based on Aspect-Orientation*. No constraint has been defined about the implementation, only a proper semantic related to the way we use the ATRIUM elements at the requirements stage.

With the aim of describing this UML support for the ATRIUM elements, two tasks have been carried out: the Metamodel description of the Goals Model (section 3.1) and the UML profile associtated to the metamodel (section 3.2 and 3.3).

3.1 Metamodel Description

The metamodel, shown in Figure 2, defines the abstract syntax for specifying Goals Model in ATRIUM. We use the prefix "A-" (from **A**TRIUM) to name metaclasses. One of the model elements is *AGoal* which allow us to describe every concern of the Goals Model, as we described above. *AGoal* has a meta-attribute called *type* which describe the type of the goal, i.e., functional or non-functional. By means of the generalization, *ARequirement* inherits this meta-attribute, i.e., it can be also typed as functional and non-functional.

Several relationships are described in the model. One of them is *AGoal/RequirementGroup* which is used for representing the refinement of a *AGoal* element into a set of goals/requirements. As shown, the meta-attribute *joinType* is defined that allow us to describe the type of refinement (AND, OR or XOR) by means of the enumeration *JoinType*. In addition, *AWeaving* relation is introduced with a meta-attribute called *matchPoinType* which allow us to specify the way the weaving is applied. Its type is *MatchPointType* that has been defined as an *Enumeration*. Eventually, a *conflict* relationship is established to describe the relation between conflicting goals/requirements. Every one of the previous relationships is inherited by *ARequirement*, that is, they can be used for it with the same meaning.

AOperationalizationGroup allow us to represent how an element *ARequirement* has associated a set of operationalizations, and the way each element *AOperationalization* contributes to its realization. The allowed contributions are described with the enumeration *ContributionType.*

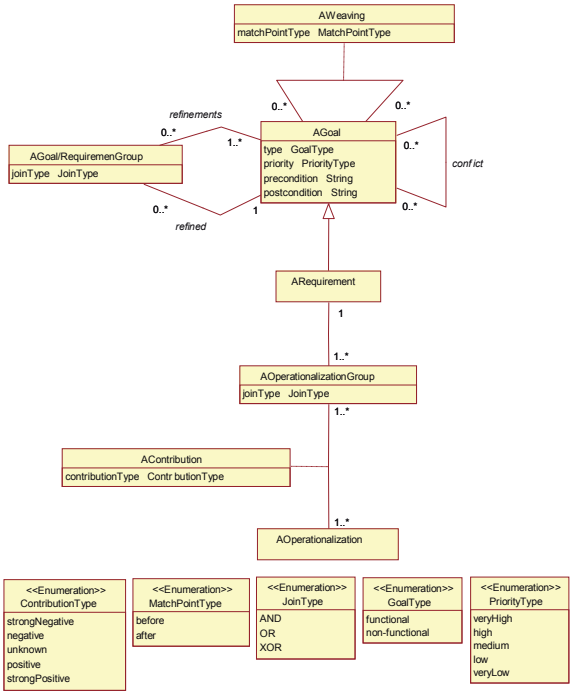


Figure 2 Metamodel of the Goals Model

3.2 UML Profile

This profile defines an extension to the reference UML 2.0 metamodel with the purpose of tailoring it to our Goals Model but keeping its semantics. Metaclasses of the UML metamodel are extended by means of a mechanism called *extension* that is represented with an arrow with filled end, as can be observed in Figure 3. For instance, *AContribution* is defined as a stereotype which extends the UML metaclass *Generalization*. Additionally, those specific features of the ATRIUM elements are defined by means of meta-attributes (tagged-values in 1.5) of the stereotypes to be described. That is the case of the meta-attribute *joinType* in *AContribution*. In a similar way, each element, which appears in the Goals Model Metamodel, is mapped to a stereotype and avoiding "Heavy-weight" extension mechanisms, as can be observed in the Figure 3.

Additionally, a set of well-formed rules, in the context of a profile, are defined to introduce the specific needed semantic. Several structural constraints can be easily extracted from the Metamodel in Figure 2. For instance, a *conflict* can be described among two Goals/Requirements, and similarly for the *weaving* relationship. However, the more relevant rules are those related to the domain semantic. These rules express constraints such as: a *ARequirement* always has to be defined by refining one or more *AGoal*.

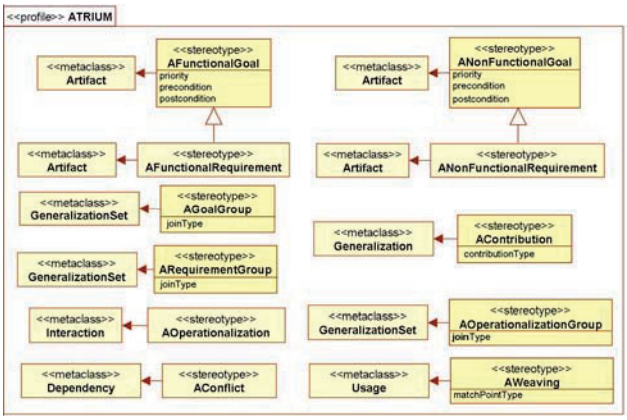


Figure 3 Goals Model Profile

3.3 Graphical Notation

This section outlines the graphic elements that may be shown in a Goals Model. Along with the visual notation for every described stereotype, a brief description is also provided. All of them are described in Table 1.

Table 1 Graphics nodes included in Goals Models

Node Type	Notation	Description
AFuntionalGoal		Indicates a software functional goal that the system must satisfy
ANonFuntionalGoal		Indicates a software non-functional goal that the system must satisfy
AGoalGroup	<<GoalGroup>>	Indicates a refinement of one goal in subgoals or requirements
AFunctional-Requirement		Indicates a functional goal which is verifiable and assignable to an agent
ANonFunctional-Requirement		Indicates a non-functional goal which is verifiable and assignable to an agent
AOperationalization		Indicates a scenario describing how agents collaborate to support a requirement
AOperationalization-Group	<<Operationalization Group>>	Indicates a set of possible operationalizations for a requirement Each operationalization is modeled as a AContribution
AContribution		Indicates how one operationalization contributes to one requirement
AConflict	<<Conflict>>	
AWeaving	<<weaving>>	Indicates a weaving relationship between two goals or requirements

4. AN EXAMPLE

This section illustrates how we have applied our proposal in the context of the European Project Environmental Friendly and cost-effective Technology for Coating Removal (EFTCoR) [8]. The scenario of this project is the hull maintenance operations of ships. Mainly, it addresses operations of coating removal, washing and re-painting of hull of ships by using a family of robots, that either perform different operations or the same operation but in a different way. The identified robotic teleoperation platform is integrated by the next subsystems:

(1) *Monitoring System*: encompasses the functionality concerning to the informational and managerial needs related to ship maintenance operation that is going to be accomplished.

(2) *Vision System*: allows the hull inspection of the working areas and provides information for automatically moving the robotic devices along the hull.

(3) *Recycling System*: retrieves the residues from the working areas and recycles them.

(4) *Robotic Devices Control Unit*: interacts with the other robotic devices with the aim of getting the needed information to control the different devices (positioning systems and cleaning tools) to be used in the maintenance tasks. It is accomplished according to the commands introduced by the operator.

Our case study focuses on the Robotic Devices Control Unit. Its architectural definition is highly relevant due to the fact that several constraints have to be satisfied in order to allow a dynamic behaviour of the system. This dynamism allows the EFTCoR to replace, at run time, each cleaning tools and positioning devices. Either change or operation has to be secure, providing a mean to stop it if any damage can be produced to the equipment, the environment or the operator. Moreover, every operation has to be scheduled to accomplish hard deadlines.

The graph on Figure 4 shows (part of) the Goals Model where the refinement of goals is reviewed. In this way, we observe how *Portability*, *Functionality* and *Efficiency* are some of the selected characteristics to become concerns for the EFTCoR system. Furthermore, this figure shows us some of the relationships of refinement that were established. For instance, the AND relationship for the goals *AdaptabilityWorkingEnvironment* and *AdaptabilityHullMaintenaceOperation* that was introduced to satisfy *Adaptability*.

On the other hand, crosscutting also appears in the specification. For instance the goals related to *Efficiency* and *Adaptability*. Both goals are applied to other goals such as *ControlPositioning* or *ControlTools*. Furthermore, conflict relationship can be described in this model, when *Security* and *Performance* are demanded goals for the system.

5. RELATED WORKS

Most works on profiles for AOSD has focused their efforts on the design stage, such as [1, 21]. However, there are no many proposals for the requirements stage. One of them was presented by Araujo et al [4]. They have described an extension for UML that provide support to a previous approach [15]. This one establishes the way non-functional requirements constrain functional requirements. In this way, functional requirements are described as Uses Cases and non-functional requirements as stereotyped Use Cases. The main problem, which this proposal shows, arises from a metamodel in permanent evolution, i.e., whenever a new non-functional requirement has to be specified, a new stereotype, with Use Case as base class, has to be defined which is a heavier mechanism than our proposal.

More related to the Goals Model, and the way it is used for eliciting requirements, is the proposal presented by Heaven and Finkelstein [9]. They have defined a UML profile (based on UML

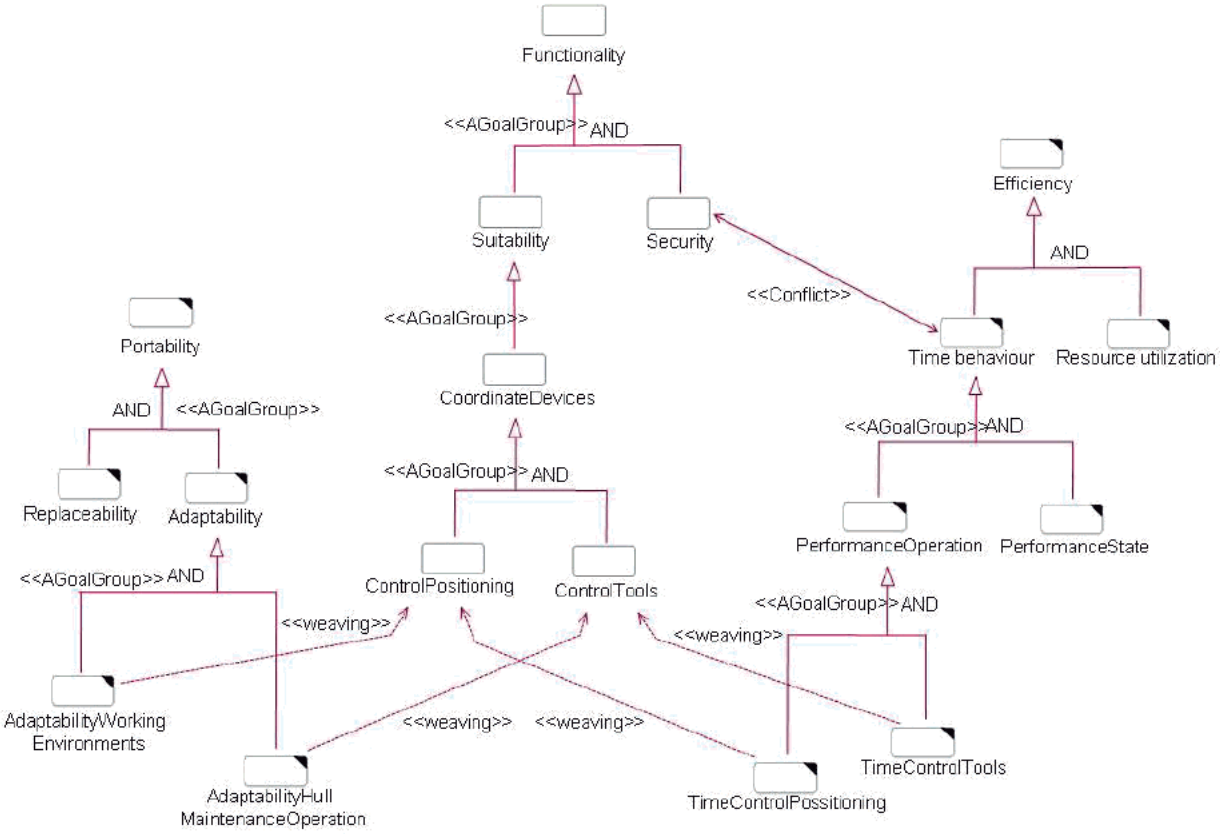


Figure 4 Partial view of the Goals Model for EFTCoR

1.4) for the Goals Model of KAOS. Several stereotypes are included that describe some common concepts with our proposal, for instance, *goals* and *AND/OR/XOR* refinement relationship. In this work, the metaclass Abstraction is used as base class for the stereotype <<reduces>>. This stereotype represents a refinement relationship between a goal and a subgoal. This alternative presents an important drawback when attempting to establish a characterization AND/OR/XOR for the whole refinement hierarchy, instead of establishing it for each subgoal refinement. In our approach, we use metaclass *GeneralizationSet* (incorporated in UML 2.0) for this purpose. It allows us to express the idea of hierarchy with the corresponding characterizations. Additionally, they do not provide the analyst with specific elements to describe the relation between functional and non-functional goals, as we have stated above, that is, the weaving relationship does not exist in their proposal. Additionally, they also introduce operationalizations and an associated model for its description. However, they do not use any interaction diagram even though they are defining the system behaviour. On the contrary, they define a set of new stereotypes to describe concepts such as *actions* or *events* instead of using the provided UML support by means of *activity or sequence diagrams*.

Another difference regards to dynamic logic has been selected as formalism instead of temporal logic that KAOS uses. This election is due to our previous experience with industrial case studies. The analyst comprehensibility and usability was greater when dynamic logic was introduced than temporal logic.

Brito and Moreira [5] have also introduced a Goal Oriented proposal, concretely, the NFR Framework [6]. Every identified concern is specified by using the best approach. In its case study Use Cases are used for functional requirements and SoftGoals Interdependency Graph (SGI) for non-functional requirement. Additionally, a template has to be fulfilled for every stated concern, where both the contributions and the required concerns are drawn. These provide information to detect the crosscutting concern whose specification is accomplished by using a table. This means that a diversity of notations is used in their approach. SGI does not play an important role along the process, only as a mean to describe non-functional requirements. This involves a greater effort in order to have a full comprehension of the system due to the specification is scattered over several artefacts.

The main advantage that our proposal offers is twofold. On one hand, only one single model is introduced for the integrated description of both functional and non-functional elements. On the other hand, our UML profile does not have to be modified to accommodate new non-functional requirements

6. CONCLUSIONS AND FUTURE WORKS

In this work we have presented the Goals Model included in ATRIUM. This Goal Model offers a relevant improvement to specification of requirements that takes the advantages of two prominent and modern approaches: Goal-Oriented and Aspect-Oriented Requirements Engineering. Furthermore, we have defined a UML profile to facilitate the practical use of our proposal in most current CASE tools. Using an example we have

illustrated the application of the Goal Model and the corresponding UML profile.

In our proposal the *aspect* concept does not explicitly appear as a constructor, as other works do. Instead, the candidate aspects implicitly arise on those goals/requirements with weaving relationships. This is due to the concept of aspect is specified in other models of the ATRIUM approach. For instance, when shallow components and connectors are identified and specified in the Scenarios Model, weaving relationships are taken into account for defining possible aspects. We have to consider that both architectural elements are defined by a gluing of aspects.

Several works are in progress related to the stated here and, mainly, to ATRIUM. They are related to the definition of the other involved models in ATRIUM and their corresponding UML visualization. We foresight the common modelling framework of UML will facilitate the management of traceability between involved models.

Also, a deep inspection about the associated semantic of weaving relationships remains as a current challenge. Until now, the traditional ones have been defined but those introduced by Rashid et al [20] can suggests new alternatives for our approach. In this sense, we think the equilibrium between the readability/simplicity of the specification and the versatility of the weaving relation should be achieved.

Another key topic is related to identification of interaction patterns for several crosscutting concerns in order to manage the possible interference among them. In the developed case studies, we observed that several concerns crosscut another one. The associated semantic of this composition and how the tradeoffs between them have to be faced must be solved in the next future.

7. ACKNOWLEDGMENTS

This work has been funded by the Spanish CICYT project DYNAMICA TIC2003-07776-C02-02.

8. REFERENCES

[1] Aldawud, O. Elrad, T. and Bader, A. UML profile for Aspect Oriented Software Development. In the *Aspect-Oriented UML Workshop*, Collocated to the Aspect Oriented Software Development Conference, (Boston, USA March 18, 2003).

[2] Anton, A. I. Goal-Based Requirements Analysis. In Proceedings of the *2nd International Conference on Requirements Engineering*, (Colorado Springs, CO, April 15 - 18, 1996).

[3] Aspect Oriented Software Development, http://www.aosd net

[4] Araújo, J., Moreira, A., Brito, I. and Rashid, A. Aspect-Oriented Requirements with UML, In the *Aspect-Oriented UML Workshop*. Collocated to the 5fth International Conference on the UML, (Dresden, Germany, October 4, 2002).

[5] Brito, I. and Moreira, A. Integrating the NFR framework in a RE model. In *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop*, collocated to the 3rd Aspect-Oriented Software Development Conference, (Lancaster, UK, March 22, 2004).

[6] Chung, L., Nixon, B. A., Yu, E. and Mylopoulos, J., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing, 2000.

[7] Dardenne, A., van Lamsweerde, A. and Fickas, S. Goal-directed Requirements Acquisition. In *Science of Computer Programming*, 20, (1993), 3-50.

[8] EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. European Project within the 5th Framework Program (GROWTH G3RD-CT-00794), 2003.

[9] Heaven, W. and Finkelstein, A. A UML Profile to Support Requirements Engineering with KAOS, *IEE Proceedings - Software*, 151, 1 (Feb. 2004), 10- 27.

[10] IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications, In Volume 4: Resource and Technique Standards, The Institute of Electrical and Electronics Engineers, Inc. IEEE Software Engineering Standards Collection.

[11] ISO/IEC Standard 9126-1 Software Engineering- Product Quality-Part1: Quality Model, ISO Copyright Office, Geneva, June 2001

[12] Lamsweerde, A. van. From System Goals to Software Architecture, In *Formal Methods for Software Architecture*, LNCS 2804, Springer-Verlag, (2003), 25-43.

[13] Lamsweerde, A. van. Goal-Oriented Requirements Engineering: A Guided Tour. In Proceedings of *5th IEEE International Symposium Requirements Engineering*, (Toronto, Canada, August 27-31, 2001), 249-263

[14] Meyer, J.J-Ch. Approach to Deontic logic: Deontic Logic viewed as Variant of Dynamic Logic. In *Notre Dame Journal of Formal Logic*, 29 (1988), 109-136.

[15] Moreira, A., Araújo, J., Brito, I.. A Requirements Model for Quality Attributes. In the *Workshop on "Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design"*. Collocated to the 1st International Conference on Aspect-Oriented Software Development, (Twente, Enschede, Holland, April 22-26, 2002).

[16] Navarro, E., Ramos, I. and Pérez, J. Goals Model-Driving Software Architecture, In Proceedings of the *2nd International Conference on Software Engineering Research, Management and Applications*, (Los Angeles, CA, USA, May 5-8, 2004), 205-212.

[17] Navarro, E., Ramos, I. and Pérez, J. Software Requirements for Architectured Systems. In Proceedings of the *11th IEEE International Conference Requirements Engineering,* (Monterey, CA, September 8-12, 2003), 365-366.

[18] Navarro, E. and Ramos, I. Requirements and Architecture: a marriage for Quality Assurance. In *Proceedings of the 8º Jornadas de Ingeniería del Software y Bases de Datos*. (Alicante, Spain, November 12-14, 2003), 69-78.

[19] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro. E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. In *Proceedings of the 3rd IEEE International Conference on Quality Software*, (Dallas, Texas, USA, November 6 - 7, 2003), 59-66.

[20] Rashid, A., Moreira, A., Araújo, J. Modularisation and composition of aspectual requirements. In Proceedings of the *2nd International Conference on Aspect-Oriented Software Development*, (Boston, Massachusetts, USA, March 17 - 21, 2003), 11-20.

[21] Suzuki, J. and Yamamoto, Y. Extending UML with Aspects: Aspect Support in the Design Phase. In the *Aspect Oriented Programming Workshop*. Collocated to the 13th European Conference on Object Oriented Programming (ECOOP'99) Springer LNCS 1743, (Lisbon, Portugal, June, 1999).

[22] UML 2.0: Superstructure Specification, OMG Adopted Specification. Ptc/ 03 August 2003.

Goals Model-Driving Software Architecture*

Elena Navarro
Department of Computer Science, UCLM, Albacete, Spain
enavarro@info-ab.uclm.es

Isidro Ramos & Jennifer Pérez
Department of Information Systems and Computation, UPV, Valencia, Spain
{iramos |jeperez}@dsic.upv.es

Abstract

On every new development new issues are taken into account to cope with either environmental or the stakeholders' needs that are evolving over time. Several approaches have faced this problem. Some of them exhibit this evolution ability by using static design/compile-time techniques whereas others introduce this ability in the system at run-time. Nevertheless, in both cases evolving requirements give rise to the need of adaptability which is inherent to every software development.
This paper sketches our work in this field in which we are concerned about the ATRIUM methodology to guide the reflexive development of architectures from the software requirements. In particular, we are detailing the first step of this methodology, i.e., the definition of the goals model whose constituents are the fundamental basis for the overall process defined in ATRIUM proving its suitability for obtaining traceable architectural models. It provides our work either to its ability to specify and manage positive and negative interactions among goals or to its capability to trace low-level details back to high-level concerns.

1. Introduction

Dynamism and evolution are currently two main software concerns. On every new development new issues appear related to the customisation of the software. They attempt to accomplish either environmental or the stakeholders' needs that are evolving over time. How to overcome deficiencies and limits exhibited by traditional development methodologies is a clear challenge that has been faced by several techniques. Some of them try to introduce this evolution ability by using static design/compile-time techniques whereas others introduce this ability in a way that evolution appears at run-time.

Aspect Oriented software Development (AOSD) [1] is related to the first type of techniques providing advantages in expressiveness by the separation of concerns. Both functional and non-functional needs, such as performance or compatibility, of the system's behaviour can be separately acquired and specified across the development lifecycle. From requirements [24] to architecture [23], design [27] and implementation [13], each concern has to be traced and easily adapted to new stakeholders' needs.

Concerning the second category of techniques, existing approaches provide high-level architectural descriptions that are well suited for runtime evolution. Architectural reflection [28] is a clear example in this sense, offering facilities for dynamic reconfiguration, in terms of composition and/or connection, to existing systems without modifying them. Several approaches [19, 23] make use of a meta-level in such a way that software systems are aware of their architectural properties and are able to adapt themselves at runtime in a simple way.

In both cases, however, evolving stakeholder's requirements give rise to the need of adaptability. It is inherent to every software development that expectations about the system are evolving over time. The Requirements Engineering process (RE) establishes the foundation on which the system-to-be should be implemented. Therefore, it has to be able to identify and define this need of adaptability into its artefacts in such a way that they can be traced to low level abstraction artefacts.

A methodology ATRIUM (Architecture generaTed from RequIrements applying a Unified Methodology) [20] illustrates a process to concurrently define requirements and architectural artefacts, in the same way that Nuseibeh proposes [22] to obtain architectural artefacts that provide ability for dynamism and evolution by means of AOSD and reflexive techniques. In this work, we are detailing the first step of this methodology, i.e., the definition of the goals model whose constituents are the fundamental basis for the overall process defined in ATRIUM.

This work is organized: Section 2 introduces ATRIUM methodology. Section 3 defines the goals model. Section 4 presents a case study. Related work is presented in Section5. Conclusions round up the paper in Section 6.

2. The ATRIUM Methodology

This methodology is concerned with the definition of software architectures from functional (FR) and non-

* This work has been funded by the Spanish CICYT project DYNAMICA (TIC2003-07776-C02-02)

Navarro, E., Ramos, I., Pérez, J. Goals Model Driving Software Architecture. 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004). 2004.

functional requirements (NFR). With this aim, it provides the analyst with a guidance along the process from an initial set of requirements to an architectural instance. ATRIUM is a not a product oriented approach but a process oriented one. It does not describe a set of metrics to determine if the requirements, functional and non-functional, are implemented into the final software. On the contrary, requirements and architectural design decisions are defined concurrently in such a way that every decision is made to achieve a given requirement.

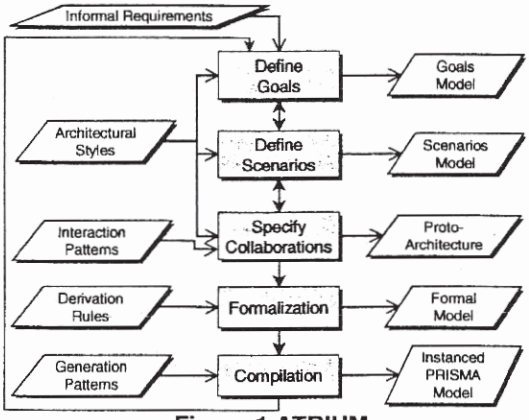


Figure 1 ATRIUM

Iterative and incremental architectural development are the two main characteristics taken into account in its conception. This concurrent description of requirements and software architecture is not as straight-forward as defining all the artefacts at once. Therefore, it seems natural to provide the analyst with an iterative process in order to incrementally develop the software artefacts. It allows him/her to reason on partial models and focus on different architectural views. ATRIUM iterates over a set of five steps (Figure 1) which are described next.

Step 1.- Goals model (GM) definition

In this step, the set of goals to be accomplished by the software system are defined by means of both functional and non-functional requirements. An informal set of requirements, stated in natural language, is the input to trigger the model definition (see section 3).

The elaboration of the Goals and Scenarios Models (see step 2) are two intertwined processes. The GM is operationalized at the next step by means of the Scenarios Model (SM). Furthermore, the analysis information provided by the SM helps us to refine and identify new goals at the next iteration. Therefore, both models are coupled, as in [25], with a meaningful advantage in terms of traceability.

Step 2.- Scenarios model (SM) definition

The analyst has to identify the set of scenarios which operationalize the established goals and compose them to form an iterating and branching model of the system's behaviour. Each scenario depicts the elements that interact to satisfy a specific goal and their level of responsibility in achieving a given task. These elements are *shallow-components*, i.e., a rough description of the components that appear into the final software architecture.

Use Cases(UC) and Message Sequence Charts (MSC) [29] are employed for the construction of the SM. The former provide us with a visual metaphor notation for scenarios, where the identified shallow components are the involved actors. Nevertheless, when we speak about Software Architecture[7], we have to identify not only the involved elements but also the relationships between them. MSCs identify the coordination structure through the temporal sequences of interaction events.

Several alternative scenarios can operationalize the same goal, just like several alternative programs can implement the same specification. In order to offer the best approach, they have to be analysed caring about conflicts that may arise among the operationalized goals.

Step 3.- Collaborations definition

The third step is aware of the collaborations among the identified shallow components. The collaborations realize the UCs through collaboration diagrams. The objective is to obtain a proto-architecture for the concrete system by using related architectural styles [26], interaction patterns and the architect's ability. Additionally, the connectors are defined according to the components interactions. These first class citizens are required to achieve a loose coupling between the components.

Step 4.- Formalization

A semantic check and analysis of the Models and the proto-architecture is required to identify eventual conflicts, e.g. different scenarios resulting in incompatible architectural configurations, and obtain the best alternative to avoid (or minimize) them. The artefacts, defined in the previous steps, are specified in the 3APL language[10], whose interpreter helps us to validate them.

A set of derivation rules are provided to generate a formal specification from scenarios, goals and collaborations, in order to assist and speed up the formalization process. This specification is validated and then used for an automatic compilation process in the next step.

Step 5.- Compilation

Using the formal model and a set of generation patterns, the translation from the requirements model to an instantiated PRISMA model specification is accomplished. PRISMA [23] (PlatafoRma oasIs para Modelos Arquitectónicos, OasIs Platform for Architectural Models) has been selected as the reference model to specify architectural elements by integrating Component-Based Software Development (CBSD) [30] and AOSD [1] approaches. This architectural model is able to be compiled into a concrete target system preserving its reflexive properties. This step is assisted by the engineer in order to refine the architectural elements.

As ATRIUM is intended to be iterative and incremental, a feedback is provided from Step 5 to 1. In this way, all the models are up-to-date all over the process.

3. Goals Model: An Artefact for RE

In the context of Requirements Engineering, the *Goal-Driven Requirements Engineering* paradigm [15] has proven useful to elicit and define requirements. More traditional systems analysis tools, such as UCs, focus on establishing the features (i.e. activities and entities) that a system will support. Nevertheless, Goal-based approaches [5, 6] focus on *why* systems are constructed providing the motivation and rationale to justify software requirements. They are not only useful for analyzing goals but also for elaborating and refining them.

Additionally, we have to bear in mind that architectural models are a bridge between requirements and the system-to-be [7] providing us with a lower abstraction level. They are used as intermediate artefacts to analyse whether the requirements are met or not. Therefore, this paradigm has two advantages that make it appropriate to systematically guide the selection among several architectural design alternatives:

- Its ability to specify and manage positive and negative interactions among goals [5, 16] allows the analyst to reason about design alternatives.
- Its capability to trace low-level details back to high-level concerns [6] is very appropriate to bridge the gap between architectural models and requirements.

These are the main reasons why the *Goals Model* has been introduced as an ATRIUM artefact to identify and describe the users' needs and expectations, their relationships and how these can be met for the target system. With this aim, several building blocks are used (see section 3.1) that are related to each other by a set of special relationships (see section 3.2) via a refinement process that uses both kinds of elements to build the Goals Model.

3.1. Building Blocks for the Goals Model

As was stated above the GM provides a number of abstractions in terms of which constraints on the software system have to be defined. A key element employed in its construction is a *goal*. It is defined as an objective that the system-to-be should achieve [15], i.e., a constraint or obligation that the system should meet. In its definition it is stereotyped as *Functional* or *Non-Functional*, according to the type of need or expectation it refers to:

- *Functional goals* describe services that the system provides, i.e., the transformations the system performs on the inputs.
- *Non-Functional goals* refer to how the system will do these transformations, for instance, in terms of performance, adaptation, security, etc. We are highlighting them because they are especially meaningful in terms of software quality [21].

Additionally, other aspects have to be stated when a goal is defined. For instance, a set of *preconditions* and *postconditions* has to be identified. Preconditions establish which situations must hold before some operation is performed. Postconditions define the situations that have to be achieved after some operation. Their evaluations help us to determine the best design alternatives among those that satisfy the postconditions for the established goals.

Moreover, each goal has to be classified according to its priority, from *very high* to *very low*, for the system-to-be. This classification helps the analyst to focus on the important issues. These priorities can arise from several factors: organizational ones when they are critical to the success of the development, constraints on the development resources, etc.



Figure 2 Visual notation for goals

A textual notation for goals definition, along with a visual notation (Figure 2), is provided to deal with the previous aspects:

```
GOAL GoalName
ID identifier
TYPE [Functional | NonFunctional]
DESCRIPTION ShortDescription
PRIORITY [Very High | High | Normal| Low,
Very Low]
AUTHOR authorName
PRE conditions
POST conditions
```

Aside from goals, other building blocks for the GM are the *Operationalizations*. When an analyst has refined the initial set of goals, he/she must offer a set of solutions that allow the system to achieve the established goals. These solutions provide architectural design choices for the target system which meet the users' needs and expectations. They are called *operationalizations* because they describe the operation of the system, i.e., the system behaviour, to meet functional and non-functional requirements. Their textual notation, together with their visual notation (Figure 3), is:

```
OPERATIONALIZATION OpName
ID identifier
DESCRIPTION ShortDescription
AUTHOR authorName
PRIORITY [Very High | High | Normal| Low,
Very Low]
```



Figure 3 Visual notation for Operationalizations

It can be noticed that there is no section in the textual notation to define the alternative solutions to satisfy a given goal. In contrast, it is in the SM where these solutions are expressed. However, entities known as operationalizations are introduced in the GM to represent conceptually each solution so that relationships among the different alternatives can be established within the GM. Operationalizations imply a coupling between the SM and the GM, and traceability between operationalizations and a

specific view of the SM is achieved by means of *identifiers*.

Additionally, operationalizations have a special property called *Priority* to express how meaningful a given solution is for the system, i.e., its relevance for the system development. It is used to face the tradeoffs between operationalization choices in case there are conflicts among them.

We can observe in the textual notation that operationalizations are not stereotyped in a similar manner to goals. This is because the same solution can be associated to different goals, both functional and non-functional. Therefore, they are not specifically stereotyped.

3.2. Relationships: An Element in the Refinement Process

The stated building blocks, goals and operationalizations, are inter-related by means of a set of *relationships*. They are in charge of gluing the different elements to complete the model and enhance its cohesion. Moreover, their relevance is not only restricted to this gluing but also they allow the analyst to introduce the rationale of the system design. The decomposition of goals or how an operationalization positively or negatively contributes to a goal, can be defined via relationships.

They are applied via a stepwise refinement process which takes an informal set of requirements, stated in natural language, as an input to trigger the model definition, and using some of the proposals in [2] to identify goals. In such way, every refinement step provides a new set of goals and/or operationalizations for the model. Therefore, the analyst ought to deal with a reduced set of building blocks at each step.

There are two types of refinements that can be applied: intentional and operational. The former describes how a goal can be reduced into a set of subgoals via *AND/OR* relationships. The latter depicts how a set of solutions address a goal by means of *AND OPERATIONALIZE / OR OPERATIONALIZE* relationships. Both, building blocks and relationships are structured as an acyclic goal graph, where the refinement is achieved along the structure, from the higher to the lower level, by applying intentional and operational refinements.

An *AND* relationship (Figure 4 (a)) between a goal *GoalX* and a set of sub-goals G_1, ..., G_N is established if the sub-goals have tobe satisfied in order to satisfy *GoalX* . Its textual notation is described as:

```
AND(Goal1 {, GoalN}) ACHIEVE GoalX
```

A goal *GoalX* is related to a to a set of sub-goals G_1, ..., G_N via an *OR* relationship (Figure 4 (b)) if *GoalX* is satisfied if at least a sub-goal is satisfied. Its textual notation is described as:

```
OR(Goal1 {, GoalN}) ACHIEVE GoalX
```

Every goal, which is too coarse-grained to be directly addressed by a solution, is refined in a set of subgoals which are a decomposition of the original one. Whenever a sub-goal is needed to achieve a goal, an AND relationship is established between them. On the contrary, if the sub-goal may optionally appear, then an OR relationship is established between them.

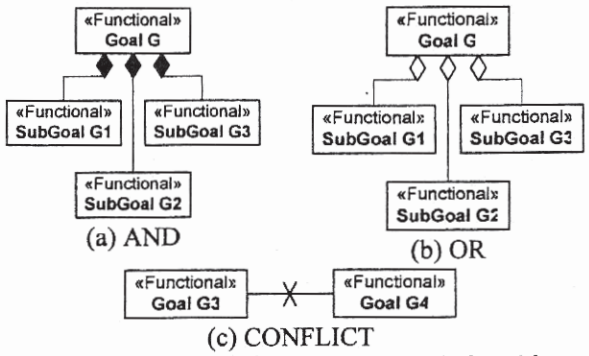


Figure 4 Visual notation for Goals Relationships

Additionally, a CONFLICT relationship (Figure 4(c)) can be set up among two goals if an incompatibility appears between them, in other words, whenever the satisfaction of a goal prevents the satisfaction of another goal. It is described as:

```
CONFLICT Goal1, Goal2
DESCRIPTION shortDescription
```

An operational refinement deals with operationalizations and goals. The alternative solutions for each goal are established by means of this decomposition. There can be a large number of valid operationalization methods that are applicable to a goal. In such a case, it is up to the analyst to examine the impact of such methods on other types of requirements and decide on what and how many operationalizing methods must be applied via AND OPERATIONALIZE/OR OPERATIONALIZE relationships. An *AND OPERATIONALIZE* relationship (Figure 5 (a)) relates the set of mandatory solutions for a goal. On the other hand, whenever several alternative solutions can be provided for a goal, the analyst can introduce them in terms of the *OR OPERATIONALIZE* (Figure 5 (b)) relationship. Their textual notation is:

```
([++|+|#|-|--]Oper2, {, [++|+|~|-|--
]Oper1}) AND OPERATIONALIZE [Goal| Oper1]
([++|+|#|-|--]Oper2, {, [++|+|~|-|--
]Oper1}) OR OPERATIONALIZE [Goal| Oper1]
```

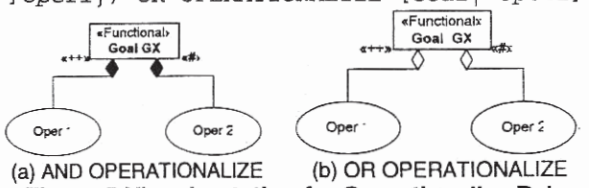


Figure 5 Visual notation for Operationalize Rel.

It is shown in this description that a set of symbols [++|+|#|-|--] are used to stereotype the relationship. They denote how an operationalization collaborates to achieve a goal. Symbols ++ and + describe a positive collaboration, i.e., it provides a sufficient or partially sufficient solution, respectively, to satisfy the related goal. On the other hand,

symbols -- and - describe a negative collaboration, i.e., the operationalization prevents or partially prevents, respectively, the satisfaction of the related goal. The # symbol is introduced to specify operationalizations whose impact (positive or negative) is unknown at the moment and is the default stereotype.

OPERATIONALIZE relationships do not only relate operationalizations to goals but also to other solutions. It is used to refine the operationalizations down to other simpler ones, i.e., to describe how a solution can be expressed in terms of a set of simpler solutions. AND OPERATIONALIZE relationships relate a set of operationalizations whose composition satisfies another one. OR OPERATIONALIZE relationships are used when alternative sub-operationalizations exist. Again, these relationships are positively or negatively stereotyped.

The intentional refinement is iteratively applied to the set of goals ending up when every sub-goal can be operationalized. Similarly, the operational refinement is sequentially applied to operationalizations until these are simple enough to draft a scenario.

3.3. Goals model and Aspects: How to Solve the Tangled Components

As was stated above, AOSD provides a mechanism to specify and relate concerns (solving the crosscutting). In this respect, these can be handled adequately avoiding that they appear tangled over the system and providing significant advantages in terms of understanding, maintenance and evolution. Concerns may or not result in concrete aspects in the end system. We need to bear in mind that PRISMA, the selected model to instantiate architectures, follows an aspect-oriented approach. Components (and connectors) are defined by a gluing of aspects, i.e., they are the *melting pot* for them. Therefore, it is necessary to identify what aspects are and which components (and connectors) are crosscut by them.

An issue arises when defining the term "concern", and more specifically, when deciding how it appears in our approach. It is difficult to answer this question because there is not a well-established understanding of the notion of concern. In AOSD, the term concern is understood as the so-called functional and non-functional properties. Our position is similar: the concerns of the system are like functional and non-functional goals. This answer may help to understand why the GM is so important in this work. The concerns are being established when we are identifying the goals (and sub-goals) of the system, by means of the iterative process stated above.

The concerns of the system are specified as nodes of the goals graph. Moreover, the GM assists us in identifying one of the main issues in AOSD: which concerns are being crosscut by others, i.e., which concerns are in conflict. The initial set of goals (concerns) could seem that interact in synergy, without any conflict. But the iterative refinement process can help us to identify if crosscutting appears. In this case, a *conflict relationship* is established between each pair of conflicting sub-goals. In order to solve these conflicts, the priorities defined over the relationships between goals and sub-goals determine the dominant sub-goal. A trouble may appear when several sub-goals show an equal priority. In such a case, the analyst must define a trade-off among them.

We have been speaking about the concerns of the system, but what about the aspects? Several works[24] have stated that concerns may or not be aspects in the end system. This is determined within ATRIUM by means of the SM which scatters the aspects over the components and connectors by using the UCs and the MSCs. The UCs identify as actors the components, the subsystems or systems which are collaborating to operationalize a goal, i.e., a concern. As a result, the UCs where the involved actors are only components will be identified as aspects. In such a way, the set of UCs, where a component C_1 is involved, is the set of aspects that compose C_1.

4. Applying ATRIUM to a Case Study

Grid Applications [11] support a new model of computing that has emerged from conventional distributed computing, but focusing on large-scale resource sharing and innovative scientific applications. This new type of computing environments raises new challenges with respect to coordination, namely the heterogeneity of the coordinated processes, the volume and volatility of the shared information and the potentially vast number of processes that require an homogeneous way of communication, abstracting the machine-dependent details. Moreover, several works [4, 12] have recognized their need for dynamic self-adaptation to changing environments to deal with resource variability, system faults, etc.

Hoschek et al show in [11] some of the exhibited requirements for a Data Grid. In particular, one of their real-world applications areas is the High Energy Physics (HEP) to allow a wide and distributed number of researchers to analyse generated data by large particle accelerators. HEP is used to design experiments, e.g. beams of protons accelerated in opposite direction, which are analysed in order to try to understand physical laws. An analysis that is carried out over a huge amount of data by interactive and batch processes but with a high performance in terms of time and space.

Moreover, a previous work [12] showed how a set of Grid Computing Nodes exhibit some needs in terms of efficiency as CPU load, number of waiting processes, or available memory. Grid Computing Nodes gather this context information. An efficient task scheduling is needed in order to optimise the available resources, i.e., self-configurable or auto-adaptive applications which perform architectural evolution based on contextual information.

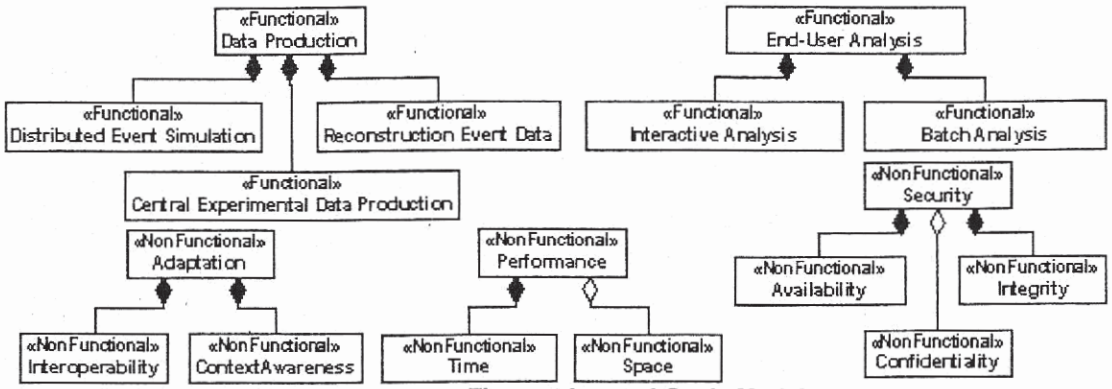


Figure 6 Iterated Goals Model

Several Functional and Non-Functional Goals can be established from these sentences. They provide us an oversimplified view of this kind of applications, but which will allow us to understand how GM is defined and its relationship with the concerns of the system-to-be. Specially, it will show how GM and SM are interleaved along the process.

Iteration 1 - Step 1 Goals Model. The initial set of Requirements is presented as five nodes in Figure 7 obtained from the previous sentences. *Data production* deals with experiments design. *End-User Analysis* with the required process to evaluate generated data. The *Performance* requirement is due to high efficiency required from this kind of applications. *Adaptation* appears because of the self-configurability or auto-adaptability that is needed. Finally, *Security* is introduced to deal with information protection.

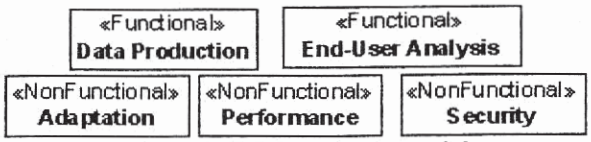


Figure 7 Original Goals Model

Iteration 2 - Step 1 Goals Model Definition. Several subgoals can be identified in order to satisfy the previous ones, as it is shown in Figure 6. For instance the *Adaptation* goal can be intentionally refined up to *ContextAwareness* and *Interoperability*. The former gathers context information to adapt the system behaviour and the latter manages the heterogeneity of related elements. Because both sub-goals are required to achieve the Adaptation goal, an AND relationship has been established between them. In a similar manner, *Time* and *Space* are the intentional refinement of Performance. However, Space and Performance are OR-related because there is no high restriction in this sense.

Iteration 3 - Step 1 Goals Model Definition. Some of the identified goals can be operationally refined. ConextAwareness and *ContextAcquisition* are AND OPERATIONALIZE-related because the latter is a required solution for the former (Figure 8).

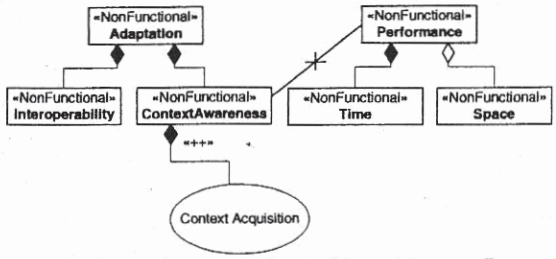


Figure 8 Iterated Goals Model (part of)

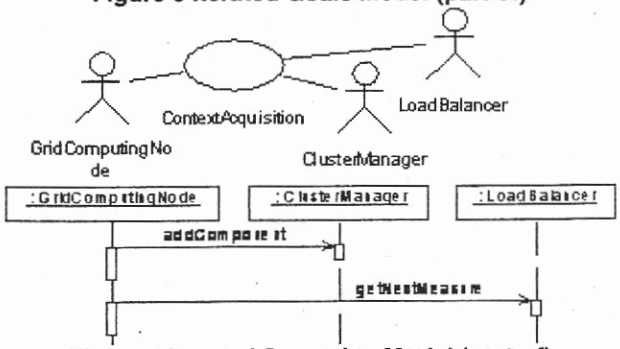


Figure 9 Iterated Scenarios Model (part of)

Iteration 3 - Step 2 Scenarios Model Definition. In order to operationalize the subgoal *ContextAware*, three actors can be identified: *GridComputingNode*, *Cluster Manager* and *LoadBalancer*. The first one has to gather context information and distribute it to the LoadBalancer and the ClusterManager in order to optimise the resources. UC and the MSC, for this sub-goal are showed in Figure 9.

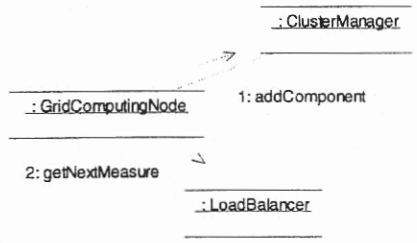


Figure 10 A view of collaboration

Iteration 3 - Step 3 Collaborations Definition. Because MSCs are closely related to collaboration diagrams it is very easy to get the latter. The result for the ContextAware goal can be observed in Figure 10. It is appreciated that the

three components (GridComputingNode, LoadBalancer and Cluster Manager) have to be disguised with this goal (aspect). Moreover, to get a loosely coupling between the involved components, a connector is defined to aggregate and analyse the relationship between them.

Iteration 3 - Step 4 Formalization. The defined models will be formalized in this step, but a detailed description is outside of the scope of this paper.

Iteration 3 - Step 5 Compilation. According to the information provided by the previous models and the proto-architecture, and after validating via 3APL, the PRISMA model can be instantiated, describing the identified elements and the configuration that depicts their relationships (Figure 11). Both GridComputingNode, LoadBalancer and ClusterManager are shallow components that have to be fully described by the analyst. This proto-architecture can be influenced when the conflict relationship (Figure 8), established between ContextAwareness and Performance is resolved.

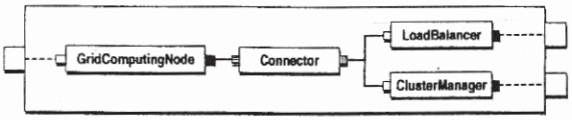


Figure 11 A preview of the Instantiated PRISMA model

We have used the previous example because the inherent dynamism of these applications. Nevertheless, the process can be instantiated in any domain where the software architecture has to deal with goals and allocation of responsibilities.

5. Discussion and Related Work

In this section, we briefly review the related works. Each of them provides the analyst with a different approach to the concurrent definition of requirements and architecture.

Brandozzi and Perry [3] have suggested the use of intermediate descriptions between requirements and architectures known as "architectural prescriptions". These describe the mappings between requirements aspects and architectural descriptions. Nevertheless, it is not clear that the overhead of an additional description is practical. Recent work on software product lines and system families has also examined the relationship between architectures and requirements. The work has focused on identifying core requirements (identified through a process of requirements prioritising) and linking them to core architectures (identified by examining the stability of various architectural attributes over time).

Hall et al [9] show an extension to *problem frames*. Problem frames define requirements as dependencies among the system and the world outside of the computer. They help the developer to focus on the problem domain, establishing a clear separation between the world and the machine. This solution is amenable to perform iterative development due to the interaction among requirements, architecture, and design: each one informing about the others and vice versa.

Grunbaccher et al [8] explore the relationships between software requirements and architectures, and propose an intermediate model, CBSP (Component - System – Bus - Property), to depict the dependencies among the key architectural elements and the stated system requirements. These dependencies are established in a polling process where the involved architects classify each requirement with respect to its architectural relevance. Based on the CBSP model elements a proto-architecture can be derived with the selection of the appropriate architectural style. Therefore, it is a systematic approach, not an automated one, which is only applied in a specific context.

Lamsweerde [15] proposes a goal-oriented approach to architectural design based on the KAOS framework. It defines an iterative refinement process from functional specifications to an abstract architectural draft. It is refined to meet domain-specific architectural constraints. The main restriction shown by this approach is related to the use of non-functional requirements. They are only used to made decision among several alternatives but they are not as relevant as functional ones.

Liu et al [17] propose a similar approach by combining scenarios and goals-based models during architectural design. The GRL language is used to define the latter and the UCM notation is used for the former. The main problem that we observe is that both models are loosely coupled resulting eventually in a lack of consistence between them.

The main advantage shown by this approach with respect to the previous is related to the guided and semi-automated process support given from requirements to architectures. Also the coupling of requirements artefacts provides a better assessment of the design alternatives. Moreover, due to the traceability among requirements and architectures, along with the reflexive properties of PRISMA, running-systems can evolve over time while remaining compliant with their evolving requirements.

6. Conclusions and Future Work

Summarizing, in this paper we show how to address the iterative development of requirements and architectures during the development of software systems. A methodology, ATRIUM, that guides the analyst, from an initial set of requirements to an instantiated, architecture has been presented. It uses the strength provided by the coupling of scenarios and goals to systematically guide through the iterative process. Moreover, it allows the traceability among both artefacts to avoid lacks of consistency.

Additionally, requirements and architectures can evolve

iteratively and concurrently, in such a way that running-systems can dynamically adapt their composition and/or topology to meet their evolving requirements. It is granted by the reflexive properties of PRISMA and by the relationships between requirements and architectures that have been established along the process.

The GM, a requirement artefact used in this process, has been defined in this work. It has been designed to allow the analyst to reason about design alternatives by using the defined relationships. Additionally, the refinement process compels him/her to focus on a specific view of the system-to-be definition and, therefore, on a partial view of the architectural model, owing to its ability to trace low-level details back to high-level concerns.

There are questions and problems that remain open: the completeness of each scenario and the completeness of the set of scenarios; the compatibility and consistency of the architectural styles; the creation of the architectural structure or its transformation on the basis of NFRs. All these subjects constitute our future work.

7. References

[1] .AOSD, http://www.aosd.net

[2] A. I. Anton, "Goal-Based Requirements Analysis", Proc. 2nd Int. Conf. on RE, Colorado Springs, CO April 15 - 18, 1996.

[3] M. Brandozzi and D. E. Perry: "Architectural Prescriptions for Dependable Systems", Proc. of the International Conference on Software Engineering, Orlando, May, 2002

[4] S. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, and N. Hu, "Software Architecture-based Adaptation for Grid Computing", Proc. 11th IEEE Conference on High Performance Distributed Computing, Edinburgh, Scotland, July 2002.

[5] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing, 2000.

[6] A. Dardenne, A. van Lamsweerde, and S. Fickas: "Goal-directed Requirements Acquisition". Science of Computer Programming, 20:3-50, 1993.

[7] D. Garlan: "Software Architecture: a Roadmap, in The Future of Software Engineering", Proc. ICSE. 91-101, Limerick, Ireland, June 4-11, 2000, ACM, pp. 91-101.

[8] P. Grünbacher, A. Egyed and N. Medvidovic: "Reconciling Software Requirements and Architectures: The CBSP Approach". Proc. 5th IEEE Int. Symp. RE, 27-31 August 2001, Toronto, Canada,pp 202-211.

[9] J. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti, "Relating Software Requirements and Architectures using Problem Frames", Proc. 10th IEEE International Conf. on RE , Essen, Germany, 9-13 September 2002.

[10] K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J.Ch. Meyer, "Agent programming in 3APL". Autonomous Agents and Multi-Agent Systems, 2(4):357-401, 1999

[11] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger "Data Management in an International Data Grid Project". 1st IEEE/ACM International Workshop on Grid Computing, Bangalore, India, December 17-20 2000.

[12] J. Jaén and I. Ramos, "A Conceptual Model for Context-Aware Dynamic Architectures", 3rd Int. Workshop on Distributed Auto-adaptive and Reconfigurable Systems, May 19-22, 2003 Providence, Rhode Island USA.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming". Proc. European Conference on Object-Oriented Programming, Finland. Springer-Verlag LNCS 1241. June 1997.

[14] A. van Lamsweerde, "From System Goals to Software Architecture", Formal Methods for Software Architecture, LNCS 2804, Springer-Verlag, 2003, pp. 25-43

[15] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", Proc. 5th IEEE Int. Symp. on RE, Toronto, August, 2001, pp. 249-263

[16] A. van Lamsweerde, R. Darimont, E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", IEEE Trans. on Software Engineering, Vol. 24, N. 1, January 1998.

[17] L. Liu, E. Yu: "From Requirements to Architectural Design - Using Goals and Scenarios, From Software Requirements to Architectures". Proc. 1st Int. Workshop From Software Requirements to Architectures (collocated ICSE), Toronto, Ontario, Canada, 12-19 May 2001.

[18] N.A.M. Maiden. "CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements", Automated Software Engineering, vol. 5, pp. 419-446, 1998.

[19] N. Medvidovic, D. S. Rosenblum and R. N. Taylor. "A Language and Environment for Architecture-Based Software Development and Evolution". Proc. 21st Int. Conf. on Software Engineering, May 1999, pp. 44-53.

[20] E. Navarro, I. Ramos and J. Pérez: "Software Requirements for Architectured Systems". Proc. 11t h IEEE International Conference RE (Poster), September 8-12, 2003 in Monterey, CA.

[21] E. Navarro, I. Ramos and J. Pérez: "Requirements and Architecture: a marriage for Quality Assurance". 8° Jornadas de Ingeniería del Soft. y Bases de Datos. November 12-14, 2003.

[22] B. Nuseibeh, "Weaving the Software Development Process Between Requirements and Architecture", Proc. 1st Int. Workshop From Software Requirements to Architectures (collocated ICSE), 12-19 May 2001, Toronto, Ontario, Canada.

[23] J. Pérez, I. Ramos, J. Jaén and P. Letelier, E. Navarro: "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", 3rd IEEE International Conference on Quality Software, Dallas, Texas, USA, November 6 - 7, 2003.

[24] A. Rashid, A. Moreira, J. Araújo, "Modularisation and composition of aspectual requirements", Proc. 2nd international conference on Aspect-oriented software development, March 17 - 21, 2003, Boston, Massachusetts.

[25] C. Rolland, G. Grosz and R. Kla: "Experience with Goal-Scenario Coupling in Requirements Engineering", IEEE Int. Symposium on RE, June 7-11, 1999, Limerick, Ireland.

[26] M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proc. COMPSAC'97, Washington, DC, Aug. 1997.

[27] J. Suzuki and Y. Yamamoto, "Extending UML with Aspects: Aspect Support in the Design Phase", AOP Workshop at ECOOP'99, Lisbon, Portugal, 1999.

[28] F. Tisato, A. Savigni, W. Cazzola, and A. Sosio. "Architectural reflection realising software architectures via reflective activities". eDO 2000, LNCS, Berlin, 2000.

[29] Unified Modeling Language™, http://www.u2-partners.org/

[30] Szyperski, C., Component software: beyond object-oriented programming. Addison Wesley, 1998.

High Level Specification of Distributed and Mobile Information Systems

Nour H. Ali, Jennifer Perez, Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n
E-46071 Valencia – Spain
{nourali, jeperez, iramos}@dsic.upv.es

Abstract. Nowadays, information systems are large and complex to develop. An important factor that influences in this complexity is that information systems are tending to be distributed with mobile components. Many technologies have emerged in dealing with distribution issues at an implementation level. However, few approaches have considered distribution from the beginning of the life cycle of software development and have dealt with distribution at a high abstraction level. In this paper, we focus on specifying distributed and mobile information systems at a high abstraction level using an approach called PRISMA. PRISMA is an architectural model which combines the Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD) to describe software architectures. PRISMA has associated an Architectural Description Language (ADL) that is separated into a Type Definition Language and a Configuration Language. We use the Type Definition Language to specify the distribution aspect as a first order citizen of the language. Furthermore, we use the Configuration Language to configure the location of the instances of the software architecture.

1 Introduction

Nowadays, information systems are large and complex to develop. An important factor that influences in this complexity is that information systems are tending to be distributed with mobile components. Many technologies have emerged in dealing with distribution issues at an implementation level. On the other hand, few approaches have dealt with distribution at a high abstraction level. Nevertheless, considering distribution in the whole life cycle of software development minimizes time and costs. Thus, efforts are reduced in the development process by taking into account distribution at an early phase, instead of only introducing it at the implementation phase.

Moreover, considering distribution at analysis and design phases of software development generates high quality distributed applications. This increment of quality is due to taking into account distribution from the beginning of software development

Ali, N., Pérez, J., Ramos, I. High Level Specification of Distributed and Mobile Information Systems. Proc. of the Second International Symposium on Innovation in Information & Communication Technology (ISSICT2004). 2004.

so applications are prepared to support its non-functional requirements. However, if distribution is not considered from the beginning, applications should be changed when they arrive to an implementation phase because they are not prepared to support distribution and the traceability between phases of the life cycle is usually lost.

As a consequence of the poor capability of the object-oriented model to describe complex structures of information systems, the Component-Based Software Development (CBSD) and the Aspect-Oriented Software Development (AOSD) have emerged. The CBSD [20] promises to control the complexity of system construction by coupling entities that provide specific services. The AOSD [4] allows separation of concerns by modularizing crosscutting concerns in a separate entity: the aspect. Aspects can be reused and manipulated independently of the rest of properties of the system.

PRISMA [17] is an approach that defines software architectures integrating the CBSD and AOSD in order to take advantage of both approaches. The integration is done by defining types of software architectures via a set of aspects. These aspects can evolve and be reused independently of the rest of aspects. In addition, PRISMA has an associated Architectural Description Language (ADL). Its ADL is separated into a Type Definition Language and a Configuration Language. The Type Definition Language defines architectural elements at a type level of the software architectures by different aspects. The Configuration Language designs the topology of the software architecture by creating and interconnecting instances of the defined types.

In this paper, our aim is to specify distribution and mobile properties at a high abstraction level using the PRISMA approach. Distribution and mobility are addressed by introducing the distribution aspect which is in charge of the dynamic locations of the instances. The distribution aspect is integrated by specifying an associated ADL. As the PRISMA ADL is separated into the Type Definition Language and the Configuration Language, distribution takes advantage of this separation. In this way, the distribution can be configured at an instance level allowing the reusability and independence of the aspect. Furthermore, communication between the elements of the architecture is independent of the location of the instances.

The paper is organized as follows. Section 2 presents some related work to ours that can be found in the literature. In section 3, we present the PRISMA architectural model and how it combines the AOSD and the CBSD. In section 4, the distribution aspect is proposed and showing how communication is established between the distributed components and connectors. Furthermore, mobility of elements is introduced. In addition, an example of a distributed bank system is specified in PRISMA. Finally, section 5 draws some conclusions and highlights some future work.

2 Related Work

Different programming languages have given rise to support distributed systems. Approaches as CORBA [6] and JAVA RMI [11] make distribution partly transparent for the developer. CORBA provides the applications with an intermediate layer or

"middleware" (e.g. the Object Request Broker (ORB)) that hides distribution, whereas the objects have to be registered initially by calls to the ORB. On the other hand, programmers have to know which objects will be distributed and develop interfaces to generate stubs and include them in the ORB. These approaches enable the programmer to define its components without taking into account distribution details as name servers. However, the problem of enabling the specification of an object's behaviour against events occurred in the ORB (e.g. indicating that a component has to be cloned when it receives a certain load of messages per a unit of time) is not supported. This type of problem has to be resolved by the programmer.

The CLIP group of the Madrid University of Technology [7] proposed an extension to the Ciao Prolog language for specifying the distribution of its modules. In this system, all modules are specified as if they were local and the distribution is configured with a new specification in the same language (with the extension proposed). Next, the compiler knows which modules should be distributed and these are compiled in a particular manner. It is important to take into account that the programmer separates the functionality from the distribution and only uses a single language.

Work has also been done in the conceptual modelling of mobile object systems using language constructs which can clearly distinguish between the mobile and stationary parts of a complex system. There is an extension of the TROLL language [1] in order to specify these mobile object systems. However, the work does not incorporate the flexibility of evolving the stationary parts into mobile parts.

Many architectural models with architecture description languages (ADL) have emerged such as LEDA [5], Acme [8] and Darwin [12, 15]. Acme is a well recognized ADL for defining software architectures; however, it is a static ADL that does not support evolution and reconfiguration. Darwin is an ADL that supports distributed message-passing; however, this language does not support mobility of the components.

The works in [11] describe the semantics in externalising a distribution dimension which achieves a separation between computation, coordination and distribution. Our approach is quite similar, but we use the aspect oriented approach [4] which encapsulates distribution issues of components in entities called aspects. We use the aspect-oriented approach because it supports the independence of the aspects from the functionalities thereby, achieving reusability, adaptability and evolution. In addition, it is a considerably mature and tested approach at the implementation level.

The rise of aspect-oriented programming languages introduced the encapsulation of the distribution code in the distribution aspect separating it from the rest of the code. This is done by extending known programming languages as AspectJ [18, 19] in the case of JAVA. Whilst an inconvenient of these low level languages is the definition of the pointcuts inside the aspects decreasing reusability. Therefore, to solve these issues the aspect-oriented software development approach emerged. It incorporates aspect concepts at early stages of development, as in the design phase, and supports code generation to AspectJ. In works as in [14] distribution issues as load balancing, caching, migration are not contemplated. Moreover, it does not use a general purpose language (i.e. the language used to specify the distribution behaviour is different from the one used to specify the functional behaviour of the architecture).In fact, when describing the functional entity (the object) Java is used,

whereas for defining the distribution aspect another language is used and for composing the aspect with the object a third language is used.

3 PRISMA Architectural Model

PRISMA [17] is a model to define architectures of complex software systems. The PRISMA architectural elements are defined by different aspects: functional, quality [16], coordination, context-awareness [10] and many others. The types of aspects forming an architectural element may vary depending on the architectural element and on the information system under consideration.

An architectural element (component, connector, system) of our architectural model can be seen as a prism. This prism has one side for each aspect (see figure 1). The most common aspects an architectural element can contain are the following:

- **Functional Aspect:** The functional aspect captures the semantics of the information system defining its structure and behaviour.
- **Coordination Aspect:** The coordination aspect defines the business rules and the synchronisation between types during their communication. Moreover, the coordination aspect includes the choreography extending the "contract" concept [3].
- **Quality Aspect:** The quality aspect specifies the non-functional requirements.
- **Replication Aspect:** The Replication Aspect specifies the features for producing replications.

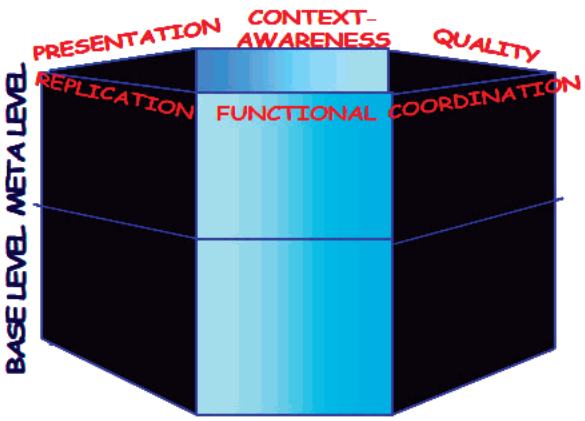


Fig. 1. A PRISMA architectural element

Our architectural model consists of different types:

- **Component:** A component is a type of the information system which does not act like a coordinator between other instances. It is formed by a set of aspects (functional, context-awareness, quality, etc), one or more

ports (**inports** and **outports**), whose type is a specific interface, and a coordination of aspects. Components interact with other architectural elements by means of their ports.

- **Connector**: A connector is a type of the information system which acts as a coordinator between instances. It is formed by aspects (coordination, context-awareness, quality, etc), a set of inroles and outroles, whose type is a specific interface, and a coordination of aspects. Connectors link and synchronise components, systems and other connectors by means of their roles.
- **System**: A system is a component that includes a set of connectors, components and other systems correctly connected. From this composition, emergent properties may arise. Moreover, a system has its own ports where binding links between its ports and the ports of its composite components are defined.

The PRISMA ADL is divided into two different levels of abstraction, the types and the interconnected instances. For this reason the ADL is presented as two languages, whose specification uses the same syntax. These languages are: the component definition language and the configuration language, respectively. Both languages are based on the OASIS 3.0 language [13] to define the semantics of architectural models. The PRISMA ADL is based on this language in order to preserve its advantages. OASIS is a formal language to define conceptual models of object-oriented information systems which allows the verification and automatic code generation of its models.

The Type Definition Language defines the types of the architecture. This language has as first order citizens: interfaces, aspects, components and connectors. The fact that interfaces and aspects are first-order citizens increases reusability because an interface may be used by several aspects and an aspect may be used by several architectural elements. Moreover, components, connectors and systems may be reused in different architectural models.

In addition, in the Type Definition Language a type is specified with the set of ports/roles, the set of aspects it is composed and the inter-aspects relationships. Each one of these relationships weaves the service of an aspect with the service of another aspect that will be executed as a consequence of the first one (crosscutting). In addition, inter-aspects relationships specify the method used to execute the service (**after**, **before** or **around**).

The configuration language is used to define instances and the topology of the architectural model. In order to do so, all needed connectors, components and systems types must be imported. Then, all the necessary instances of the imported types must be defined and, finally, the topology of the model is expressed by interconnecting such instances. Such interconnection process is achieved by creating instances of connector's synchronizations and system's bindings.

The use of different languages to specify types and topologies provides more advantages than the use of only one language. One of these advantages is the improvement of reusability and the maintenance of the types' libraries due to the independent management of types and topologies. Another advantage is the clear difference between the type evolution (Component Definition Language) and the architecture evolution (Configuration Language). In this way, there are different

evolution services in each language: evolution services to evolve types and evolution services to evolve the connections between instances to achieve dynamic reconfiguration.

4 Distribution Support in the PRISMA ADL

So far, we have described the PRISMA architectural model without considering distribution or mobility properties. In this section we aim at extending PRISMA to be able to describe systems with elements that are location-aware conserving reusability, context independence of the components (components shouldn't know to whom they are connected with neither where there connectors are located) and mobility. In this section we introduce how PRISMA supports distributed software architectures. Then, we will introduce how elements can be specified to be mobile. Furthermore, we will show how a small part of a distributed bank system is specified using the PRISMA ADL with its distribution properties.

4.1 Distribution in PRISMA

Surely, to do so in PRISMA a distribution aspect should be added to the set of aspects of the model (see figure 2). The distribution aspect specifies the features that define dynamic location of the instances. Therefore, if the system is local a distribution aspect is not added to the set of aspects of the architectural types.

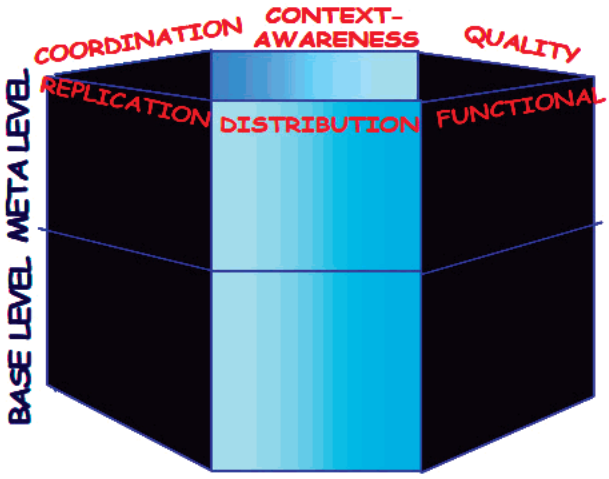


Fig. 2. A PRISMA type with distribution aspect.

The distribution aspect should specify the locations of the instances. However, if the location is specified in the distribution aspect, the same distribution aspect cannot be reused in different types if instances are distributed in different locations. This

problem is solved using the PRISMA ADL due to that it is separated into the Type Definition Language and Configuration Language. Thus, the Type Definition Language specifies that a type has a location without assigning it a value. The value of the location is assigned when types are instantiated in the Configuration Language.

The distribution aspect template will intend to specify a set of patterns detected in [2]. The patterns describe causes for the mobility and the replication of instances when faced with distribution issues such as saturation, unbalance load, latency, etc and changes in properties of the system.

```
01 Distribution Aspect name [specifies
Interfaces ];
02 Identification
03    Id: (Id);

04 Attributes
     05    Id:nat;
     06    Location : LOC NOT NULL;

07 Services
08    InService();
        Valuations
09    OutService();

10 Constraints…
11 Preconditions…
12 Triggers…
13 Operations [transactions]…
14 Protocols…

15 End Distribution Aspect name ;
```

Fig. 3. Distribution Type Definition Template.

The template of the Type Definition Language to specify the distribution aspect (see Figure 3) contains the following parts:

- The aspect specifies the semantics of the services of a set of interfaces (see line 01 in Figure 3). The services of these interfaces can be invoked by the environment.
- Each aspect has an identification mechanism that distinguishes it from the other aspects in the PRISMA library (see line 02-03 in Figure 3). This is to enable the reusability of the aspects.
- The aspect specifies its set of attributes with their valuations to specify how service invocations affect attribute values (see lines 04 -06 in Figure 3). The attribute *Location* has as an abstract data type called *LOC*. This data type hides the different mechanisms of locations of an architectural element at a physical level, e.g. it can be a

URL, an IP, etc. The *Location* attribute should have a value when an architectural element is instantiated, this is indicated by the *NOT NULL*.

- Services that allow a change in the attributes values including the interface services must be specified (see line 07-09 in Figure 3). In addition, when it is necessary to differentiate between the client and server behaviour of the same service, the service is typed as ***out*** to specify that it is a service to be requested by the architectural element type and the service is typed as ***in*** to specify that it is a service to be offered by the type. Each *inService* should specify how its execution (invocation) changes the values of the attributes by the reserved word *Valuations.*
- The constraints specify integrity restrictions that always have to be satisfied.
- The preconditions are a set of conditions that have to be satisfied in order to invoke a service.
- The triggers, as the name indicates, specify the triggers that occur.
- The operations specify a set of services considered as one unit; they could be transactional or not.
- The protocols specify how the invocation of services can be organized through time.

Another issue left to overcome is how can components communicate with each other through connectors if components and connectors are distributed in different locations preserving context independence. Therefore, we solve this problem by defining a type in the Type Definition Language called Attachments. This type contains the services that perform attachments between ports and roles of components and connectors respectively. At the Configuration language the Attachment services are used at an instance level. The Attachment type is as follows:

Type Attachment Name
Identification
Name:(name);
Attributes
name: string;
Services
create(name)**new;**
attach(Inport.Component, Outrol.Connector);
attach(Outport.Component,InRol.Connector);
End_Attachment Type;

4.2 Mobility

An instance can be mobile, if the service **move(LOC)** with its corresponding **valuation** (defining that the actual location changes to a new one) is specified in the distribution aspect that defines the architectural element type which it is an_instance_of. If the distribution aspect specifies an interface containing the service **move(LOC),** the environment can control the mobility of the instance because the **move** service is published by the interface and it can be invoked by another instance. Otherwise, the mobility is under control of the proper instance.

The distribution aspect is as follows for specifying mobility:

Distribution Aspect name **[specifies** interfaces];

Identification
Id: (Id);

Attributes
Id:nat;
Location**: LOC;**

Services
move(newLoc:**LOC**);
Valuations
[move(newLoc:**LOC**)] Location:=newLoc**;**
….
….
End Distribution Aspect name**;**

4.3 Example: Bank System

In this section, we will show how to specify using our ADL a distributed bank system. An administrator of a bank system is a mobile user. She/he determines where to move her/himself i.e. she/he controls from where to access to the bank system. Whereas, the administrator can also move a bank account to her/his location. Therefore, the mobility of the account can be controlled by the environment. As well, when the account detects that its principal bank office has changed, the account moves itself to the location of this bank office.

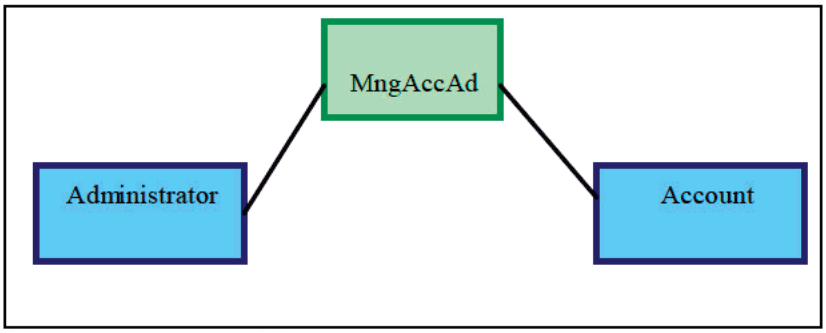


Fig. 3. The types of elements in the Bank system.

First of all, the interfaces that contain services that are shared between the different components of the software architecture are defined. The interface Mblty declares the service move(LOC) which will be shared between more than one component, in our case between the Administrator and the Account. The interface MdfyAccount declares another set of services that are shared between componets (the set of services

that the Administrator can invoke from the account). These interfaces are shown as follows:

Interface Mblty
  Services
    Move(LOC),
End Interface Mblty;

Interface MdfyAccount
  Services
    Change_Name;
    Change_PBankOffice;
    Change_AccountNumber;
End Interface MdfyAccount;

Furthermore, to specify the different distribution behaviours of the components, the distribution aspects are specified. The distribution aspect ExtMbile specifies **Mblty** interface indicating that the environment can control the location of an instance because the **move** service is published by this interface.

Distribution Aspect ExtMbile **specifies** Mblty;

Identification
  Id: (Id);

Attributes
  Id:nat;
  Location**: LOC;**
Services
  move**(**newLoc**:LOC);**

Valuations
move(newLoc**:LOC**) Location:= newLoc**;**

End Distribution Aspect name**;**

The distribution aspect Mbile specifies that it has a mobile behaviour but the service *move* is not accessible to the environment meaning that the mobility is under the type control. In addition, two *move(LOC)* services have been declared in the *Services*. The *move* declared **In** represents the server behaviour of the service i.e. declaring that **In*move*** is specified in the aspect. Whereas, the **Out*move*** specifies the client behaviour of the move i.e. that the type with this distribution aspect can control the mobility of a type in the environment.

Distribution Aspect Mbile;

Identification
  Id: (Id);

Attributes
Id:nat;
Location: **LOC;**

Services
Inmove(newLoc:**LOC**);
Valuations
[**In**move(newLoc**:LOC**)] Location:= newLoc**;**

Outmove(Location:**LOC**);
Triggers
[**Out**move(location:LOC)] **when {In** move(newLoc**:LOC**) **}**

End Distribution Aspect name**;**

After declaring the different distribution behaviours architectural elements types can have, we specify for each type a distribution behaviour. As the Administrator is independent in specifying its mobility and also can influence on the mobility of other components, it imports the distribution aspect Mbile into its definition.

Component_type Administrator
Inport … …
End_Inport;
Outport
Update_Account: MdfyAcc;
Move_Account: Mblty;
End Outport;

Import Distribution Aspect: Mbile;
….
End_Component Administrator;

The Account imports the distribution aspect ExtMbile due to that an Administrator can invoke its service move. In addition, the distribution aspect is weaved with the functional aspect through the service move and change_PBankOffice. This is to specify that after the principle bank office of the account changes (a functional property) move the account to the bank office location (a distribution property).

Component_type Account
Inport
Updates_Acc: MdfyAcc;
Moves_Acc: Mblty;
End_Inport;
Outport
…
End_Outport;

Functional Aspect
Import BankFunctionality: Functional Aspect;
Weaving
Import Distribution Aspect : ExtMbile
move(**LOC**) **after** Change_PBankOffice;
End_ Distribution Aspect
End_Weaving;
End_Functional Aspect
End_Distribution Aspect;
End_Component;

In the Configuration language the instances of the administrator and account are created specifying their location. Then the attachments between the components ports and connectors roles are created as follows:

Architectural Model BankSystem
Import Types
Components
Account, Administrator;
Connectors
MngAccAd;
Instances
Administrator1(*value_of_location*): Administrador;
Account1(*value_of_location*): Account;
MngAccAd1(*value_of_location*): MngAccAd;
End_Instances;

Attachment a.attach(Update_Account.Administrator,Update_Account.MngAccAd1);
Attachment b.attach(Move_Account.Administrator, Move_Account.MngAccAd1);
Attachment c.attach(Updates_Acc.Account, Updates_Acc. MngAccAd1);
Attachment d.attach(Moves_Acc, Moves_Acc. MngAccAd1);

End_Arquitectural Model BankSystem;

Through the following section and example we have shown how the distribution aspect is specified independently of the other concerns of the software architecture. This enables the distribution aspect to evolve and to be reused independently of the other concerns of the system.

5 Conclusions and Future Works

In this paper, we have demonstrated that distribution can be specified at high abstraction level using PRISMA model. PRISMA has been presented as an aspect-oriented component-based approach for describing distributed software architectures.

The distribution properties are separated in the distribution aspect allowing the manipulation and reusability independently of the rest of properties. This huge advantage is provided by the aspect oriented approach of PRISMA model. Moreover, types are reusable and instances are mobile and context-independent. This is due to the separation of the ADL into the Type Definition Language and the Configuration Language. This separation allows us to specify the distribution aspect as a first-order citizen and to instantiate the same type in different locations.

In the medium term many works have to be explored. We will intend incorporating these patterns defined in [2] in the PRISMA ADL. Moreover, one of our objectives is to be able to have a graphical notation in UML for the distribution properties of PRISMA ADL. Furthermore, the implementation of a model compiler is necessary to be able to automatically generate distributed applications from the ADL.

References

[1] Ahlbrecht, P., Eckstein, S. and Neumann, K. Language Constructs for Conceptual Modelling of Mobile Object Systems*: In Proc. 4th Int. Symp. on Collaborative Technologies and Systems, Simulation Series*, Orlando, USA,2003, p 121-126.

[2] Ali, N.H., Silva J., Jaen, J.,Ramos I., Carsi, J.A., and Perez J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proceedings of 15th IASTED , Parallel and Distributed Systems, Acta Press(Marina del Rey, C.A.,USA, November 2003), p 820-826.

[3] Andrade, L.,and Fiadeiro, J., "Interconnecting Objects via Contracts". OOPSLA'99. Tutorial 56.

[4] Aspect-Oriented Software Development, http://aosd.net

[5] Canal, C. *A Language for the Specification and Validation of Software Architectures*, (In Spanish) P.H.D Thesis, Malaga, Spain, 2000.

[6] CORBA Official Web Site of the OMG Group: http://www.corba.org/

[7] Correas, J. and Bueno, F. A Configuration Framework to Develop and Deploy Distributed Logic Applications: *ICLP01 Colloquium on Implementation of Constraint and Logic Programming Systems*, Cyprus, 2001.

[8] Garlan, D. "Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events." Third International School on Formal Methods 2003, LNCS, Bertinoro, Italy, September 2003, pp 1-24.

[9] Herrero, J.L. *Proposal of a Platform, Language and Design, for the Development of Aspect Oriented Applications.* (In Spanish)P.H.D. Thesis, Extremadura, Spain, 2003.

[10] Jaén, J. and Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures: *23rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with ICDCS2003*, Providence, Rhode Island, USA, 2003, p 138-146.

[11] JAVA Remote Method Invocation (RMI) Specification of SUN Microsistems: http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC html

[12] Kramer, J., Magee J. and Sloman, M. Configuring distributed systems: *In Proceedings of the 5th ACM SIGOPS European Workshop*, Mont St Michel,September 1992.

[13] Letelier, P., Sánchez, P., Ramos I., and Pastor,O. OASIS 3.0, A formal focus for the object oriented conceptual modelling(In Spanish)(Universidad Politécnica de Valencia, SPUPV-98.4011, ISBN 84-7721-663-0, 1998).

[14] Lopes, A., Fiadeiro J.L. and Wermelinger, M. "*Architectural Primitives for Distribution and Mobility*", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002: 41-50.

[15] Magee, J., Dulay, N., Eisenbach S. and Krammer, J. Specifying Distributed Software Architectures: *Proc. of the 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, 1995, 137-153.

[16] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures: *3rd IEEE International Conference on Quality Software (QSIC 2003)*, Dallas, Texas, USA, November 2003, p 59-66.

[17] Pérez, J., Ramos, I., Lorenzo, A., Letelier, P. and Jaén, J. PRISMA: OASIS Platform for Architectural Models(In Spanish), In Proceedings of VII Jornadas de Ingeniería del Software y Bases de Datos, JISBD,(Escorial Madrid, Spain, 2002), 349-360.

[18] Soares, S. and Borba, P. PaDA: A Pattern for Distribution Aspects: *In Second Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, 2002, p 87-99.

[19] Soares, S., Laureano, E. and Borba, P. Implementing Distribution and Persistence Aspects with AspectJ: *Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02*, Seattle, WA, USA,2002, p 174-190

[20] Szyperski, C., *Component software: beyond object-oriented programming*, (New York, USA: ACM Press and Addison Wesley, 1998).

Analysis of a Distribution Dimension for PRISMA

Nour Ali, José Ángel Carsi, Isidro Ramos

Polytechnic University of Valencia
Department of Information Systems and Computation
Camino de Vera s/n
E-46021 Valencia – España
{nourali, pcarsi, iramos}@dsic.upv.es

Abstract. In this paper, we analyze the introduction of a distribution dimension to an architectural model called PRISMA. PRISMA describes software architectures integrating the aspect-oriented software development (AOSD) and the component-based software development (CBSD). One of the main objectives of PRISMA is to properly model distributed systems. Therefore, this work proposes different approaches in adding the distribution dimension, evaluating each one of the approaches to some quality attributes such as reusability, mobility, distributed communication and flexibility in order to come to a conclusion for a distribution model. The final conclusion achieved at the end is an intermediate between the AOSD and CBSD making architectural elements aware of their distribution by adding a distribution aspect to the rest of its aspects and making the architectural model location aware by adding location information to attachments which are the entities that glue components and connectors of an architectural model.
Keywords: distribution, mobility, software architectures, AOSD, architecture description languages, reusability.

1 Introduction

Although many technologies have emerged in dealing with distribution issues at an implementation level, there are few approaches that have dealt with the space of the problems of distribution at a high abstraction level. Thus, considering distribution at analysis and design phases of software development high quality distributed applications are generated supporting the traceability between phases of the life cycle minimizing time, costs and efforts in the development process.

As a consequence of the poor capability of the object-oriented model to describe complex structures of information systems, the Component-Based Software Development (CBSD) and the Aspect-Oriented Software Development (AOSD) have emerged. The CBSD [5] promises to control the complexity of system construction by coupling entities that provide specific services. The AOSD [3] allows separation of concerns by modularizing crosscutting concerns in a separate entity: the aspect. Aspects can be reused and manipulated independently of the rest of properties of the system.

Our aim is to specify distribution and mobile properties at a high abstraction level using combining the AOSD and the CBSD through PRISMA [10]. PRISMA defines software architectures integrating the CBSD and AOSD in order to take advantage of

Ali, N., Carsí, J. A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2004.

both approaches. The integration is done by defining types of software architectures via a set of aspects. These aspects can evolve and be reused independently of the rest of aspects. In addition, PRISMA has an associated Architectural Description Language (ADL) which is separated at a type level and at a configuration level.

This work proposes different approaches for adding the distribution dimension, evaluating each one of the approaches to some quality attributes such as reusability, mobility, expressiveness and distributed communication in order to come to a conclusion for a distributed model. This model should preserve the characteristics of PRISMA as reusability and flexibility and respect the concepts and definitions of each of its first order citizens which are defined basing on the AOSD and CBSD.

The paper is organized as follows. Section 2 compares some related work to ours that can be found in the literature. In section 3, we present the PRISMA architectural model concluding the possible entities to which we can add to them distribution. In section 4, the different approaches for adding the distribution dimension to PRISMA are discussed evaluating each approach to achieve its drawbacks and benefits coming to a final proposal for a distribution model. Finally, section 5 draws some conclusions and highlights some future work.

2 Related Works

The CBSD arose in the early 90´s to simplify the construction of distributed systems. Since then a great number of architecture description languages (ADL) emerged such as Acme [6] and LEDA [4], nevertheless most of them do not support the constructs for describing distributed and mobile systems.

The works on an architecture level that have supported distribution is the ADL Darwin [9]. Darwin uses Pi Calculus to define the semantics of distributed message-passing. However, this language does not support the mobility of the components and has not been updated to support the new distribution technologies.

Recent works done in supporting distribution and mobility are the works in [8] which describe the semantics in externalising a distribution dimension. This distribution dimension is very similar to a connector but instead of containing the business logic, it controls the rules for mobility and locations achieving a separation between computation, coordination and distribution. Our approach is quite similar, but we use the aspect oriented approach [3] which encapsulates distribution issues of components in entities called aspects.

The rise of aspect-oriented programming languages introduced the encapsulation of the distribution code in the distribution aspect separating it from the rest of the code. This is done by extending known programming languages as AspectJ [11]in the case of JAVA. Whilst an inconvenient of these low level languages is the definition of the pointcuts inside the aspects decreasing reusability. Therefore, to solve these issues the aspect-oriented software development approach emerged. It incorporates aspect concepts at early stages of development, as in the design phase, and supports code generation to AspectJ. In works as in [7] distribution issues as load balancing, caching, migration are not contemplated. Moreover, it does not use a general purpose language (i.e. the language used to specify the distribution behaviour is different from the one used to specify the functional behaviour of the architecture).

3 PRISMA Architectural Model

PRISMA is an architectural model [10] that combines the AOSD and CBSD to describe software architectures. The architectural elements of the software architectures are described by weaving a set of aspects such as the functional, coordination and quality. The basic architectural elements are:

Components: They are the architectural elements that hold the functionalities of the architectural model offering and requesting services through ports (which are typed with interfaces that contain services). These architectural elements are unaware of their environment (connectors, other client and server components).

Connectors: They are the architectural elements that synchronize the services path through their roles (which are typed with interfaces that contain services). However, they are also unaware of the components they synchronize their services.

The architectural elements have been conceptualized in this manner in order to maximize the reusability as high as possible not only for components but also for connectors. Therefore, the components and connectors are unaware of each other.

In order to form architectural models the components and connectors are instantiated using the configuration language and then attached using entities called *Attachments*. The attachments are defined by attaching a port of a component with a role of a connector.

The entities we consider for adding the distribution dimension without affecting on the PRISMA model are the attachments and architectural elements. As attachments are aware of the architectural elements they attach, additional location and distribution information has only to be added to them. The architectural element has its distribution properties encapsulated in a distribution aspect which is added to its set of aspects.

4 Analyses of the Different Distribution Approaches

In this section, we will present the different approaches in which PRISMA can add its distribution dimension to obtain a distribution model. In each of the approaches we will use a simple bank system example to go through the benefits and drawbacks of each approach. The example is the following (see Figure 1):

An administrator of a bank system is a mobile user. She/he determines where to move her/himself i.e. she/he controls from where to access to the bank system. Whereas, the administrator can also move a bank account to her/his location. Therefore, the mobility of the account can be controlled by the environment. As well, when the account detects that its principal bank office has changed, the account moves itself to the location of this bank office (i.e. the component moves due to a change in its state).



Fig. 1. Part of the Bank System architectural model where the lines between the architectural elements represent attachments.

4.1 Case 1: Components and Connectors Unaware of their Distribution.

This approach proposes that the architectural elements are unaware of their location and are not in control of their mobility. To do so, the entities responsible of the distribution properties are the *attachments* (see Table 1). The attachments should be aware of the locations of the architectural elements they connect, have the mobility services and restrict the locations of the architectural elements they connect.

Table 1. Instances of components and connectors are unaware of their location and Attachments are the enties that are location aware of the instances they attach.

Location / ArchE	ComponentInstance	ConnectorInstance
Component	----	----
Connector	----	----
Attachment	X	X

```
01  Administrator.Port<>Connector.Role2;
02      Administrator.Location=X;
03      Administrator.Constraints
04          always{Location< x}
05      Administrator.Services
06          move(newLoc:LOC);
             Valuations
               Location:=newLoc;
07      Connector.Location=Y;
08        Connector.Constraints
09          always{Location>0 & Location< x}
10      Connector.Services
11          move(newLoc:LOC)
             Valuations
               Location:=newLoc;

12  Connector.Role1<>Account.Port
13      Connector.Location=Y;
14        Connector.Constraints
15          always{Location>0 & Location< x}
16      Connector.Services
          move(newLoc:LOC);
           Valuations
17           Location:=newLoc;
        Account.Location=Z;...
```

Fig. 2. The PRISMA like specification of the attachments in Case 1.

The specification of the attachments for the example for this approach would look as in Figure 2. Each attachment is specified by identifying which port of an instance of a component is attached with which role of an instance of a connector (see lines 01 and 12). Then for each attachment, the distribution properties of its participants (component or connector instances) are specified. For example, for the attachment between the administrator and connector, the distribution properties of the administrator and the distribution properties of the connector are specified (see lines 02-11). Let's go through the following properties of this approach:

- Reusability: The distribution properties as noticed are tangled in the specification of the attachments. A mechanism that provides reusability should be applied. PRISMA uses the aspect-oriented approach in which we could use distribution aspects to encapsulate the properties, however, an aspect should be weaved with a functionality that attachments neglect. This

is why in the specification, for each of the attachments of the connector the same distribution properties are repeated.

- Mobility: The mobility of each architectural element has to go through the attachments of that architectural element. For example, when the Administrator decides to move an account to its location the move service of the Account should be invoked in all its associated attachments. In order to do so the Account cannot be attached to other components by a different connector of the Account. Another limitation is that there cannot be any constraint that the environment of a component cannot affect on its mobility. This is due to the fact that for the attachments all architectural elements are the environment. Therefore, they cannot differentiate between the petitions of mobility that come from the environment and the petition for mobility that occur due to the change of state of an architectural element.
- Expressiveness: The expressiveness of mobility is very complicated. This is due to the fact that all requested mobility services will pass through the attachments. In this case, there should be a mechanism to differentiate between the distribution services requested to the attachments and the services requested to the architectural elements that pass through the attachments.
- Distributed Communication: The communication is very appropriate. This is because the attachments are aware of the locations of each of the components and connectors they attach. In this way, when the attachments receive the service of an interface (port or role), they know to whom to send it and to where. This approximation could be very similar to a name service in which the names of the interfaces and there references are stored.

Going through the previous points we can conclude that this approach has a good contribution in distribution communication. However, the reusability which is one of the goals that PRISMA achieves is lost. In addition, the mobility of the architectural element and its expressiveness is somehow complicated. Therefore, we need to solve them in some manner.

4.2 Case 2: Components and Connectors Aware of their Distribution.

In this section, we are going to analyze the situation where the architectural elements are aware of their location and have the authority to control it (see table 2). In PRISMA the properties of architectural elements are encapsulated in aspects which are then weaved with each other. Therefore, in this approach the distribution properties of each architectural element are described in a distribution aspect that is added to them.

Table 2. Instances of components and connectors are the only entities of the architecture aware of their locations through their distribution aspect.

Location / ArchE	ComponentInstance	ConnectorInstance
Component	X	----
Connector	----	X
Attachment	----	----

This approach is used in the definition of the distribution aspects in [1] and contains the full example specified in this approach. Figure 3, shows the specification of the Account at a type level. The distribution properties are specified including a location attribute at the type definition level. Then the distribution aspect is weaved with the other aspects of the architectural element in the Component definition by specifying the synchronizations of the services between the aspects. Then in the configuration level when the architectural elements are instantiated the location of each instance is given a value.

```
Distribution Aspect Mbile using
Mblty;
Attributes
 Location: LOC NOT NULL;
 X: LOC Not null;
Constraints
 always{ Location>0 & Location< x} ;
Services
 assignLocation(L:LOC)
    Valuations
           Location:=L;
 move(out                  Location);
End Distribution Aspect Mbile;
```

```
Component_type Account
Ports
    Moves_Acc: Mblty;
End_Ports;
Functional Aspect
    Import BankFunctionality;
   Weaving
      Import Distribution Aspect
: strangerAffectsMyMobiility
      move(LOC)              after
Change_PBankOffice;
  End_Weaving;
End_Component;
```

Fig. 3. Specification of the Account component at the type definition language.

Analyzing this approach the following can be noticed(see Figure 3):

- Reusability: The distribution properties are encapsulated in a distribution aspect which can be specified independently of the architectural elements they will be added to. Thus, different architectural elements with same distribution behaviour can use the same distribution aspect.
- Mobility: The mobility of each architectural element can be controlled by the architectural element. This allows to restrict e.g. that the Administrator is in control of its own locations and that the environment cannot affect its locality. In addition, in this way an Account can specify that both the environment (the Administrator can move it) and its mobility can depend on its state.
- Expressiveness: The expressiveness of mobility is expressed as an additional service of the architectural element. The architectural element can specify that it offers its service to the environment by offering it in a port.
- Distributed Communication: As the architectural elements are unaware of the other architectural elements of the architecture, the locations of the destinations of the messages are unknown. This lacks the availability to model a distributed communication.

Table 3 resumes the different properties of each approach. The inserting of the distribution dimension in an additional aspect of the architectural elements is quite a tidy approach because it follows the PRISMA characteristics and uses the aspect-oriented approach to define a distribution model. However, the specification of a distributed communication in the PRISMA model gives additional functionality in representing the technologies of middleware platforms such as CORBA.

Table 3. The contribution of each approach on the PRISMA model.

Properties / Approach	Reusability	Mobility	Expressiveness	Distributed Communication
1	X	X	X	√
2	√	√	√	X

As a conclusion, we can make a combination of the two previous approaches as can be seen in Table 4. We can preserve the idea of adding a distribution aspect with the distribution properties that have control over the mobility and as well add location information of the instances of the architectural elements to the attachments. In this way we can have the benefits of both approaches. We achieve an adequate distribution model represented at the type definition language with the distribution aspects as in Figure 3 and the following specification at the configuration level:

```
Architectural Model
Administrator: Admin(Location=X);
Connector: Con( Location=Y);
Account: Acc( Location=Z);
Attachments
(Admin.Location, Admin.Port)←→(Con.Location, Con.Role1);
(Con.Location, Con.Role2)←→(Acc.Location, Acc.Port)
End_Architectural Model
```

Table 4. Components and Connectors are aware and control their locations. In addition the Attachments are location aware of the instances they attach.

Location / ArchE	ComponentInstance	ConnectorInstance
Component	X	----
Connector	----	X
Attachment	X	X

The previous specification shows that when the architectural elements are instantiated, a value to their locations is given. Then the location of each instance is registered in the attachments. When an instance changes its location, a notification has to be made to its attachment in order to change the location values. This service is provided by the PRISMA meta-level.

5 Conclusions and Further Work

In this paper, we have made an analysis to achieve an appropriate distribution model for PRISMA. Different approaches for adding the distribution dimension have been gone through to find their drawbacks and benefits. The final conclusion is to add a distribution aspect to the architectural elements in order to control the dynamic locations. In addition, to make the architectural model location aware we added location information to attachments which are the entities that glue components and connectors of an architectural model.

In future woks we will keep studying the conclusion approach in more detail incorporating meta-level information in order to notify the attachments of a change in the locations of the architectural elements.
In addition, we will incorporate patterns defined in [2] in the PRISMA ADL.

Moreover, one of our objectives is to be able to have a graphical notation in UML for the distribution properties of PRISMA ADL. Furthermore, we are working on the implementation of a model compiler to be able to automatically generate distributed applications from the ADL.

References

[1] Ali, N.H., Perez J., Ramos I. "High Level Specification of Distributed and Mobile Information Systems", Proceedings of Second International Symposium on Innovation in Information & Communication Technology ISIICT 2004, Amman, Jordan, 21-22 April, 2004.

[2] Ali, N.H., Silva J., Jaen, J.,Ramos I., Carsi, J.A., and Perez J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proceedings of 15th IASTED , Parallel and Distributed Systems, Acta Press(Marina del Rey, C.A.,USA, November 2003), p 820-826.

[3] Aspect-Oriented Software Development, http://aosd net

[4] Canal, C. A Language for the Specification and Validation of Software Architectures, (In Spanish) P.H.D Dissertation, Malaga, Spain, 2000.

[5] CORBA Official Web Site of the OMG Group: http://www.corba.org/

[6] Garlan, D. "Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events." Third International School on Formal Methods 2003, LNCS, Bertinoro, Italy, September 2003, p 1-24.

[7] Herrero, J.L. *Proposal of a Platform, Language and Design, for the Development of Aspect Oriented Applications*. (In Spanish)P.H.D. Thesis, Extremadura, Spain, 2003.

[8] Lopes, A., Fiadeiro J.L. and Wermelinger, M. "Architectural Primitives for Distribution and Mobility", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, p 41-50.

[9] Magee, J., Dulay, N., Eisenbach S. and Krammer, J. Specifying Distributed Software Architectures: *Proc. of the 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, 1995, 137-153.

[10] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures: 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 2003, p 59-66.

[11] Soares, S., Laureano, E. and Borba, P. Implementing Distribution and Persistence Aspects with AspectJ: *Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02*, Seattle, WA, USA,2002, p 174-190

[12] Szyperski, C., *Component software: beyond object-oriented programming*, (New York, USA: ACM Press and Addison Wesley, 2002)

Especificación de requisitos software basada en características de calidad, separación de *concerns* y orientación a objetivos*

Elena Navarro[1], Patricio Letelier[2], Isidro Ramos[2] y Bárbara Álvarez[3]

[1]Departamento de Informática, UCLM, Albacete
enavarro@info-ab.uclm.es
[2]Departamento de Sistemas de Información y Computación, UPV, Valencia
{letelier, iramos}@dsic.upv.es
[3]Departamento Tecnologías de la Información y Comunicaciones, UPCT, Cartagena
Barbara.Alvarez@upct.es

Aunque existe consenso respecto al contenido de la SRS, no sucede lo mismo con su organización. Es evidente que de acuerdo con el sistema, la audiencia y la actividad realizada con ella, la SRS debería ser presentada según diversos criterios para su estructuración. Uno de los principales problemas a resolver es el *crosscutting* (entrelazado) de requisitos, el cual tiende a producir especificaciones enmarañadas. En este trabajo se aborda la problemática de la organización de la SRS aplicando técnicas de orientación a aspectos para resolver dicho *crosscutting*. Además, hacemos uso de la orientación a objetivos para establecer una adecuada trazabilidad desde objetivos del software hacia requisitos específicos, y de éstos a operacionalizaciones realizadas por componentes software. El estándar ISO/IEC 9126 se establece como framework instanciable en la estructuración de objetivos y requisitos. La propuesta presentada en este trabajo se enmarca en ATRIUM, una metodología para definición concurrente de requisitos y arquitecturas software.
Keywords: Ingeniería de Requisitos Orientada a Objetivos, DSOA, Arquitectura Software, Modelos de Calidad

1. Introducción

La organización de la *Especificación de Requisitos Software* (SRS) en sistemas grandes y/o complejos presenta una serie de desafíos sin solución satisfactoria hasta el momento. Dicha organización es clave para asegurar la calidad de la SRS, respecto características tales como: corrección, completitud, consistencia y facilidad de modificación. Su organización también es crucial para la eficaz explotación de la SRS en la elaboración de otros artefactos derivados o relacionados. Las recomendaciones del estándar IEEE 830-1998 [9] son un punto de referencia en cuanto al contenido que debe estar presente en una SRS. En dicho estándar se incluyen algunas sugerencias

* Este trabajo ha sido subvencionado por el proyecto CICYT DYNAMICA (TIC2003-07804-C05-01 y TIC2003-07804-C05-02).

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004). 2004.

respecto a posibles organizaciones para los requisitos específicos del software. Sin embargo, en la práctica, cuando el número de requisitos y/o la complejidad es considerable, dichas sugerencias no son suficientes.

Se pueden encontrar en la literatura diferentes aproximaciones para la especificación de requisitos. El enfoque *Orientado a Objetivos* para Ingeniería de Requisitos [10] es una de ellas, donde un conjunto inicial de objetivos se descompone en subobjetivos y finalmente en requisitos específicos asignables a agentes software. Ésta aproximación permite tratar adecuadamente la separación de *concerns* (propiedades o áreas de interés) de un sistema con el propósito de soportar evolución, adaptabilidad, comprensibilidad, etc. C*oncerns* que pueden variar desde características no funcionales, como la seguridad o la tolerancia a fallos, hasta características funcionales. El problema que plantea esta aproximación es que no proporciona mecanismos para manipular aquellos intereses que aparecen dispersos en el sistema, cruzando o cortando diferentes elementos del mismo (fenómeno referido como *crosscutting*). En cambio, la *Ingeniería de Requisitos Orientada a Aspectos* (AORE) [14] permite gestionar ese *crosscutting* mediante la utilización del mecanismo denominado *weaving*.

El objetivo de este trabajo es presentar nuestra propuesta para la organización de requisitos del software, que integra orientación a objetivos y orientación a aspectos y utiliza el estándar ISO/IEC 9126 [8] como framework para la selección de *concerns*. Además, el proceso descrito en este trabajo representa la primera actividad de ATRIUM [12], una metodología que guía al analista, a lo largo de un proceso iterativo, desde un conjunto de necesidades de usuario/sistema hasta la instanciación de la arquitectura, expresada mediante PRISMA [13].

El contenido del resto de trabajo está organizado como se describe a continuación. La sección 2 introduce el estándar ISO/IEC 9126 y cómo se utiliza en nuestra propuesta. La sección 3 presenta en detalle el conjunto de tareas que conducen a la definición del modelo de objetivos. El caso de estudio al que se ha aplicado ese conjunto de tareas se muestra en la sección 5. Finalmente, las secciones 6 y 7 hacen una descripción de los trabajos relacionados y las conclusiones obtenidas.

2. ISO/IEC 9126 para la selección e identificación de *concerns*

El estándar ISO/IEC 9126 es un modelo que define las características de calidad del producto software. Por ello puede ser empleado como framework inicial en la captura y organización de objetivos y requisitos, es decir, como taxonomía de *concerns*. De esta forma, a medida que se vayan recolectando las necesidades informales del software, éstas pueden ser analizadas, desglosadas y organizadas. Así, se puede gestionar el *croscutting* de la especificación, reduciendo o eliminando otros inconvenientes tales como: redundancia, inconsistencias, etc.

La taxonomía definida por este estándar va más allá de la tradicional clasificación de requisitos funcionales y no funcionales (que poco contribuye a la organización de los requisitos). De hecho, gran parte de los llamados requisitos funcionales encuadrarían en la sub-característica “idoneidad”. Aunque el ISO/IEC 9126 nos ofrece un conjunto

bastante amplio de concerns, estos podrían ampliarse si fuera necesario. En este caso, al aplicar nuestra propuesta emergen las siguientes alternativas:

a. Considerar una nueva dimensión, para organizar objetivos/requisitos, adicional a la taxonomía propuesta por el ISO/IEC 9126. Esta opción resulta interesante cuando las características adicionales son tan relevantes como las ya consideradas y el *croscutting* con ellas pudiese ser alto.
b. Incluir una nueva característica/sub-característica extendiendo la taxonomía. Esta alternativa sería recomendable cuando no se trata de aspectos tan relevantes como los ya considerados y/o no se espera que pudiesen generar demasiado *crosscutting*.
c. Tratar dicho elemento como un atributo del objetivo/requisito. Esta opción se sugiere cuando se trata de aspectos no tan relevantes (no se trata precisamente de objetivos ni de requisitos) o que no involucrarían considerable *crosscutting*, pero que resultan interesantes sobre todo de cara a agrupaciones necesarias para presentar los objetivos/requisitos.

El IEEE 830-1998 ofrece algunos criterios y pautas para la organización de los requisitos específicos, reconociendo que no existe una organización óptima que se ajuste a todos los sistemas. Entre los criterios de organización mencionados están: modo de operación del sistema, clase de usuario, entidades del problema, etc. Así, tal como lo recomienda la alternativa c), en lugar de extender la taxonomía, dichos elementos pueden tratarse como atributos de los objetivos/requisitos. De esta forma, es posible ofrecer una vista basada en agrupaciones a través de dichos elementos, aunque no formen parte de la taxonomía inicial. Por ejemplo, podríamos tratar la sección "requisitos lógicos de la base de datos" (incluida en la IEEE 830-1998) como una dimensión adicional (denominada *data*).

3. Modelo de objetivos y *concerns* del sistema

Tanto la identificación como la especificación de los diferentes *concerns* y el posible *crosscuting*, se plantean en este trabajo mediante la definición de un conjunto de tareas (Fig. 1) que conforman la actividad 1 de la metodología ATRIUM. El Modelo de Objetivos [11], generado mediante la realización de esta actividad permite tanto la especificación de *objetivos*, *requisitos* y *operacionalizaciones*, como de los diferentes *croscuttings* que puedan aparecer. En la construcción del Modelo de Objetivos, el estándar ISO/IEC 9126, descrito en la sección anterior, se emplea como framework instanciable, sirviendo al analista como una descripción inicial del conjunto de *concerns* del sistema. La Fig. 1 establece el flujo de trabajo para la elaboración del Modelo de Objetivos junto con los artefactos de entrada necesarios para su realización. Aunque ésta muestra un flujo secuencial de aplicación de las tareas, en la práctica es iterativo. Ello es debido a que las distintas tareas que lo integran, se ejecutan con cada nuevo objetivo que se identifica o selecciona.

La primera de estas tareas trata la *Identificación/Selección de los Objetivos*. Como podemos observar una entrada de esta tarea es el modelo de calidad ISO/IEC 9126. Este modelo proporciona una vista inicial de aquellos *concerns* que podrían ser significativos para el sistema. De manera que el analista, iterativamente, seleccionará aquél que considere apropiado e iniciará a partir de él su especificación y

refinamiento. A medida que se realiza la captura de requisitos puede ser conveniente incluir nuevos *concerns* para incluir adecuadamente objetivos/requisitos adicionales.

Las *Necesidades de Usuario/Sistema* constituyen otra de las entradas de esta tarea. Éstas nos ofrecen información para identificar nuevos objetivos/requisitos. Cada uno de los objetivos establecidos, mediante este proceso, quedará enmarcado dentro de la especificación de *concerns* establecida la ISO/IEC 9126, mediante una estructura de grafo. Esto nos permitirá conseguir dos mejoras, por un lado facilitar la legibilidad de la especificación, y por otro, conducir el proceso de captura y análisis de requisitos.

En la tarea de *Especificación del Objetivo* se establecen no sólo los atributos que lo conforman, como son su nombre, prioridad, etc., sino también cuáles son las relaciones de composición (AND/OR/XOR) con sus objetivos padres ([11] ofrece una descripción completa del modelo de objetivos).

Para cada objetivo, no sólo se especifican las relaciones de refinamiento, sino también aquellas relaciones de *weaving* necesarias para gestionar el *crosscutting* que aparece en la especificación. Estas relaciones pueden ser estereotipadas de acuerdo a la semántica que tradicionalmente se ha dado en AOSD, es decir, *before* y *after*. Esto nos permite expresar, por ejemplo, cómo la satisfacción de un objetivo/requisito queda supeditada a la satisfacción previa (posterior) de otro requisito/objetivo que ha establecido una relación *before* (*after*) con él. Aunque en esta propuesta se ofrece este conjunto de relaciones no se descarta el uso de otras adicionales.

Este proceso de refinamiento de objetivos, continúa hasta que el objetivo es asignable a un agente del sistema. En este momento, pasamos de un refinamiento intencional a un refinamiento operacional y con ello a la especificación de un requisito. Una diferencia importante entre un requisito y un objetivo estriba en que para los requisitos se exige que puedan ser verificables. Así, para la *Identificación/Especificación de requisitos*, se sigue un proceso similar al de los objetivos, mediante la definición de sus atributos y relaciones de composición. También, se establecen aquellas relaciones de *weaving* que sean aplicables.

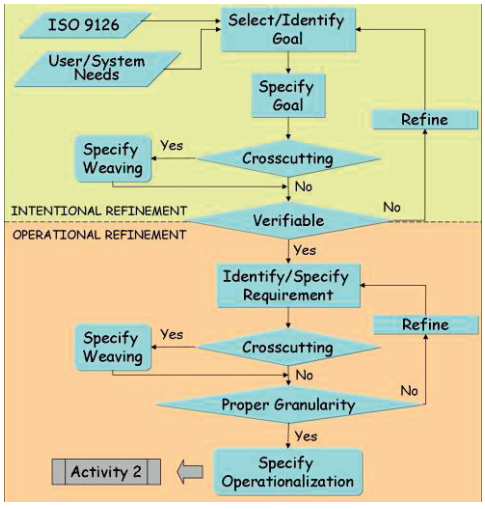


Fig. 1. Flujo de trabajo para la definición de objetivos/requisitos

Tras la especificación de requisitos, la siguiente tarea es la *Especificación de la operacionalización*, es decir, la definición del agente o conjunto de agentes que colaboran en su realización. En la definición del modelo de objetivos, la operacionalización es únicamente una descripción de la solución propuesta para la realización, actuando ésta como entrada de la actividad 2. Esta última es la encargada, mediante la descripción del escenario correspondiente, de elaborar una definición completa de la solución. La introducción de esa descripción en el Modelo de Objetivos nos permite incorporar las relaciones entre esa solución y los requisitos ya definidos en el modelo. Así, podemos reflejar cómo una solución puede colaborar positivamente en la realización de un requisito y negativamente en la realización de otros, consiguiendo con ello un análisis más exhaustivo del conjunto de soluciones.

4. Caso de Estudio

Esta sección ilustra cómo hemos aplicado nuestra propuesta en el contexto de un sistema real. Esta experiencia fue desarrollada gracias a la colaboración en el proyecto europeo *Environmental Friendly and cost-effective Technology for Coating Removal* (EFTCoR) [4] cuyo objetivo fundamental está relacionado con las operaciones de mantenimiento de cascos de buques. Principalmente, trata las operaciones de eliminación de revestimiento, limpieza y re-pintura del casco por una familia de robots que realizan diferentes operaciones o la misma pero de distinta manera.

La plataforma de teleoperación identificada [7] está integrada por diferentes subsistemas centrándose nuestro caso de estudio en la especificación de la *Unidad de Control de los Dispositivos Robóticos* (RDCU). Ésta tiene que interaccionar con el resto de subsistemas con el fin de obtener la información necesaria para controlar los diferentes dispositivos (Sistemas de Posicionamiento y Herramientas de Limpieza) empleados en las tareas de mantenimiento, conforme a las operaciones introducidas por el operador. La definición arquitectónica de la RDCU es especialmente compleja debido a las restricciones que han de satisfacerse para permitir un comportamiento dinámico del sistema final. En tiempo de ejecución, cada herramienta de limpieza o dispositivo de posicionamiento podría ser reemplazado. Además, tanto el cambio de herramientas como su operación tienen severos requisitos de seguridad, proporcionando mecanismos para la parada si se prevé algún daño al operador o al equipamiento. Asimismo, cada operación de mantenimiento ha de ser programada para su realización dentro de unos límites temporales concretos. A continuación se detallan algunos fragmentos de las necesidades de usuarios (descritas en [4]) analizados durante la elaboración del modelo de objetivos:

Positioning systems and tool can work simultaneously. The RDCU is responsible of co-ordinating their actions according to their operational states, the mission parameters, and the current state of the environment.
(1) The operational commands for positioning systems and activating the tool can be easily and efficiently issued to the system. *(2)The system should react to such commands efficiently. In some cases, the execution time of the commands should be smaller than a given deadline*
(1)The possibility of using different coating removal technologies (blasting, water pressure). Though the chosen technology for the EFTCoR project is blasting, the RDCU should be open to incorporate cleaning tools based on other technologies. *(2) The possibility of using the system for different maintenance tasks, including at least the fresh water washing before blasting and the painting after blasting.* *(3) The possibility of using different positioning systems and different combinations of primary and secondary positioning systems.* *(4) The possibility of using different tools for the same or different processes (already considered in the (1))*

A partir de las sentencias anteriores podemos apreciar la existencia del *croscutting* en la especificación, como es el caso de los objetivos relacionados con *Eficiency* y *Adaptability*. Ambos se aplican sobre otros objetivos como son el *ControlPositioning* o *ControlTools*. La aplicación de la tarea de *Identificación/Selección de Objetivos* usando como entrada los fragmentos anteriores, generó como resultado la lista de objetivos que aparecen en la Tabla 2 (ver [11] para más detalles sobre sus atributos):

El grafo de la Fig. 2 muestra (parte de) el Modelo de Objetivos donde se revisa el refinamiento de los objetivos de la Tabla 1. Así, se observa como *Portability*, *Functionality*, y *Efficiency* son algunas de las características seleccionadas como *concerns* para el sistema EFTCoR. También dicha figura nos muestra algunas de las relaciones de refinamiento que se establecieron. Por ejemplo, la relación AND entre el objetivo *AdaptabilityWorkingEnvironment* y *AdaptabilityHullMaintenace-Operation* introducida para conseguir la satisfacción de *Adaptability*. Por otra parte, para el objetivo *AdaptabilityWorkingEnvironment* se ha establecido una relación de *weaving* con el objetivo *ControlPossitioning*.

Tabla 1. Descripción (parcial) de Objetivos para EFTCoR

GOAL	DESCRIPTION
Functionality	The system has to provide functions which meet stated and implied needs
Suitability	The system has to provide an appropriate set of functions for specified tasks and user objectives
CoordinateDevices	The RDCU has to coordinate robotic devices according to the current mission procedure and global system state
ControlPossitioning	The RDCU has to control the positioning devices
ControlTools	The RDCU has to control the tools attached to positioning devices (blasting head, painting tool, etc)
Efficiency	The system should use efficiently their resources
TimeBehaviour	The system should respond with appropriate speed
PerformanceOperation	The system should respond with appropriate speed to the operation commands
Portability	The system should be transferable from one environment to another
Replaceability	The system should be able to use different software product for the same purpose
Adaptability	The systems has to be adaptable for different specified environments
AdaptabilityWorkingEnvironments	The system operation has to be adaptable to different working environments
AdaptabilityHullMaintenanceOperation	The system operation has to be adaptable to different hull maintenance operations

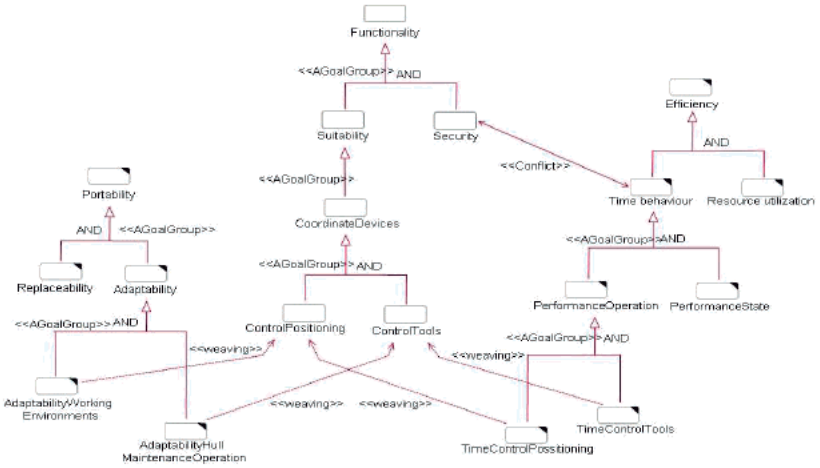


Fig. 2 (Vista parcial del) Modelo de Objetivos para EFTCoR

5. Trabajos Relacionados

Existen numerosos trabajos relacionados con el tratamiento de aspectos en distintos niveles de abstracción, tanto de implementación como de diseño. En el nivel de requisitos, las propuestas se han focalizado en cómo identificar y especificar cuáles serán los *concerns* del sistema. Pero no solamente esto, sino también en cómo determinar cuáles de estos intereses serán realizados como aspectos, en posteriores artefactos, de manera que se cumpla la propiedad de cierre [5].

Rashid et al [14] han propuesto un modelo para ingeniería de requisitos que identifica los *concerns* en una etapa temprana del desarrollo, juntamente con los requisitos. Este modelo permite relacionar y establecer la influencia de los aspectos tempranos (*early aspects*) sobre posteriores etapas del desarrollo. Sin embargo, esta propuesta no permite reconstruir la especificación para ofrecer una mejor comprensión al usuario final en la validación de la misma.

La propuesta de Brito et al [2] es con la que se presentan más puntos en común. Se plantea la utilización de un modelo de objetivos, en su caso el NFR Framework [3], para la especificación de requisitos no funcionales y otra técnica, tal como casos de uso o viewpoints para la especificación de los requisitos funcionales. Así, extiende el framework con un nuevo tipo de relación denominada *concerns requeridos* para determinar los posibles crosscuting con otros requisitos. En cambio, nuestra propuesta ofrece al analista un mismo mecanismo para la especificación de ambos tipos de requisitos facilitando con ello tanto su utilización como su comprensibilidad.

Otra importante diferencia que plantea nuestra propuesta se refiere al tratamiento uniforme de objetivos/requisitos, con relación a las categorías aplicables para su catalogación. En cambio, en otras propuestas se focaliza explícitamente en la distinción entre funcionales o no funcionales, empleando técnicas distintas para la descripción de ambos tipos de objetivos/requisitos. Ello conlleva un coste considerable en cuanto a legibilidad y mantenibilidad. Por otra parte, ninguna de las propuestas ofrece al analista un marco de trabajo inicial que le ayude en la identificación de los *concerns* del sistema.

6. Conclusiones y trabajos futuros

En este trabajo hemos presentado una propuesta para la captura y análisis de requisitos en la que se integran orientación a objetivos y a aspectos. Gracias a la Orientación a Objetivos, se mantiene una trazabilidad explícita desde los objetivos hacia los requisitos, y desde estos últimos hacia las operacionalizaciones. Por otro lado, la aproximación orientada a aspectos permite gestionar de manera elegante y efectiva el *crosscutting* que puede aparecer en la SRS cuando se entremezclan objetivos/requisitos. Definiendo relaciones de *weaving* entre los elementos de un objetivo/requisito se consigue una representación óptima, desde la cual puedan incluso presentarse los requisitos informales en su forma original si fuese necesario. Se ha definido un flujo de trabajo en el cuál se integran ambas aproximaciones, detallando una serie de tareas que ofrecen guías para la elaboración y organización de la especificación de requisitos.

Otra de las ventajas que ofrece nuestra propuesta se refiere a la utilización del ISO/IEC 9126 como punto de partida en cuanto a posibles *concerns* del sofware. Con nuestra propuesta es posible ajustarse a las recomendaciones del IEEE 830-1998 en cuanto a contenido de la SRS pero contando con significativas ventajas en cuanto a flexibilidad y versatilidad en la organización de los requisitos.

Diversos retos quedan todavía por resolver. Uno de ellos está relacionado con las relaciones de *weaving*. Concretamente, con la idoneidad del conjunto de relaciones ofrecido y su semántica. Para ello, la problemática ofrecida por el caso de estudio facilitará la verificación y validación del mismo. Especialmente, cuando las relaciones de *weaving* establecidas y los *concerns* o los requisitos identificados tengan trazabilidad hacia aspectos en artefactos arquitectónicos, siguiendo el proceso planteado por ATRIUM.

Referencias

[1] Aspect-Oriented Software Development, http://www.aosd net

[2] I. Brito, A. Moreira, "Integrating the NFR marco de trabajo in a RE model", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (collocated to 3rd. AOSD), March 22, 2004 - Lancaster, UK.

[3] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, Non-Functional Requirements in Software Engineering, Kluwer Academic Publishing, 2000

[4] EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. (GROWTH G3RD-CT-00794) European Project within the 5th Framework Program, 2003

[5] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, H. Ossher, "Discussing aspects of AOP", Communications of the ACM, Vol. 44 , Issue 10, pp. 29 – 32, October, 2001.

[7] C. Fernández, J.A. Pastor, P. Sánchez, B. Álvarez, A. Iborra: "Co-operative Robots for Hull Blasting in European Shiprepair Industry. Robotics and Automation Magazine (RAM)", special issue on Industrial Robotics Applications & Industry-Academia Cooperation in Europe. New Trends and Perspectives. (To appear in September 2004)

[8] ISO/IEC Standard 9126-1 Software Engineering- Product Quality-Part1: Quality Model, ISO Copyright Office, Geneva, June 2001

[9] IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications, en Vol. 4: Resource and Technique Standards, The Institute of Electrical and Electronics Engineers, Inc. IEEE Software Engineering Standards Collection.

[10] A. van Lamsweerde "Goal-Oriented Requirements Engineering: A Guided Tour", Invited Paper for 5th IEEE International Symposium on RE, Toronto, August, 2001, pp. 249-263.

[11] E. Navarro, I. Ramos and J. Pérez: "Goals Model-Driving Software Architecture", 2nd International Conference on Software Engineering Research, Management and Applications, May 5-8, 2004, Los Angeles, CA, USA.

[12] E. Navarro, I. Ramos and J. Pérez: "Software Requirements for Architectured Systems". Proc. 11th IEEE International Conference on Requirements Engineering, pp. 365-366, September 8-12, 2003, Monterey, CA, USA.

[13] J. Pérez, I. Ramos, J. Jaén, P. Letelier and E. Navarro, "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", Proc. 3rd IEEE International Conference on Quality Software, Dallas, Texas, USA, pp. 59-66, November 6 - 7, 2003.

[14] A. Rashid, P. Sawer, A. Moreira and J. Araújo, "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", Proc. International Conference on Requirements Engineering, pp. 199-202, September 9-13, 2002, Essen, Germany.

Orientación a Aspectos y Orientación a Objetivos: Una propuesta para su integración*

Elena Navarro[1], Patricio Letelier[2], Isidro Ramos[2] y Bárbara Álvarez[3]

[1]Departamento de Informática, UCLM, Albacete
enavarro@info-ab.uclm.es
[2]Departamento de Sistemas de Información y Computación, UPV, Valencia
{letelier, iramos}@dsic.upv.es
[3]Departamento Tecnologías de la Información y Comunicaciones, UPCT, Cartagena
Barbara.Alvarez@upct.es

1. Introducción

Existen diferentes paradigmas para tratar la separación de *concerns* (propiedades o áreas de interés) de un sistema con el propósito de facilitar evolución, adaptabilidad, comprensibilidad, etc. Estos *concerns* pueden ser características funcionales o no funcionales, como portabilidad o tolerancia a fallos. El Desarrollo de Software Orientado a Aspectos (AOSD) [1] es una de estos paradigmas. AOSD ofrece un conjunto de mecanismos que permite gestionar aquellos intereses que aparecen dispersos en el sistema, cruzando o cortando diferentes elementos del mismo (fenómeno referido como *crosscutting*). AOSD identifica estos *concerns* expresándolos como *aspectos* [5] y tratándolos de forma explícita. Recientemente diversas propuestas han surgido, tales como [13] que proponen una identificación de *aspectos candidatos*, en la etapa de captura y especificación requisitos (AORE).

Por otra parte, un enfoque que nace en la comunidad de Ingeniería de Requisitos es la Orientación a Objetivos [9] (OOR). Ésta permite, mediante un proceso de refinamiento y descomposición, pasar desde intereses u objetivos del sistema a requisitos asignables a agentes del mismo, es decir, a cualquier componente activo tales como humanos, dispositivos cualquier componente software del sistema.

El objetivo de este trabajo es presentar nuestra propuesta para la captura, especificación y organización de requisitos, integrando dos aproximaciones: AORE y OOR. Asimismo, se utiliza el estándar ISO/IEC 9126 [7] como punto de partida en el proceso de definición de objetivos. El enfoque aquí presentado es una de las actividades de ATRIUM [11], una metodología para la definición de arquitecturas software (AS) a partir de requisitos. PRISMA [12] es el modelo elegido para la especificación de AS debido a su habilidad para la generación automática de código y por sus capacidades reflexivas en tiempo de ejecución.

* Este trabajo ha sido subvencionado por el proyecto CICYT DYNAMICA (TIC2003-07804-C05-01 y TIC2003-007804-C05-01)

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Orientación a aspectos y Orientación a objetivos: una propuesta para su integración. Taller de Desarrollo de Software Orientado a Aspectos, junto a JISBD'2004. 2004.

2. Modelo de objetivos: *concerns* del sistema

El Modelo de Objetivos (MO) de ATRIUM, previamente descrito en [10], está inspirado en KAOS [2] y NFR Framework.[3]. Desde el punto de vista exclusivo de orientación a objetivos nuestro trabajo se distingue por incluir el tratamiento tanto de requisitos funcionales como no funcionales en el MO. Así, no sólo se establece la funcionalidad del sistema sino también aquellas restricciones, calidad de servicio u objetivos de desarrollo, igualmente importantes para la definición de AS. Otra diferencia de nuestra propuesta es la incorporación de técnicas de Orientación a Aspectos para gestionar el *crosscutting* que suele aparecer en una Especificación de Requisitos del Software (SRS), especialmente cuando el sistema es complejo y/o de gran envergadura.

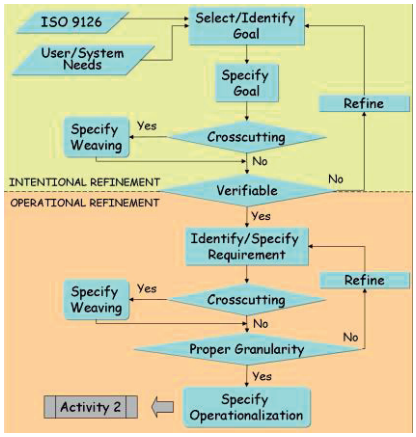


Fig. 1. Flujo de trabajo para la definición de objetivos/requisitos

La **Fig. 1** establece el flujo de trabajo para la elaboración del MO, junto con los artefactos de entrada utilizados en cada tarea. Aunque en la **Fig. 1** sólo se muestra un flujo secuencial de aplicación de las tareas, en la práctica el proceso es iterativo. Ello es debido a que las distintas tareas que lo integran, se ejecutan con cada nuevo objetivo que se identifica o selecciona.

La primera de estas tareas aborda la *Identificación/Selección de los Objetivos*. Una entrada de esta tarea es el modelo de calidad ISO/IEC 9126, el cual es utilizado como marco de trabajo de partida, ofreciendo al analista una clasificación inicial de alto nivel para los posibles *concerns* del sistema. En la medida que se realiza la captura de requisitos puede ser conveniente incluir nuevos *concerns* para incluir adecuadamente los objetivos/requisitos adicionales, o finalmente descartar aquellos no usados.

Las *Necesidades de Usuario/Sistema* constituyen otra de las entradas de esta tarea. Éstas nos ofrecen información para identificar nuevos objetivos/requisitos. Cada uno de los objetivos establecidos mediante este proceso quedará enmarcado dentro de la especificación de *concerns* establecida por el modelo ISO/IEC 9126, estableciendo una estructura de grafo. Esto nos permitirá conseguir dos mejoras, por un lado

facilitar la legibilidad de la especificación, y por otro, conducir el proceso de captura y análisis de requisitos.

En la tarea de *Especificación del Objetivo* se establecen no sólo los atributos que lo conforman, como son su nombre, prioridad, etc., sino también cuáles son las relaciones de composición (AND/OR/XOR) con sus objetivos padres.

Para cada objetivo, no sólo se especifican las relaciones de refinamiento, sino también aquellas relaciones de composición[*] necesarias para gestionar el *crosscutting* que aparece en la especificación. Este proceso de refinamiento de objetivos, continúa hasta que el objetivo es asignable a un agente del sistema. En este momento, pasamos de un refinamiento intencional a un refinamiento operacional y con ello a la especificación de un requisito. Una diferencia importante entre un requisito y un objetivo estriba en que para los requisitos se exige que puedan ser verificables. Así, para la *Identificación/Especificación de requisitos*, se sigue un proceso similar al de los objetivos, mediante la definición de sus atributos y relaciones de composición. También, se establecen aquellas relaciones de composición que sean aplicables.

Tras la especificación de requisitos la siguiente tarea es la *Especificación de la operacionalización*, es decir, la definición del agente o conjunto de agentes que colaboran en su realización. En la definición del MO la operacionalización es únicamente una descripción de la solución propuesta para la realización, actuando ésta como entrada de la siguiente actividad de la metodología, la actividad *Definición de Escenarios*. Esta última es la encargada, mediante la descripción del escenario correspondiente, de elaborar una definición completa de la solución. La introducción de esa descripción en el MO nos permite incorporar las relaciones entre esa solución y los requisitos ya definidos en el modelo. Así, podemos reflejar cómo una solución puede colaborar positivamente en la realización de un requisito y negativamente en la realización de otros, consiguiendo con ello un análisis más exhaustivo del conjunto de soluciones.

3. Caso de Estudio

Esta sección ilustra cómo hemos aplicado nuestra propuesta en el contexto de un sistema real. Esta experiencia fue desarrollada gracias a la colaboración en el proyecto europeo *Environmental Friendly and cost-effective Technology for Coating Removal* (EFTCoR) [4]. El objetivo fundamental de este proyecto está relacionado con las operaciones de mantenimiento de cascos de buques. Principalmente, trata las operaciones de eliminación de revestimiento, limpieza y re-pintura del casco del buque por una familia de robots que realizan diferentes operaciones o la misma pero de distinta manera. La plataforma de teleoperación identificada está integrada por los subsistemas descritos en [6]. Nuestro caso de estudio se centra en la especificación de requisitos de la *Unidad de Control de los Dispositivos Robóticos* (RDCU). Su

[*] En la presentación de nuestro trabajo hemos preferido utilizar el término "composición" en lugar de *weaving*, usado este último en el paradigma orientado a aspectos. En ingeniería de requisitos la semántica se acerca más a una composición de elementos que a un *weaving* determinando saltos en la ejecución.

definición arquitectónica es especialmente compleja debido a las restricciones que han de satisfacerse para permitir un comportamiento dinámico del sistema final. En tiempo de ejecución, cada herramienta de limpieza o dispositivo de posicionamiento podría ser reemplazado. Además, tanto el cambio de herramientas como la operación tienen severos requisitos de seguridad, proporcionando mecanismos para la parada si se prevé algún daño al operador o al equipamiento. Asimismo, cada operación de mantenimiento ha de ser programada para su realización dentro de unos límites temporales concretos. Algunos fragmentos de las necesidades establecidas para la RDCU descritos en [4]:

Positioning systems and tool can work simultaneously. The RDCU is responsible for co-ordinating their actions according to their operational states, the mission parameters, and the current state of the environment.
(1) The operational commands for positioning systems and activating the tool can be easily and efficiently issued to the system. *(2)The system should react to such commands efficiently. In some cases, the execution time of the commands should be smaller than a given deadline*
(1)The possibility of using different coating removal technologies (blasting, water pressure). Though the chosen technology for the EFTCoR project is blasting, the RDCU should be open to incorporate cleaning tools based on other technologies. *(2) The possibility of using the system for different maintenance tasks, including at least the fresh water washing before blasting and the painting after blasting.* *(3) The possibility of using different positioning systems and different combinations of primary and secondary positioning systems.* *(4) The possibility of using different tools for the same or different processes (already considered in the first point)*

A partir de las sentencias anteriores podemos apreciar la existencia del *croscutting* en la especificación, como es el caso de los objetivos relacionados con la *Eficiency* y *Adaptability*. Ambos se aplican sobre otros objetivos como son el *ControlPossitioning* o *ControlTools*. La aplicación de la tarea de *Identificación/Selección de* Objetivos usando como entrada los fragmentos anteriores, generó como resultado la lista de objetivos que aparecen en la Tabla 1 (en [10] se describen otros atributos aplicables).

El grafo de la **Fig. 2** muestra (parte de) el MO incluyendo los objetivos de la Tabla 1. Así, se observa como *Portability*, *Functionality*, y *Efficiency* son algunas de las características seleccionadas como *concerns* para el sistema EFTCoR. También dicha la figura nos muestra algunas de las relaciones de refinamiento que establecieron. Por ejemplo, la relación AND entre el objetivo *AdaptabilityWorkingEnvironment* y *AdaptabilityHullMaintenace-Operation* introducida para conseguir la satisfacción de *Adaptability*. Por otra parte, para el objetivo *AdaptabilityWorkingEnvironment* se ha establecido una relación de composición con el objetivo *ControlPossitioning*.

Tabla 1. Descripción (parcial) de Objetivos para EFTCoR

GOAL	DESCRIPTION
Functionality	The system has to provide functions which meet stated and implied needs
Suitability	The system has to provide an appropriate set of functions for specified tasks and user objectives
CoordinateDevices	The RDCU has to coordinate robotic devices according to the current mission procedure and global system state
ControlPossitioning	The RDCU has to control the positioning devices
ControlTools	The RDCU has to control the tools attached to positioning devices (blasting head, painting tool, etc)
Efficiency	The system should use efficiently their resources
TimeBehaviour	The system should respond with appropriate speed
PerformanceOperation	The system should respond with appropriate speed to the operation commands
Portability	The system should be able to be transferred from one environment to another
Replaceability	The system should be able to use different software product for the same purpose
Adaptability	The systems has to be adaptable for different specified environments
AdaptabilityWorkingEnvironments	The system operation has to be adaptable to different working environments
AdaptabilityHullMaintenanceOperation	The system operation has to be adaptable to different hull maintenance operations

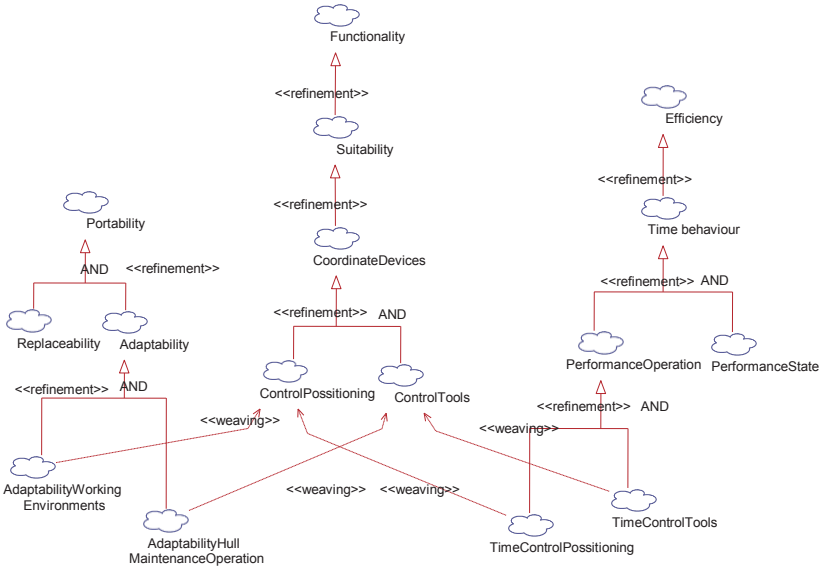


Fig. 2. (Vista parcial del) Modelo de Objetivos para EFTCoR.

4. Conclusiones y trabajos futuros

En este trabajo hemos presentado resumidamente un enfoque para la captura y análisis de requisitos en la que se integran orientación a objetivos y orientación a aspectos. Gracias a la OOR, se mantiene una trazabilidad explícita desde los objetivos hacia los requisitos, y desde estos últimos hacia las operacionalizaciones que se materializan en los componentes software. Por otro lado, el enfoque orientado a aspectos permite gestionar de manera elegante y efectiva el *crosscutting* que puede aparecer en la SRS cuando se entremezclan objetivos/requisitos. Definiendo relaciones de composición entre los elementos de un objetivo/requisito se consigue una representación óptima, desde la cual puedan incluso presentarse los objetivos/requisitos en su forma original si fuese necesario. Se ha definido un flujo de trabajo en el cuál se integran ambas aproximaciones, detallando una serie de tareas que ofrecen guías para la elaboración y organización de la especificación de requisitos. Otra de las ventajas que ofrece nuestra propuesta se refiere a la utilización del ISO/IEC 9126 como punto de partida en cuanto al establecimiento de posibles *concerns* (intereses del sistema). Con nuestro enfoque es posible ajustarse a las recomendaciones del IEEE 830-1998 [8] en cuanto a contenido de la SRS pero contando con significativas ventajas en cuanto a flexibilidad y versatilidad en la organización de los requisitos.

Es importante destacar que el concepto de aspecto no aparece explícitamente en el MO, es decir, no existe ningún constructor manifiesto para él, tal y como ocurre el paradigma Orientado a Aspectos, al menos en un nivel de diseño o implementación.

En el MO un *aspecto candidato* es aquel objetivo/requisito que presenta relaciones de composición con varios otros objetivos o requisitos (más de uno al menos).

Diversos retos quedan todavía por resolver. Uno de ellos está relacionado con las relaciones de composición entre objetivos/requisitos. Concretamente, con la idoneidad del conjunto de relaciones ofrecido y su semántica. Para ello, la problemática ofrecida por el caso de estudio facilitará la verificación y validación del mismo. Nuestro interés es que para las relaciones de composición y los objetivos/requisitos identificados se mantengan relaciones de trazabilidad hacia artefactos arquitectónicos, según el proceso planteado por ATRIUM.

Referencias

[1] Aspect-Oriented Software Development, http://www.aosd net

[2] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed Requirements Acquisition. In *Science of Computer Programming*, 20, (1993), 3-50.

[3] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, Non-Functional Requirements in Software Engineering, Kluwer Academic Publishing, 2000

[4] EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. European Project within the 5th Marco de trabajo Program (GROWTH G3RD-CT-00794), 2003.

[5] T. Elrad, R. E. Filman and A. Bader, "Aspect-oriented programming: Introduction", Communications of the ACM, Vol. 44 , Issue 10, pp. 29 – 32, October, 2001.

[6] C. Fernández, J.A. Pastor, P. Sánchez, B. Álvarez, A. Iborra: "Co-operative Robots for Hull Blasting in European Shiprepair Industry. Robotics and Automation Magazine (RAM)", special issue on Industrial Robotics Applications & Industry-Academia Cooperation in Europe. New Trends and Perspectives. (To appear in September 2004)

[7] ISO/IEC Standard 9126-1 Software Engineering- Product Quality-Part1: Quality Model, ISO Copyright Office, Geneva, June 2001

[8] IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications, en Volumen 4: Resource and Technique Standards, The Institute of Electrical and Electronics Engineers, Inc. IEEE Software Engineering Standards Collection.

[9] A. van Lamsweerde "Goal-Oriented Requirements Engineering: A Guided Tour", Invited Paper for 5th IEEE International Symposium on Requirements Engineering, Toronto, August, 2001, pp. 249-263.

[10] E. Navarro, I. Ramos and J. Pérez: "Goals Model-Driving Software Architecture", 2nd International Conference on Software Engineering Research, Management and Applications, May 5-8, 2004, Los Angeles, CA, USA.

[11] E. Navarro, I. Ramos and J. Pérez: "Software Requirements for Architectured Systems". Proc. 11th IEEE International Conference on Requirements Engineering, pp. 365-366, September 8-12, 2003, Monterey, CA, USA.

[12] J. Pérez, I. Ramos, J. Jaén, P. Letelier and E. Navarro, "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", Proc. 3rd IEEE International Conference on Quality Software, Dallas, Texas, USA, pp. 59-66, November 6 - 7, 2003.

[13] A. Rashid, P. Sawer, A. Moreira and J. Araújo, "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", Proc. International Conference on Requirements Engineering, September 9-13, 2002, Essen, Germany.

Dynamic Evolution in Aspect-Oriented Architectural Models*

Jennifer Pérez, Nour Ali, Jose Ángel Carsí, and Isidro Ramos

Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n
E-46022 Valencia, Spain
{jeperez, nourali, pcarsi, iramos}@dsic.upv.es

Abstract. This paper presents a solution to the evolution problem of software architectures. This solution is provided by PRISMA. PRISMA is an architecture modeling approach that integrates the advantages of Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD). This integration is reflected in its model and in its Architecture Description Language (ADL). In this paper, PRISMA is presented as a framework to evolve aspect-oriented and component-based architectures by requirements-driven evolution. The evolution is supported by means of a meta-level and the reflexive properties of PRISMA ADL which have been implemented as a middleware. In addition, it is demonstrated how the evolution services of the PRISMA meta-level permit the run-time evolution of software architectures using an industrial case study, the *TeachMover* Robot.

1 Introduction

Complex information systems frequently undergo changes in their functional and non-functional requirements. This is due to the fact that they are exposed to a high set of variability sources and they have a dynamic nature. In these software systems, the reconfiguration of the architecture topology at run-time is one of the most important requirements, especially in real-time systems. The relevance of this dynamic reconfiguration is due to the fact that complex systems usually cannot stop their activity.

In the last few years, there has been greater interest in evolution research in order to reduce the time and the cost of the maintenance process and to provide a solution for dynamic evolution. Thus, dynamic evolution appears as one of the main challenges of software.

* This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, DYNAMICA project TIC2003-07776-C02-02. Moreover, it is funded by the Microsoft Research Cambridge, "*PRISMA: Model Compiler of aspect-oriented component-based software architectures*" Project

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Dynamic Evolution in Aspect-Oriented Architectural Models. European Workshop on Software Architecture. 2005.

Software architectures [4] have emerged as a solution for the design and the development processes of complex information systems. There is a wide variety of architectural models, but there is no consensus about the different concepts and approaches. However, there is no doubt whatsoever that dynamic evolution of architectures is necessary to overcome changes in the requirements. This evolution must be able to evolve the structure as well as the behaviour of architectures.

PRISMA is an approach to develop complex information systems. This approach provides a model and an architecture description language (ADL). The PRISMA model defines software architectures by integrating aspect-oriented software development (AOSD) [1] and component-based software development (CBSD) [18]. In this paper, we present how the PRISMA model provides a solution for dynamic evolution and supports changes in aspect-oriented architectures requirements by means of a meta-level. The meta-level contains the definitions of architectural elements as data in order to update them by executing evolution services. In this way, the execution of services is reflected in the architecture by its updating the data (the concept of reflection). As a result, the PRISMA meta-level allows for the evolution of components and aspects as well as the dynamic reconfiguration of architectures.

The PRISMA prototype has been developed as a middleware using the .NET technology. This middleware contains the PRISMA meta-level in order to evolve the architectures at run-time. The dynamic reconfiguration of PRISMA approach has been tested using a real case study, the *TeachMover* robot. The *TeachMover* robot belongs to the tele-operation systems domain. One of the main features of systems of this kind is that they must deal with changes in architecture without stopping their execution. In this paper, we briefly introduce the *Teach Mover* robot case study and how the meta-level supports the dynamic reconfiguration of its architecture.

The structure of the paper is as follows: Section 2 presents a brief summary of the PRISMA model and the meta-model. Section 3 details the main characteristics of the PRISMA evolution. Section 4 presents the tele-operation domain and the *TeachMover* case study which is used through out the paper to exemplify the presented evolution techniques. Section 5 explains how the PRISMA evolution has been implemented. Section 6 compares related works on architecture evolution. Finally, conclusions and further work are presented in section 7.

2 Architectural Model

The PRISMA architectural model allows the definition of architectures of complex software systems [12]. Its main contributions are the integration of the AOSD [5] and the CBSD [17] and its reflexive properties. In this way, PRISMA specifies different characteristics (distribution, safety, context-awareness, coordination, etc.) of an architectural element (component, connector) using aspects, and it has a meta-level to evolve its architectural models.

A PRISMA architectural element can be seen from two different views, internal and external. The internal view (see Figure 1) shows an architectural element as a prism. Each side of the prism is an aspect of this architectural element. In this way, we represent that an architectural element of PRISMA is formed by a set of aspects;

whereas, the external view (see Figure 1) is an architectural element that encapsulates its functionality as a black box and publishes a set of services that it offers to the rest of the architectural elements.

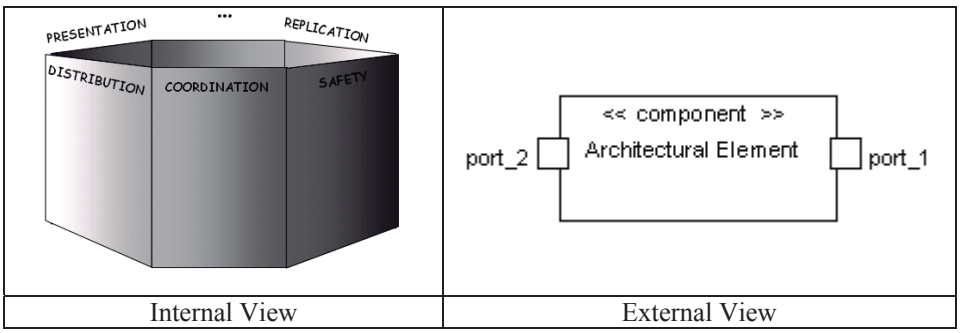


Fig. 1. Views of an Architectural Element

The PRISMA meta-level has been represented by means of a metamodel which contains one metaclass for each PRISMA concept. These metaclasses define the set of properties and evolution services for each considered concept in the model.

In the following, we present the main concepts of the PRISMA metamodel and their metaclasses with their evolution services.

2.1 Aspect

In our work, an aspect represents a specific *concern* (safety, coordination, distribution, etc) that crosscuts the software architecture, this means, those *concerns* that do not crosscut the architecture will not be an aspect. In order to avoid these *crosscutting-concerns*, a PRISMA architectural element is formed by a set of aspects that describe it from the different *concerns* of the architecture. The kind of aspects that form an architectural element depends on the *concerns* of the information system that we are specifying.

PRISMA is based on OASIS [6] to define the semantics of architectural models in a formal way and to preserve its main advantages, that is, the validation and the verification of architectural models and the automatic generation of code from the ADL. Due to the fact that PRISMA is based on OASIS, aspects are defined by attributes, services, preconditions, valuations, triggers and protocols. In addition, an aspect defines a *concern* and may specify the semantics of some interfaces (set of services published by a component). In Figure 2, we present the *aspect* package of the PRISMA metamodel.

The evolution services of the *aspect* metaclass (see Table 1) allow us to create and destroy aspects. Moreover, they permit the modification of their name, the interfaces that they specify, and the kind of characteristics that they define as well as the addition and removal of properties from the aspect.

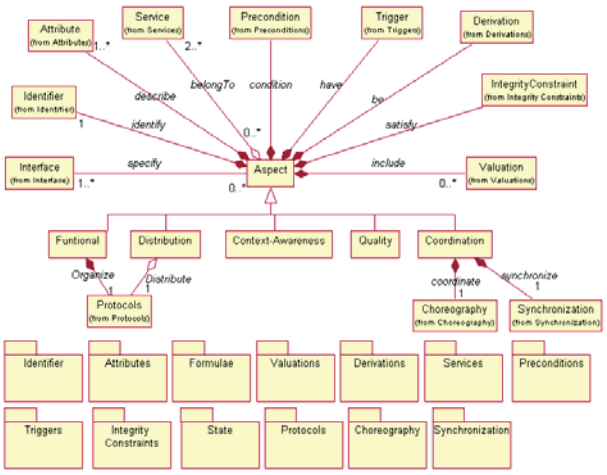


Fig. 2. The aspect package of the PRISMA metamodel

Table 1. Evolution services of the *Aspect* metaclass

```
Metaclass Aspect
 new NewAspect(name: string, type: AspectType[1]);
 destroy DestroyAspect;
 usingInterface(name: string);
 removeInterface(name: string);
 changeName (Newname: string);
 changeType(Type: AspectType);
 addAttribute (name: string, Type: type);
 removeAttribute(name: string);
 addServices(name:string, Type:type,
             ParameterList:string);
 removeService(name: string);
  … …
End_Metaclass Aspect;
```

[1] AspectType is a predefined enumerated type which contains the following list of values: coordination, distribution, quality, presentation, context_awareness, navigational and others. However, the user can also define new kinds of aspects.

The *NewAspect* service allows us to create a new aspect; its parameters define its name and the kind of properties that the aspect specifies. However, the *DestroyAspect* service destroys an aspect.

In addition, a set of services to modify a specific aspect are provided. For example, the *AddAttribute* service adds a new attribute to the aspect, but the *removeAttribute* service deletes an attribute from the aspect. Finally, services to change the aspect are provided (*changeName, changeType, etc)*.

These evolution services have associated constraints which must be satisfied before or after their execution in order to maintain the consistence of the architecture.

- **Weaving**

Simply defining an aspect (crosscutting concerns) is not enough. The methods indicating how an aspect is weaved (connected) with the rest of the aspects must also be determined. The weaving indicates that the execution of an aspect service can generate the invocation of services in other aspects (see Figure 3). Nevertheless, in order to preserve the independence of the aspect specification from the aspect weaving, the weaving is specified externally to the aspect.

As a PRISMA architectural element is formed by a set of aspects, the weaving is part of the architectural element specification; it is the glue of the aspects forming a prism (Internal View). This glue is achieved using the weaving methods that the model provides. The weaving methods are operations that describe the causality of the weaving services. The weaving methods which are typical of the AOP are the following:

- **after**: aspect1.service is executed **after** aspect2.service
- **before**: aspect1.service is executed **before** aspect2.service
- **instead**: aspect1.service is executed **in place of** aspect2.service

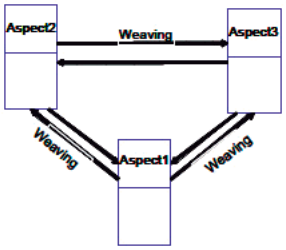


Fig. 3. Aspect weaving

Next, we present the *weaving* package of the PRISMA metamodel (see Figure 4) and the evolution services of the *weaving* metaclass (see Table 2).

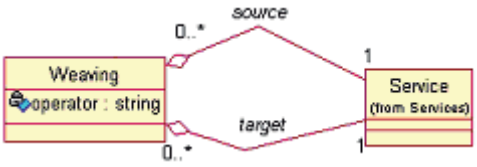


Fig. 4. The weaver package of the PRISMA metamodel

Table 2. Evolution services of the *Weaving* metaclass

```
Metaclass Weaving
   New NewWeaving(SourceAspect: string,
                  SourceService:string,
                  Operator:{after, before, instead},
                  TargetAspect:string,
                  TargetService: string);
   destroy DestroyWeaving();
   ChangeSourceWeaving (NewAspect: string,
                        NewService: string);
   ChangeOperator(NewOperator:{after, before,instead});
   ChangeTargetWeaving (NewAspect: string,
                        NewService: string);
End_Metaclass Weaving;
```

Similar to the *aspect* metaclass, the *weaving* metaclass (see table 2) allows us to create and destroy weavings. Moreover, its evolution services permit the modification of its properties. This update is separated into three parts, the first two of them modify one of the services that participate in the weaving and the third one changes the weaving operator.

2.2 Architectural Elements

2.2.1 Components

A PRISMA component is an architectural element that captures the functionality of the information system and does not act as a coordinator between other architectural elements. It is formed by an identifier, a set of aspects, its weaving relationships and the ports that offer and request services of a specific interface.

We present the *component* package of the PRISMA metamodel in Figure 5 and the evolution services of the *component* metaclass in Table 3.

The *NewComponent* service allows us to create a new component. However, the *DestroyComponent* service destroys the component. In addition, a set of services to modify a component is provided. For example, the *AddPort* service adds a new port to the component, but the *removePort* service deletes a port from the component.

It is important to take into account that there are evolution services that affect the architecture configuration such as the *AddPort* and the *RemovePort* services. The execution of services of this kind leads to the execution of other evolution services in order to adapt the configuration, thereby preserving the consistency of the architecture. For example, if a port is removed, its connections must be deleted.

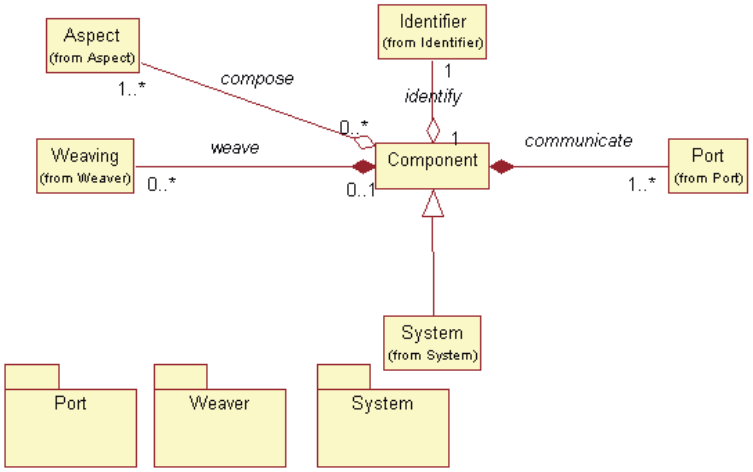


Fig. 5. The component package of the PRISMA metamodel

Table 3. Evolution services of the *Component* metaclass

```
Metaclass Component
  new      NewComponent(Name: string);
  destroy  DestroyComponent(Name:string);
           AddAspect (Name: string, type: AspectType);
           RemoveAspect(name: string);
           AddPort(Name: string, Interface: string);
           RemovePort(name: string);
           AddWeaving(InAspect: string,InService:string,
                  OutAspect:string, OutService: string);
           RemoveWeaving(name: string);
End_Metaclass Component;
```

2.2.2 Connectors

A PRISMA connector is an architectural element that acts as a coordinator between other architectural elements. It is formed by an identifier, a set of aspects, its weaving relationships and one or more roles, whose signature is a specific interface. These roles represent points of interaction among components.

We present the *connector* package of the PRISMA metamodel in Figure 6 and the evolution services of the *connector* metaclass in Table 4.

Similar to the *component* metaclass, the *connector* metaclass (see table 4) allows us to create and destroy connectors. Moreover, it permits the modification of its properties, and it can affect the architecture configuration.

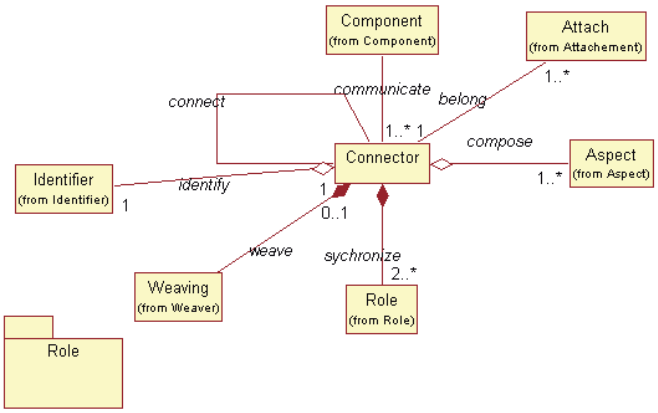


Fig. 6. The connector package of the PRISMA metamodel

Table 4. Evolution services of the *Connector* metaclass

```
Metaclass Connector
  new      NewConector(Name: string);
  destroy DestroyConector();
  AddAspect(Name: string, type: AspectType);
  RemoveAspect(name: string);
  AddRole(Name: string, Interface: string);
  RemoveRole(name: string);
  AddWeaving(InAspect: string,
  InService:string,
  OutAspect:string,
  OutService: string);
  RemoveWeaving(name: string);
End_Metaclass Connector;
```

2.2.3 Systems

PRISMA components can be simple or complex. The complex ones are called systems. A PRISMA system is a component that includes a set of connectors, components and other systems that are correctly attached. Figure 7 presents the *system* package of the PRISMA metamodel and Figure 5 shows the inheritance relationship between the metaclasses component and system.

The evolution services of the *system* metaclass allow us to create and destroy systems. Moreover, the *system* metaclass permits the modification of the system name and the set of components, connectors, attachments, bindings and ports that it includes. A system definition must specify the connection and composition relationships (attachments and bindings, respectively) among the architectural elements that it contains. The attachment relationship establishes the connection among ports of components and roles of connectors. The binding relationship defines the composition between the system and the architectural elements which it contains by means of its ports. The signatures of the metaclass services are presented in Table 5.

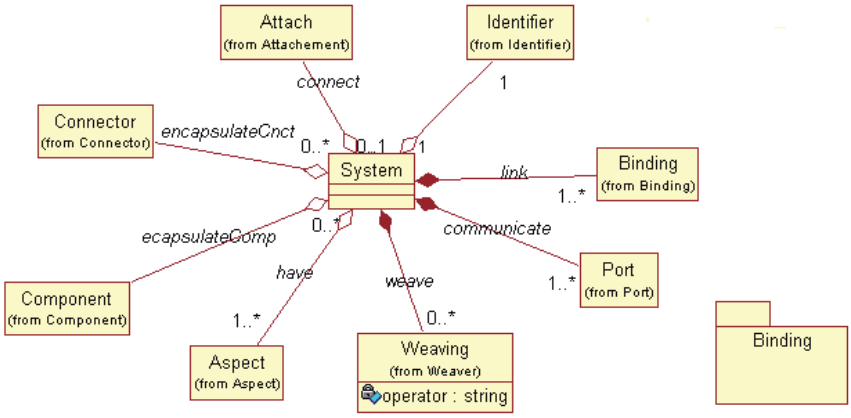


Fig. 7. The system package of the PRISMA metamodel

Similar to the *component* metaclass, the *system* metaclass (see Table 5) allows us to create and destroy systems. Moreover, it permits the modification of its properties, and it can affect the architecture configuration.

Table 5. Evolution services of the *System* metaclass

```
Metaclass System
   new NewSystem(Name: string);
   destroy DestroySystem;
   AddComponent(SystemName: Name: string);
   RemoveComponent(name: string);
   AddConnector(Name: string);
   RemoveConnector(name: string);
   AddAttachment(Component: string, Port: string,
                 Connector: string, Role: string);
   RemoveAttachment(name: string);
   AddBinding(Architectural_element: string,
              CPort: string, System: string,
              SPort: string);
   RemoveBinding(name: string);
   AddPort(Name: string, Interface: string);
   RemovePort(name: string);
End_Metaclass System;
```

3 PRISMA Evolution

PRISMA supports software evolution by means of a meta-level and the reflexive properties of its ADL. In this way, PRISMA provides the evolution of aspect-oriented software architectures and their dynamic reconfiguration.

The PRISMA architectures are defined at two different levels of abstraction: the type definition level and the configuration level. The first level defines architectural types (interfaces, aspects, components, connectors and systems) with a high abstraction level. The PRISMA types defined in this level are stored in a PRISMA library so that they can be reused by other types or specific architectures. The second level designs the architecture of software systems by creating and interconnecting instances of the defined architectural types in the previous level. In other words, in this level, we specify the topology of a specific software system.

The fact that PRISMA architectures are defined at two different levels of abstraction implies that the PRISMA evolution can be classified into two kinds: type evolution and configuration evolution (reconfiguration).

The evolution of types is achieved by invoking evolution services that do not affect the structure of the architecture or the communication among architectural elements. These evolution services only update the internal specification of architectural

elements (internal view).

An architecture configuration is evolved by invoking evolution services that update the number of architectural elements, the communication among them, and the structure of the architecture. These evolution services affect the external view of the architectural elements and belong to the *interface, port, role, attachment, binding, component, connector* and *system* metaclasses (see Table 3, 4 and 5). The evolution services of the *port and role* metaclasses can modify the architecture configuration because they can create and destroy ports or roles of a specific component or connector, respectively. At the same time, these changes can produce the evolution of the architecture attachments and the architecture bindings by creating, destroying or modifying them. This is due to the fact that attachments and bindings are defined by using ports and roles. In addition, the creation of new instances of components, connectors and systems produces a big change in the architecture because it implies other changes such as new ports, roles, attachments or bindings. In the same way, the destruction of instances of components, connectors and systems produces meaningful changes in the architecture.

Finally, it is important to keep in mind that the evolution of PRISMA architectures can be done at run-time. The evolution at run-time allows the dynamic reconfiguration of PRISMA architectures. This dynamic reconfiguration is achieved by executing evolution services at run-time. In this way, it is possible to enforce the business rules that produce the architectural changes at run-time.

4 A Case Study: The *TeachMover* robot

We are currently working on the *PRISMA: Model Compiler of Aspect-Oriented Component-Based Software Architectures* Microsoft® Research Cambridge project. This project consists of the specification of industrial systems such as the EFTCoR teleoperation system [13]. EFTCoR is a robotic platform that cleans the hulls of ships and that has strong requirements in terms of adaptability to different devices, safety for operators, response time, etc. These systems need dynamic reconfiguration to overcome their large set of variability sources.

Before developing the software architecture of EFTCoR [3], we specified and implemented the PRISMA architecture of the *TeachMover* robot [19]. The *TeachMover* is simpler than EFTCoR, but it has the same architectural features. We are now implementing EFTCoR reusing the implemented components of the *TeachMover* robot.

The *TeachMover* robot is composed by a set of joints that permit its movement: *Base, Shoulder, Elbow* and *Wrist*. In addition, it has a *Tool* to perform different tasks. In this case study, the tool is a gripper where open and close actions allow it to pick up objects (see Figure 8). The functionality of the *TeachMover* robot is to move objects from an initial position to a final one. The movements of the robot are ordered by an operator from a computer.

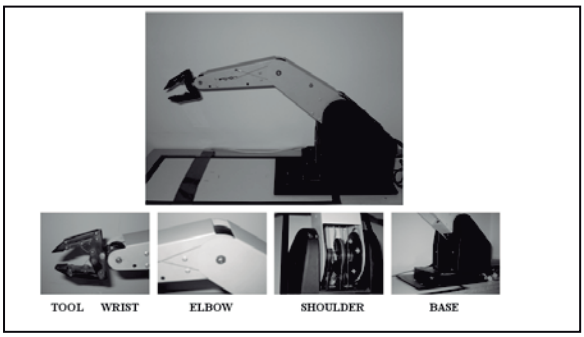


Fig.8. The *TeachMover* Robot

The *TeachMover* architecture has a lot of components, connectors and attachments at different levels of abstraction. Due to the complexity of the system, in this paper, we only focus on the higher level of abstraction because it is more familiar to the reader and it is easier to understand. Figure 9, shows this level of the *Teach Mover* architecture. It consists of two components, *Operator* and *RUC*(Robot Unit Controller) that are connected through a connector (*CnctMovement*). The components and connector are inside of a system called *Teach Mover.* In addition, this system includes two attachments to link each component with the connector by means of their ports and roles (*Commands*, *Robot*). Figure 9 also shows that the Operator and the Robot are local (they are situated in the same node) as they form part of the same system.

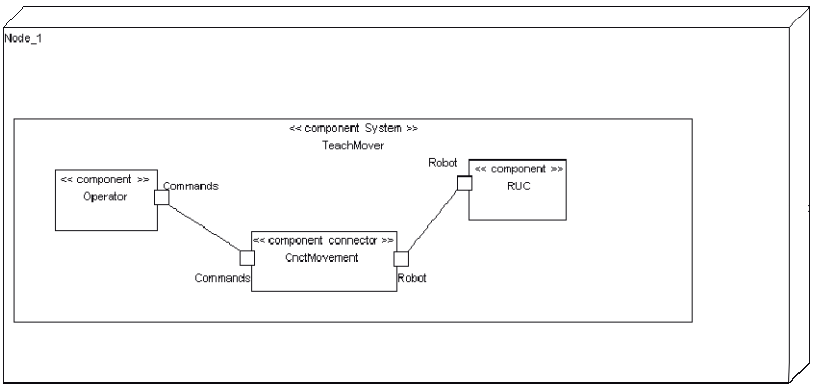


Fig. 9. Architecture Configuration between the Operator and the Robot

5 PRISMA Meta-level implementation

The main purpose of PRISMA is to become a framework to model software architectures, providing their automatic code generation, evolution and maintaince. Therefore, we have implemented the PRISMA approach using the .Net platform as a middleware to execute PRISMA software architectures. The middleware provides the evolution services of the metalevel to evolve PRISMA software architecture.

This section introduces the PRISMA middleware and explains how the implemented evolution services of the middleware are used in a reconfiguration case of the TeachMover architecture.

5.1 PRISMA Middleware

To implement PRISMA applications, an abstract middleware which sits above the .Net platform has been developed (see Figure 10). This middleware offers the extra functionalities and characteristics of PRISMA which .Net does not directly offer. Thus, each PRISMA type (*Aspects*, *Components*, *Attachments* and *Bindings*) has been mapped to a .Net construct. In this way, the evolution services of the PRISMA meta-level are provided by the middleware.

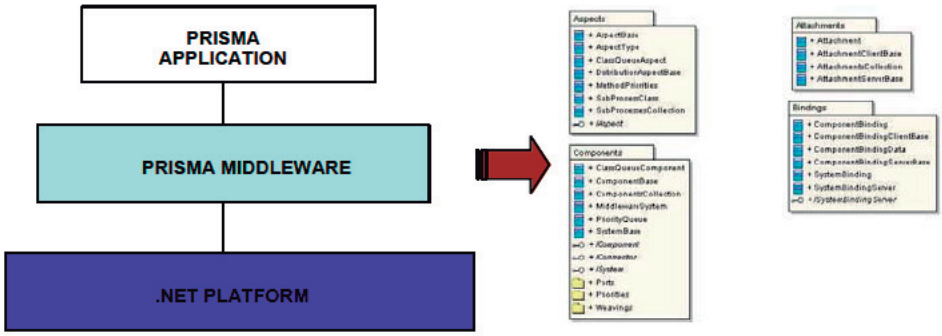


Fig. 10. The PRISMA Middleware

As PRISMA also specifies software architectures of distributed systems, this has also been taken into account in the development of the middleware. A PRISMA middleware has to run on each node where a PRISMA application needs to be executed (see Figure 11). Each middleware manages the architectural element instances that are executing in a specific node, providing the instances with necessary maintenance and evolution services. In order to maintain the consistence of the distributed software architecture and to make the instances work as if they were local instances, each middleware is able to interchange information with the other middlewares of the different nodes of a software architecture. We have used the .Net Remoting technology [10] to implement the distributed communication.

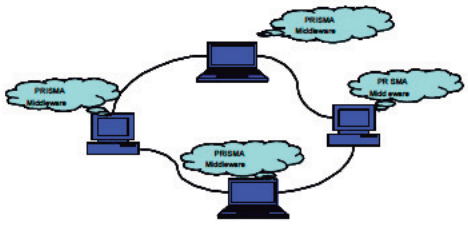


Fig. 11. PRISMA middleware running in distributed nodes.

In our case study (see Figure 9), each node of the *Teach Mover* architecture must have a middleware running in order to enable the execution of the software architecture.

5.2 Dynamic Reconfiguration in the *TeachMover* Architecture

In this section we are going to demonstrate how the evolution techniques previously presented are applied to the TeachMover Robot case study and how an evolution service unchains other evolution services to preserve the consistency of the architecture.

As mentioned above, the *TeachMover* robot belongs to the tele-operation domain and one of the main features of systems of this kind is the fact that they need to deal with changes in architecture without stopping their execution. One of the requirements that emerged after the teach mover architecture was implemented is that the operator could move the robot from computers located in different places. Taken into account that the initial configuration of the *TeachMover* architecture is the one shown in Figure 9, our first implementation version of this system does not satisfy this requirement. This is because the operator component is forced to be at the same computer (node) as the robot controller (RUC) component. This restriction exists because these components are inside the same system and the operator component cannot move itself without moving the entire system. For this reason, the initial configuration does not allow the operator component movement from one computer to another and to access to the robot controller (RUC) component in a distributed way. In order to overcome this requirement at execution time, the architecture can reconfigure itself by externalizing the Operator component from the *TeachMover* system. In this way, the Operator component can access the TeachMover system from another node independently of which node it is working at.

Figure 12 shows how the architecture should be configured for the operator be able to move the robot from different places. As Figure 12 shows the Operator component is connected to the TeachMover system in a distributed way and is not part of the system. To perform this change, a set of meta-level services must be executed at run-time. These services are provided through the PRISMA middleware, which contains the meta-level implementation. The set of executed meta-level services is discussed bellow. It is important to keep in mind that the new Node, where the Operator component is going to be moved to, should also have the middleware executing.

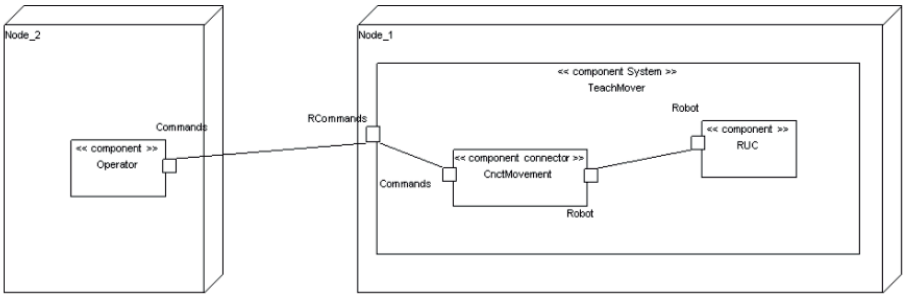


Fig. 12. The reconfiguration of the architecture in Figure 8 caused by the mobility of the Operator

When the operator component decides to move to a new node (*Node_2*), it notifies the decision to the middleware of Node_1. This middleware request the *NewComponent(Operator)* service of the meta-class *Component* from the middleware of Node 2 (see Table 3). The same Operator Component exists at both, Node_1 and Node_2 (this is performed by serializing the type). When the Operator component is created in Node_2, the middleware of Node_1 is notified and it destroys the Operator component on its site. Node_1 performs this by executing the *DestroyComponent(Operator)* of its meta-class *Component*. The results are shown in Figure 13. The operator component is at Node_2 and the *TeachMover* system, which encapsulates the attached *CnctMovement* and RUC, is at Node_1.

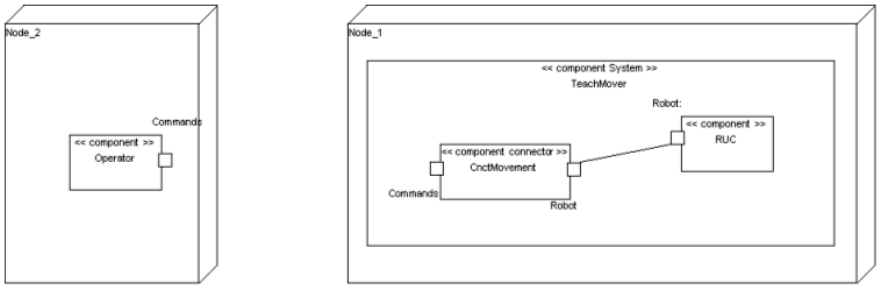


Fig. 13. The result of applying the *NewComponent(Operator)* and *DestroyComponent(Operator)* to the architecture in Figure 9

However, this is not enough to enable the Operator component to communicate with the *RUC* component. Therefore, the middleware of Node_1 requests from the TeachMover system to execute the *AddPort(RCommands, ICommands)* of the meta-class *System* (see Table 5). This service adds a port to the *TeachMover* system called *RCommands* of the *ICommands* interface type. The middleware then requests TeachMover system to execute the *AddBinding(CnctMovement, Commands, TeachMover, RCommands)* of the meta-class *System*. Thus, a binding is created between the *TeachMover* and the *CnctMovement*. Once this is performed, the Node_1 middleware request the execution of the *AddAttachment(Operator, Commands,*

*TeachMover, RCommands)*service to middleware of Node_2 in order to create an attachment between the distributed *TeachMover* and *Operator* [2] . The final configuration of the architecture is achieved without stopping the execution of the system, the final configuration of the architecture (see Figure 12). This proposal allows the operator to become a mobile component which can access the robot in a distributed manner.

6 Related Works

This work is focused on providing a complete framework to support architecture evolution. This evolution framework is for a specific model called PRISMA. PRISMA takes into account various relevant solutions such as Component-Based Design, the Separation of Concerns Aspect-Oriented and a powerful reflexive meta-level.

Many Architecture Definition Languages (ADLs) that support evolution have been proposed. Each one has positive and negative aspects. An interesting comparison between these languages is done in the work by [14]. The proposal by Loques & Leite [7] is very similar to our approach. Their Model R-RIO incorporates dynamic reconfiguration by means of a reflexive architecture, but the use of the meta-level is quite different from ours. Other approaches use meta-programming techniques but at an implementation level such as the work by [9]. In GUARANA [11], a complex meta-level is defined at compilation time in an ad-hoc and independent way.

However, these proposals do not provide technological support for evolution that preserves the features of the model. Moreover, they have not been tested with complex case studies such as tele-operation systems. Furthermore, there is no aspect-oriented ADLs that provide evolution techniques for aspects, components and dynamic reconfiguration of architectures. As a result, PRISMA is presented as a framework to evolve aspect-oriented and component-based software architectures of complex systems at run-time.

7 Conclusions and Future Work

In this paper, the PRISMA model has been briefly presented in order to show how it is possible to define aspect-oriented architectures. The paper classifies evolution into different kinds and explains how the model overcomes each kind of evolution. Thus, a solution for the evolution of aspect-oriented software architectures has been provided. As a result, the cost of people and time invested in the maintenance process of the PRISMA architectural models is reduced.

[2] The connector between the *Operator* component and the *TeachMover* system has been omitted because it is a relay station. The connector is implicitly supported by the existing attachment between the *Operator* and the *TeachMover*.

It is also important to keep in mind that the division of the PRISMA architecture specifications into two levels of abstraction allows us to distinguish between the evolution of reusable types and the evolution of a specific architecture.

This paper presents the main concepts of the PRISMA metamodel in detail , it also presents its implementation by means of a middleware and the results that we have achieved executing a real case study such as the TeachMover robot.

Further work will be dedicated to improving the automatic code generation of the middleware. Sometimes, it is not necessary to evolve the whole system. As Díaz states in his work [2], systems can be divided into a static part and a volatile part. The volatile part is the one that can undergo changes due to the modification of business rules; however, the static part cannot undergo changes during the life of the application. Specifically, we want to improve the generation code-time and the run-time of the middleware by means of generating only the part of the metalevel that is needed to evolve the volatile part of a specific architecture.

Over the long term, our work with regard to software evolution will be related to the data evolution problem of software architectures, where we will apply our previous experience about data migration and data evolution of object-oriented conceptual schemas [14, 15, 16].

8 Bibliography

1. Aspect-Oriented Software Development http://aosd.net
2. Diaz, O., Paton, N. W.: Stimuli and Business Policies as Modelling Constructs: their definition and validation through the event calculus. Proc. CAiSE, Lecture Notes in Computer Science, Vol. 1250. Springer-Verlag,A. Olive and J. Pastor (eds) (1997) 33-46.
3. EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794. (2002)
4. Garlan, D., Kompanek, A.: Reconciling the Needs of Architectural Description with Object-Modeling Notations. Proceedings «UML» 2000, Lecture Notes in Computer Science, Vol. 1939. Springer-Verlag, York, UK, October 2-6, (2000) 498-512.
5. Kiczales, G., Hilsdale, E., Huguin, J., Kersten, M., Palm, J., and Griswold, W.G: An Overview of AspectJ. In proceedings of the European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 2072. Springer-Verlag (2001)
6. Letelier, P., Sánchez, P., Ramos I., Pastor O.: OASIS 3.0, A formal language for the object oriented conceptual modeling. Polytechnic University of Valencia, SPUPV-98.4011, ISBN 84-7721-663-0 (1998) (In Spanish).
7. Loques, O., Sztajnberg, A., Leite, J., Lobosco, M.: On the Integration of Meta-Level Programming and Configuration Programming. In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, Lecture Notes in Computer Science Vol. 1826, Springer-Verlag, Heidelberg, Germany, June (2000).191-210.
8. Medvidovic, N., Taylor, R. N.: A classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions of SW Engineering, Vol. 26, nº 1, January (2000)
9. McGurren, F., Conroy, D.: X-ADAPT: An Architecture for Dynamic Systems. Workshop on Component-Oriented Programming, ECOOP, Málaga, Spain (2002)
10. Microsoft .Net Remoting: A Technical Overview,

http://msdn microsoft.com/library/default.asp?url=/library/en-/dndotnet/html/hawkremoting.asp
11. Oliva, A., Garcia, I. C., Buzato, L. E.: The Reflective Architecture of Guaraná. Technical Report IC-98-14. Computation Institute, Campiñas University, April (1998)
12. Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E.: PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. In proceedings of 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November (2003)
13. Pérez, J., Ali, N.H., Ramos, I, Pastor, J. A., Sánchez, P., Álvarez, B.: Tele-operated Systems Development using the PRISMA approach. VIII conference on Software Engineering and Databases, Alicante, Spain (2003)
14. Pérez, J., Carsí, J A., Ramos, I.: On the implication of application's requirements changes in the persistence layer: an automatic approach. Workshop on the Database Maintenance and Reengineering (DBMR'2002), IEEE International Conference of Software Maintenance, Montreal, ISBN: 84-699-8920-0, Canada, October 1st (2002) 3-16.
15. Pérez J., Carsí J. A. and Ramos I., "ADML: A Language for Automatic Generation of Migration Plans*", The First Eurasian Conference on Advances in Information and Communication Technology*, Tehran, Iran, octubre http://www.eurasia-ict.org/ © Springer LNCS vol n.2510 (2002)
16. Pérez, J., Anaya, V., Cubel, J M., Domínguez, F., Boronat, A., Ramos, I., Carsí, J A.: Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. Software Evolution Through Transformations: Towards Uniform Support throughout the Software Life-Cycle Workshop (SET'02), First International Conference on Graph Transformation(ICGT2002), ENTCS vol n. 72.4, Barcelona, Spain, October (2002)
17. D'Souza, D., Wills, A.: Objects, Components and Frameworks with UML. The Catalysis approach. Addison-Wesley (1.999)
18. Szyperski, C.: Component software: beyond object-oriented programming. ACM Press and Addison Wesley, New York, USA (1998).
19. The *TeachMover* Robot, http://www.questechzone.com/microbot/teachmover htm

Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology

Jennifer Pérez, Nour Ali, Cristóbal Costa-Soria, Jose A. Carsí, Isidro Ramos
ISSI, Department of Information Systems and Computation
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022, Valencia, Spain
{jeperez | nourali | ccosta | pcarsi | iramos} @dsic.upv.es

ABSTRACT

Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD) have emerged in the last few years as new paradigms of software development. Both approaches provide techniques to improve the structure and reusability of the code. In addition, Aspect-Oriented Programming (AOP) permits the reduction of the maintainability and development costs of the final code by means of the separation of concerns in *aspects*. However, the .NET framework does not provide support for the Aspect-Oriented approach. In this paper, we present a solution for this lack found in .NET technology by means of a .NET middleware called PRISMANET. PRISMANET is based on the PRISMA approach, which integrates the advantages of AOSD and CBSD and supports dynamic reconfiguration of software architectures at run-time. This middleware has been completely developed using the .NET framework and has been tested with real case studies, such as the *Teach Mover Robot.* As a result, PRISMANET extends the .NET technology by the execution of aspects on the .NET platform, the reconfiguration of software architectures (local and distributed) and the addition and removal of aspects from components at run-time.

Keywords

Aspect-Oriented Programming (AOP), Component-Based Software Development (CBSD), dynamic reconfiguration of software architectures, addition and removal of aspects at run-time, concurrency, distribution, mobility.

1. INTRODUCTION

Complex structures, non-functional requirements, reusability and run-time evolution are leading properties that current software systems need to deal with. Two software development approaches have emerged to respond to these needs: Component-Based Software Development (CBSD) [Szy98] and Aspect-Oriented Software Development (AOSD) [AOS05].

On the one hand, CBSD decomposes the system into reusable entities called components that provide services to the rest of the system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2005 *conference proceedings*,
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

On the other hand, AOSD allows for the separation of concerns by modularizing crosscutting concerns into a separate entity, called *aspect*. The encapsulation of the aspect allows for the reusability of the same aspect in different objects and the evolution of an aspect without affecting the rest of objects and aspects. The main effort in this approach has been made at the implementation level. As a result, this approach has emerged as a new paradigm of software development. However, the .NET framework does not provide support for this new approach, making the use of .NET technology by the "Aspect-Oriented Community" unfeasible.

In this paper, we present a solution that provides support to the AOSD by means of the PRISMANET middleware. PRISMANET implements PRISMA. PRISMA is an approach to develop complex information systems that provides a model and an Architecture Description Language (ADL).

Pérez, J., Ali, N., Costa, C., Carsí, J. A., Ramos, I. Executing Aspect-Oriented Component-Based Software Architectures on . NET Technology. Proc. of the 3rd International Conference on .NET Technologies. 2005.

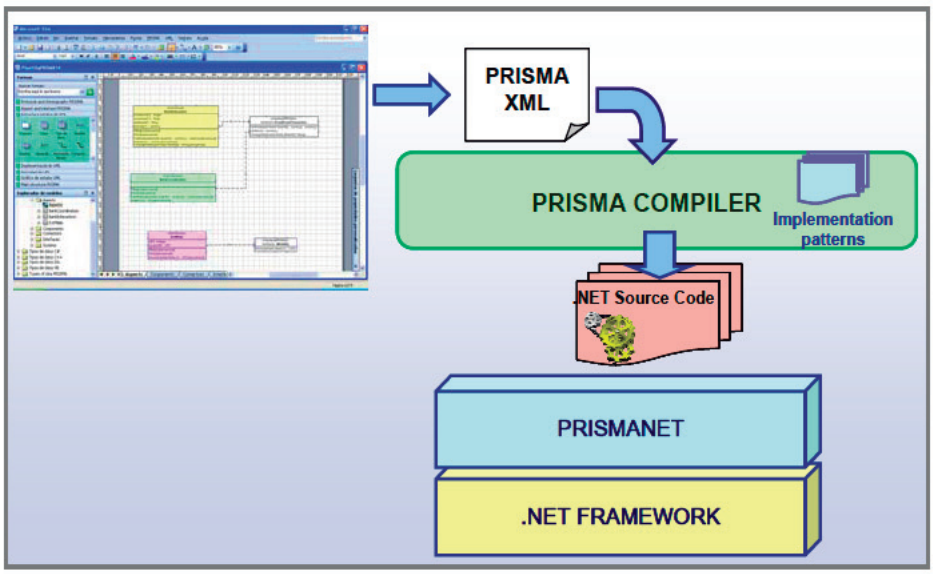


Figure 1. PRISMA approach

The PRISMA model defines software architectures by integrating AOSD and CBSD. In addition, PRISMA supports evolution by means of a meta-level. Its meta-level allows the evolution of types and the dynamic reconfiguration of architectures. The PRISMANET includes the PRISMA model, its meta-level and the distribution support for mobility.

PRISMANET not only extends the .NET technology by the incorporation of aspects, but it also provides the reconfiguration of software architectures (local and distributed) and the addition and removal of aspects from components at run-time. These complex features have been successfully implemented thanks to the mechanisms that the .NET technology provides to deal with them. The most important .NET technology mechanisms [Rob03] that we have used are: delegates, reflection, serialization, .NET Remoting [Mic05] and dynamic code generation.

As a result of the PRISMANET implementation, PRISMA software architectures can be developed and can be executed on the .NET platform. In addition, PRISMANET allows .NET programmers to develop applications with aspect-oriented, mobility and dynamic evolution properties.

Currently, the PRISMA approach allows the development of software systems with all its advantages by extending PRISMANET classes. The middleware has been developed and tested with real case studies, such as the *TeachMover* robot [Tea05] and the *EFTCOR* teleoperation system [EFT02] to clean the hulls of the ships. As a result of this work, we are able to move the robot with the Aspect-Oriented .NET technology and to develop the PRISMA CASE model compiler based on the middleware. For this reason, we are currently developing the compiler in order to automatically generate C# code from graphical diagrams (see Figure 1).

The goal of this paper is to show how the aspect-oriented, mobility and run-time evolution properties of PRISMANET have been implemented using the .NET technology mechanisms that have been previously mentioned.

The structure of the paper is as follows: Section 2 briefly introduces the basic concepts of the PRISMA model to understand the middleware implementation. Section 3 explains the PRISMANET middleware implementation in detail: aspects and components, concurrency, mobility and run-time evolution. Section 4 presents a comparison with other approaches that introduce aspect-oriented programming in .NET technology and points to the disadvantages that PRISMA overcomes. Finally, conclusions and further work are presented in section 5.

2. PRISMA

The PRISMA model allows for the definition of architectures of complex software systems [Per03]. Its main contributions are the integration of the AOSD and the CBSD and its reflexive properties. In this way, it specifies different characteristics (distribution, safety, coordination, etc.) of an architectural element (component, connector) using aspects, and it is able to evolve its architectures by means of a meta-level.

A component is an architectural element that captures the functionality of the system and does not act as a coordinator among other elements. However, a connector is an architectural element that acts as a

coordinator among components. In the PRISMA model, the connector does not have the references of the components that it connects to and vice versa. In this way, both components and connectors are reusable. The channels between two connected architectural elements have their references. The channels that connect components and connectors are called *attachments*.

Architectural elements can be seen from two different views, internal (*white box view)* and external (*black box view*). The *white box view* shows an architectural element as a prism being an aspect of this architectural element each side of the prism (see Figure 2); whereas, the *black box view* encapsulates its functionality and publishes a set of services that offers to other architectural elements (see Figure 3).

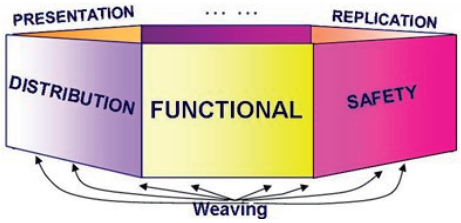


Figure 2. White box view of a component

A PRISMA aspect represents a specific *concern* (safety, coordination, distribution, etc) that crosscuts the software architecture. This means that those *concerns* that do not crosscut the architecture are not going to be an aspect. In order to avoid these *crosscutting-concerns*, a PRISMA architectural element is formed by a set of aspects that describe it from the different *concerns* of the architecture. The kinds of aspect (safety, coordination, distribution, etc) that form an architectural element depend on the *concerns* of the information system that is being specifying.The main elements that form an aspect are the following:

- *Attributes*: store information about the characteristics of the aspect.
- *Valuations*: specify the changes in attribute values by the execution of a service.
- *Services:* offer functionality of a specific *concern.*
- *Protocols:* describe the order and the state in which a service could be executed.

A component is formed by a set of aspects (functional, distribution, etc.), their synchronization relationships (*aspects weaving*) and one or more ports. These ports represent the interaction points among components. The type of ports is an interface that publishes a set of services.

The weaving is the glue of the aspects forming a prism. The weaving determines how an aspect is connected (synchronized) with the rest of the aspects. It indicates that the execution of an aspect service can generate the invocation of services in other aspects. However, to preserve the independence of the aspect specification from the aspect weaving, the weaving is specified outside the aspect and inside the component. The weaving methods are operations that describe the causality of the weaving services. The weaving methods are commonly used in the AOP. They are as follows:

- **after**: aspect1.service is executed **after** aspect2.service
- **before**: aspect1.service is executed **before** aspect2.service
- **instead**: aspect1.service is executed **in place of** aspect2.service

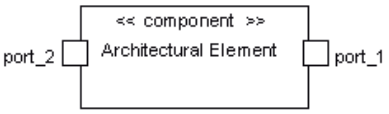


Figure 3. Black box view of a component

3. .NET MIDDLEWARE

PRISMA ADL (Architecture Description Language) is a specification language independent of the development platform. For this reason, an abstract middleware that sits above the .NET platform has been developed to implement .NET PRISMA applications. This middleware is called PRISMANET, and its implementation has been carried out in C# language using the standard techniques that the .NET framework provides, that is, without extending the development platform. As a result, PRISMANET can be executed in the .NET platform without having to do anything else other than starting the execution of the middleware.
PRISMANET offers the extra functionalities and characteristics which .NET does not directly provide. It allows for the execution of aspects, the reconfiguration of software architectures (local and distributed), the load of components, the creation of execution threads, the management of the local components, the addition and removal of aspects from components at run-time, the mobility and replication, etc.

PRISMANET architecture

The PRISMANET architecture is constituted by two modules: server and framework (see Figure 4):

- **PRISMA Server:** This module provides services to manage, move, maintain and evolve components.
- **PRISMA Framework:** This module is the user interface that offers the user the available services of the Server module. In addition, the state messages of the middleware are displayed on this user interface.

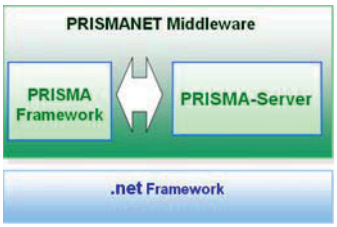


Figure 4. PRISMA Middleware

As PRISMA specifies software architectures of distributed systems, distribution needs has also been taken into account in the development of the middleware. PRISMANET has to run on each node where a PRISMA application needs to be executed (see Figure 5). Each middleware manages the architectural elements instances that are being executed in its specific node, providing the necessary distribution, mobility, maintenance and evolution services to the instances. In order to keep the consistence of distributed software architectures and to make the instances work as if they were local instances, each middleware is able to interchange information with the other middlewares of the different nodes of a software architecture.

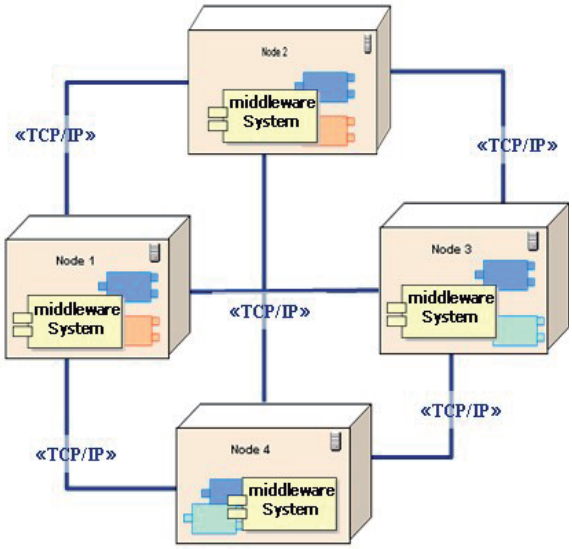


Figure 5. PRISMA Middleware running in distributed nodes

There are three kinds of communications concerning PRISMANET and the applications that run on it:

- Calls from the components to the middleware to ask for mobility and replication services.
- Communication among different components as a result of the execution of the application.
- Communication among different middlewares to find out locations of components, to move components, to evolve the architectures, etc.

PRISMA Model Implementation

Each concept defined in the PRISMA model has been implemented in the Server module of the PRISMANET. In this section, we focus on the aspects and components. The implementation has been carried out preserving the following features:

- The run-time evolution of applications must be possible. As a result, the dynamic code generation to add and remove aspects, components, connectors and attachments must be allowed.
- The implementation has to be as close as possible to the model in order to facilitate the future automatic code generation.
- The execution of attachments, connectors and components must be concurrent. In addition, the concurrency among the different aspects that form a component must be preserved.

3.2.1. Aspects

An aspect has been implemented as a C# class called *AspectBase* of the PRISMANET. This class stores the name of the aspect and its thread reference.

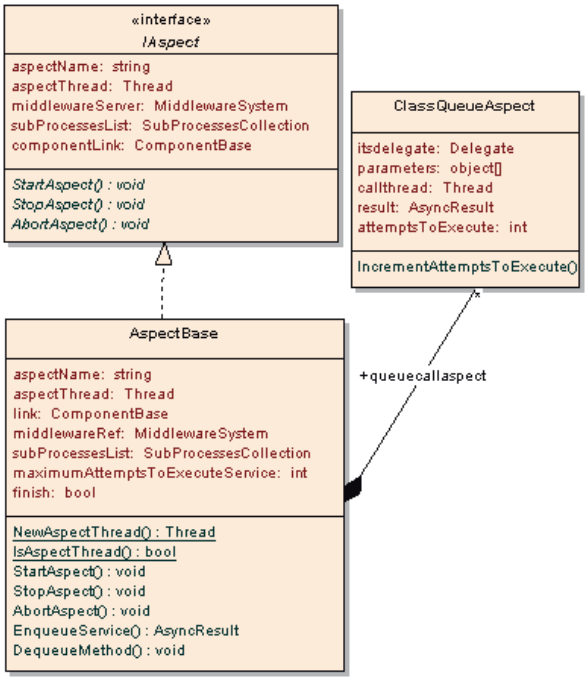


Figure 6. *AspectBase* class of PRISMANET middleware[1]

The *AspectBase* class has the references of the component and the middleware that it belongs to in order to request them services. In addition, as the

[1] The set of classes that appear in the figure have been automatically generated from the source code of PRISMANET using the Sparx tool [Spa05]

middleware must guarantee the execution of services without blocking the requesters, when a service of an aspect requires its execution while another service is being processed, the aspect stores the service that can not be immediately attended in a queue. As a result, the aspect thread is continuously processing the requests of the queue (see Figure 6). Finally, it is important to emphasize that the *AspectBase* class offers three services: *startAspect* to start the execution of the aspect thread, *stopAspect* to stop the execution of the aspect thread and *abortAspect* to definitively stop the execution of the aspect thread.

The kinds of aspects that can be defined in the PRISMA model are unlimited. However, each one has the functionality described above. For this reason, they are a subclass of the *AspectBase* class and inherit this functionality (see Figure 7).

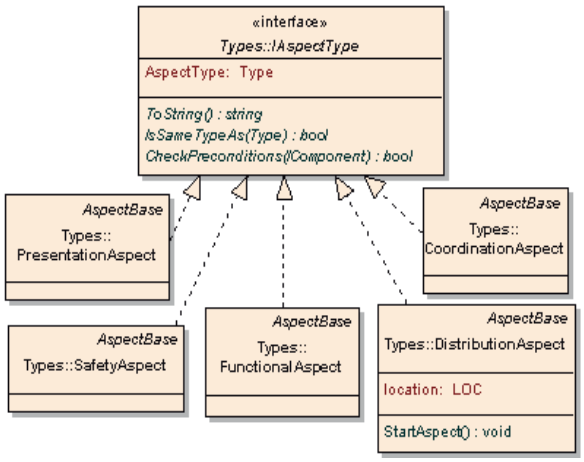


Figure 7. Classes of several kinds of aspects[1]

As a result, PRISMANET allows the implementation of a specific aspect by creating a C# class that inherits from one of the classes that represent one kind of aspect. It is important to keep in mind that aspects must be *serializable* in order to enable the mobility of aspects in distributed architectures.

In a specific aspect, the PRISMA *attributes* are implemented as private variables. The PRISMA *services* are programmed as private methods that implement their respective *valuations*. They also check whether their execution is enabled in accordance with the established order of the *protocol*. An example of a specific safety aspect is presented below:

```
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Middleware;
using PRISMA.Attachments;

namespace Robot
[Serializable]
public class SMotion : SafetyAspect
{
  #region Definition of PRISMA Variables
      double minimun;
      double maximun;
  #endregion

  public SMotion(double initialMinimum,
                 double initialMaximum) :
                 base("Smotion")…

  public AsyncResult Check(double newAngle,
                 out bool secure)…
}
```

Finally, it is important to emphasize that specific aspects are packaged in an assembly in order to facilitate their distribution over the network and their integration in a library.

3.2.2. Components

A component has been implemented as a C# class called *ComponentBase* of the PRISMANET. This class stores the name of the component, its own thread reference and its middleware reference and the dynamic list of aspects. It stores two attributes to control whether the component is going to stop or move, as well as the references to the ports to be able to receive and request services. In addition, the *ComponentBase* class offers the following services: *Start* to initiate the component thread execution; *Stop* to stop temporarily the component thread execution; *Abort* to stop definitively the component thread execution; *IsWeaved* to query if an aspect of the component is weaved with another aspect; *AddAspect* and *RemoveAspect* to add and remove aspects from a component; and *AddWeaving* and *RemoveWeaving* to add and remove weavings from a component (see Figure 8).

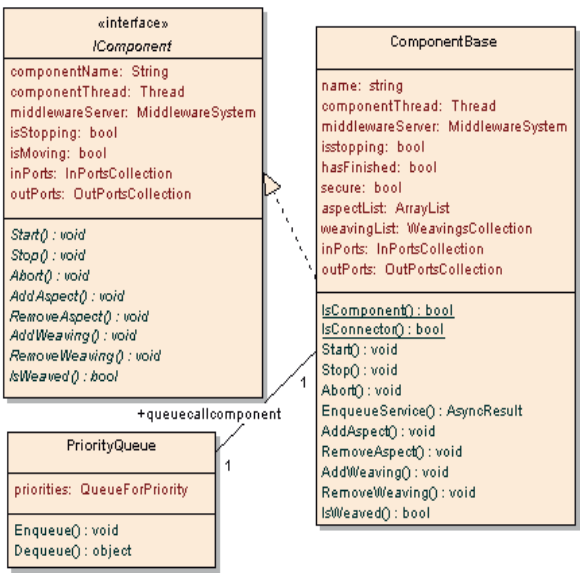


Figure 8. *ComponentBase* class of PRISMANET middleware[1]

3.2.3. Weavings

Weavings have been implemented as a dynamic linked list with three levels of depth. This list is part of the component that it belongs to. Thus, this weaving implementation facilitates the management and evolution of the weavings. The dynamic list is implemented by the *WeavingsCollection* C# class. Each element of this dynamic list is an instance of the *AspectTypeNode* C# class that contains the aspect type and another dynamic list called *weavingAspectList.* Each element of the *WeavingAspectList* is an instance of the *WeavingNode* C# class. This class stores the service name, which triggers the weaving execution as well as, a delegate of this service for its dynamic invocation. It also stores three more lists, each of which belongs to a weaving operator (*after, before, instead).* These lists contain instances of the *WeavingMethod* C# class. This class stores the delegate, which points to the method that must be executed as a result of the weaving (*methodDelegate)*. It also stores the method that has triggered the weaving execution (*origMethod),* and the weaving type (see Figure 9).

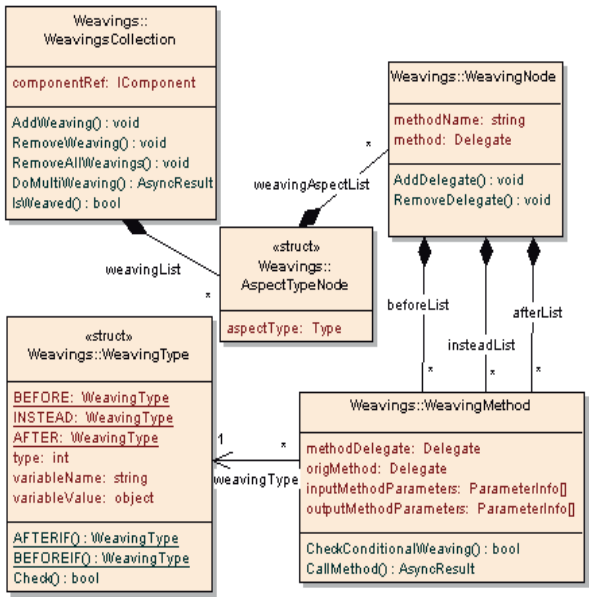


Figure 9. Dynamic list of weavings[1]

As a result, PRISMANET allows the implementation of a specific component by creating a C# class that inherits from the *ComponentBase* class. It is important to keep in mind that components must be *serializable* in order to enable the mobility of components in distributed architectures. An example of a component called *Actuator* is presented below:

```
using System;
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace Robot
{
[Serializable]
 public class Actuator : ComponentBase
 {

   public Actuator(string name,
   MiddlewareSystem middlewareSystem) : base
   (name,middlewareSystem)
   {
   /* ***************************
    * * DEFINITION OF ASPECTS * *
    ***************************/
   // Creation of Functional Aspect
   AddAspect(new FActuator());
   // Creation of Safety Aspect
   AddAspect(new  SMotion(initalMinimum,
                          initialMaximum));

  // Achieving the references of the aspects
   IAspect functionalAspect =
        GetAspect(typeof(FunctionalAspect));
   IAspect safetyAspect =
            GetAspect(typeof(SafetyAspect));

  /* ****************************
   * DEFINITION OF  WEAVINGS * *
   ****************************/
   // Weaving MoveJoint
    AddWeaving(functionalAspect,"MoveJoint",
         WeavingType.AFTERIF("secure",true),
            safetyAspect,"Check",functions);

   /* ***************************
    * * DEFINITION OF PORTS * *
    ***************************/
    InPorts.Add("IMotionJointPort",
                "IMotionJoint",
                 functionalAspect);

    OutPorts.Add("IMotionJointPort",
                 "IMotionJoint");}}}
```

Execution Model

When the execution of a service is requested from a component, the request comes from the port that publishes the service (step 1, Figure 10). The port sends the request to the queue of the component (step 2, Figure 10). Once the component thread extracts the requested service from the queue, the component checks if the requested service has weavings associated to it (step 3, Figure 10). If the service does not have any weavings, its delegate is asynchronously executed so that the component can process another request from the queue. The delegate execution consists of adding the service to the queue of the corresponding aspect. Next, the aspect thread executes the service (step 5, Figure 10). However, if the service has weavings associated to it, before executing step 5, the service is sent to the weaving manager (step 4, Figure 10). The manager processes weavings creating its own thread and freeing the component from this task.

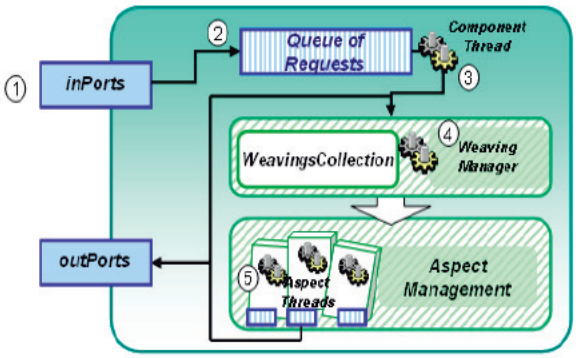


Figure 10. The execution model of a component

With regard to starting or stopping a component, when the middleware calls the *start* service of a component, the component calls the *startAspect* service of each one of its aspects. On the other hand, when the middleware calls the *stop* or *abort* services of a component, the threads of its aspects must also be stopped (*stopAspect)* or aborted (*abortAspect*). In the case of stopping a component in a secure way (*stop* service), a set of operations must be performed in order to achieve a secure state that will permit the start of the component execution in the future. A component is in a secure state when it does not have requests in its aspect queues and there are no executing services. These operations consist of not allowing anymore services in their queues and processing every service that was stored in the queue before the stop execution.

Adding and removing aspects at run-time

Aspects can be added to and removed from a component at run-time. The *addAspect* service inserts a new aspect inside a component. This method verifies that the kind of aspect that is going to be added does not already exist in the component, since only one aspect of each kind can exist in a component. The method updates the references of the aspect to the component and middleware and adds the aspect to the aspect list of the component. Finally, dynamic code generation is used to update the component constructor in order to make the changes consistent. The *removeAspect* service deletes an aspect from a component. First, the method stops the aspect that is going to be removed in a secure way. Second, it removes the aspect from the aspect list of the component and its associated weavings. Finally, the dynamic code generation is used to update the changes.

Distribution Model and Mobility

PRISMANET supports the distributed communication and the mobility of the components. It provides the distributed communication among components without making components aware of each other.

3.5.1. Distributed Communication among elements through Attachments

To make components as reusable as possible, they do not have references to other components they communicate with. Therefore, the components are unaware of the components they communicate with. The distributed communication among components is the responsibility of attachments. Thus, an attachment has the references of the communicating components.

To support attachments, the middleware contains three classes: the *Attachment* class, the *AttachmentServerBase* class and the *AttachmentClientBase* (see Figure 12). For each component port, there is at least one instance of an *Attachment* class. When a component instance is created, the PRISMANET middleware creates the instances of the attachments associated to each port. Each PRISMA port has been implemented into two queues, a client (outPort) and a server queue (inPort) (see Figure 10), there also exists a Server Attachment and a Client Attachment for each Attachment. An instance of the *Attachment* class automatically instantiates an *AttachmentClientBase* and an *AttachmentServerBase* class.

An *AttachmentClientBase* instance has a thread that listens to a specific outport of a component instance. When the *AttachmentClientBase* instance detects that there is a petition in the queue, the petition is redirected to the instance of an *AttachmentServerBase*. Thus, the *AttachmentClientBase* instance has a reference or a proxy of the *AttachmentServerBase*. The *AttachmetServerBase* is a MarshalByRefObject class of the .NET Remoting framework. This has been necessary to create a proxy of the instance to allow the *AttachmentClientBase* instance to access to it remotely.

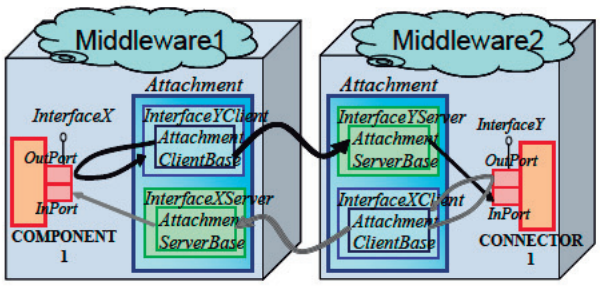


Figure 11. Two distributed architectural elements connected by attachments

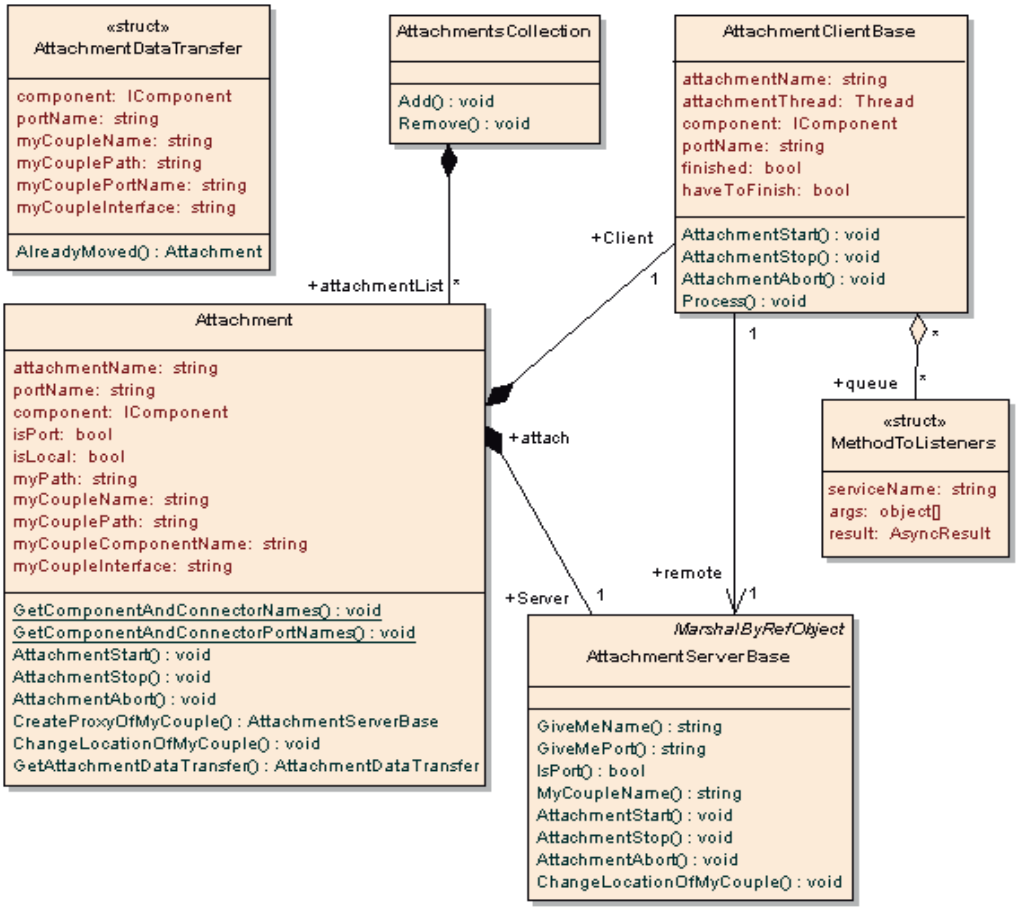


Figure 12. A logical view of the attachments in the middleware[1]

Figure 11, shows how two distributed components are connected together. Component1 has an AttachmentClientBase that listens to its Output queue and redirects services to the AttachmentServerBase of Component2. In addition, Component2 also has an AttacmentClientBase that listens to its Output queue and redirects the services to the AttachmentServerBase. Each AttachmentClientBase and AttachmentServerBase of a component are associated by an Attachment.

The attachments are solely responsible for the distributed communication of the components. The PRISMANET middleware only participates in the creation of attachments instances between its component instances. To store the list of attachments in its site, each middleware has an *AttachmentCollection* class (see Figure 12).

The use of attachments approximation for distributed communication does not only allow for the reusability of the elements but also makes distributed applications independent of a centralized Domain Name Server (DNS). Thus, the attachments of a component can be seen as a distributed DNS that contains the necessary references that allow an instance to perform the needed communications. Our approach prevents the failures which may be generated as a result of failures produced by a centralized DNS such as load saturation and deadlock. In addition, if a certain attachment between two architectural elements fails, their communication among others is not affected.

3.5.2. Mobility of the elements in PRISMA

Mobility is defined as the process of transferring a component instance and its code to a new host. The transferred component instance must continue executing at the new host, while conserving its state and maintaining the same execution point.

Current technologies do not offer this definition of mobility nor does .NET. Therefore, the mobility has to be simulated. To implement the mobility, we have marked all classes of components, aspects and the inPorts and outPort queues of the component with the [Serializable] attribute. The ability to serialize (to pass them by value) is provided by the .NET Framework. However, this is not enough. The mobility process has to ensure that the instance is at a consistent state before it is serialized. The steps to enable a mobility process are presented in the following section.

3.5.2.1. The execution of a mobility decision

A distribution aspect of a component encapsulates the different decisions related to mobility. This enables the reusability of the different mobility decisions in different components. As a result, the component controls the mobility decisions and even if the environment wants to make a mobility decision, it has to go through the distribution aspect of a component.

Figure 13 shows an interaction diagram of part of the mobility process performed at the site where the mobility decision of a component has been executed. It shows the interchange of messages until the component instance is transferred to the RemoteMiddleware.

When a mobility decision is satisfied, the distribution aspect asynchronously calls the PRISMANET middleware on its site to indicate that it is willing to move (*move*, Figure 13). The distribution aspect also notifies its component thread to prepare itself to be in a secure state so that it can be serialized (by executing the *Stop* of a Component, Figure 13). A component is at a secure state when the queues of its aspects are empty and when there is no service being executed. Therefore, the component thread stops processing services from its queue. However, services can be queued in the component inport because the queue is also serializable. In addition, the component thread notifies the aspect threads to stop when they finish processing services from their queues and when they finish executing all the services (*StopAspect*, Figure 13). When the component and aspect threads finally stop, the PRISMANET is notified and it is able to move the component.

3.5.2.2. Preparing to Move Attachments

The attachments also have to be prepared for mobility if a component is moved. This is because the attachments are the communication channels that allow others to communicate with a specific component. Therefore, it is also important to involve the attachments when a component is moved.

When a component instance is completely stopped, PRISMANET executes a service called *PrepareTo-MoveAttachments* (Figure 13). This service fetches the attachments of a component by going through its ports and finding the listeners to these ports. It checks which attachments connect the mobile component with distributed ones. The information associated with each attachment is saved in a structure called the *AttachmentDataTransfer* (see Figure 12). The information contains the references of the *AttachmentServerBase* instances that are connected to the *AttachmentClientBase* instances of the component. It also contains the name of the component instance that is connected to the mobile component instance, and others.

When this task is completed, the middleware stops the thread of the *AttachmentClientBase* instances of the component. In addition, the *AttachmentServer-Base* instances of the distributed component instances, connected to the mobile component instance, are unregistered from Remoting by using the *Disconnect(MarshalByRefObject)* service. This is previously performed in order not to allow the transfer of services to and from the mobile component object.

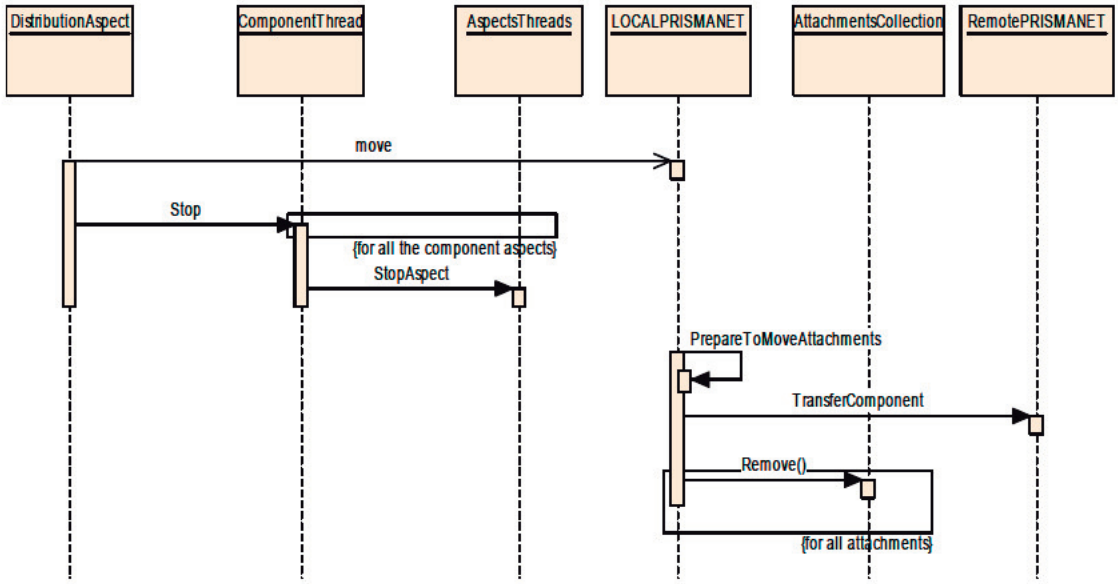


Figure 13. A simple interaction diagram showing the tasks done by at the local middleware site of the transferred component

Finally, a list with all the *AttachmentDataTransfer* structures of a component is created. This is necessary for the new middleware, where the component is going to be transferred, to allow it to recreate the attachments on its site.

3.5.2.3. Transferring component instances

When all the information for the mobility is prepared, the component instance with InPorts and OutPorts is serialized, and the list with the information of the attachments is transferred to the middleware of the new host. The transfer process is performed in a try/catch block in order to recover from any failure that may occur while making the transfer (*TransferComponent*, Figure 13). When the object is correctly serialized, the original component object is destroyed.

In addition, the attachments associated with the mobile component are removed from the list of attachments that exist in the site of the current middleware by executing the *Remove()* method of the *AttachmentsCollection* (*Remove*, Figure 13).

3.5.2.4. Process after transferring a component instance

When the component instance is moved, the receiving middleware updates the list where it stores the components that are executing on its site by adding the component instance moved (*componentList.Add* Figure 14).

Then the middleware uses the information stored in *AttachmentServerBase* structure list in order to create the Attachments of the component instance (*createLocalAttachment*, see Figure 14). However, this is not enough because the instances of the *AttachmentClientBase* of other component instances that are connected to the instances of the *AttachmentServerBase* still have the old references or proxies. Therefore, the new proxies of each *AttachmentServerBase* of the component instance are sent to the connected instances of the *AttachmentClientBase* (*sendNewLocationToCouple*, Figure 14). Afterwards, the thread of each *AttachmentClientBase* of the component instance is started as well as the thread of the component instance.

As the inPorts have not been stopped while the moved component instance was preparing itself to be in a secure state and to be moved, the component instance can start executing the services which where queued at the first middleware and were not processed.

In this way, a mobility approximation has been implemented preserving the state of the object after moving it.

Our approach clearly distinguishes between moving an object and allowing remote calls to it. This can be done thanks to the implementation of the attachments. In .NET Remoting, instances cannot be MarshalByRefObjects and serializable at the same time. As well, even if a MarshalByRefObject is serialized a proxy is created, and it is not the real object that is transferred. Therefore, using attachments the indirect reference remoting to component instances is allowed as well as their mobility.

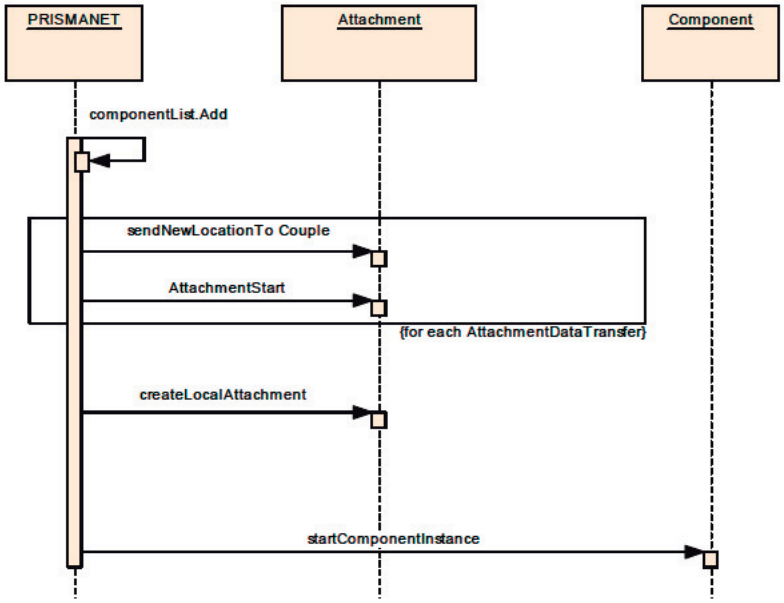


Figure 14. An interaction diagram showing the tasks done by the new middleware site after the component is transferred

4. RELATED WORKS

Currently, there is an increased interest in Aspect-Oriented Programming (AOP) which is becoming a widely used programming technique. AOP was initially developed for Java environments through AspectJ [Kic01] and is being transferred to other platforms such as .NET by means of extensions. However, the existing .NET approaches for supporting AOP are still in an early phase.

AspectC#[Kim02] and SourceWeave.Net [Jac04] support AOP in .NET having available the source code of the base code, the aspects and the weavings. These approaches propose joining the base code with the aspects by specifying the weavings in an XML file. Weave.Net [Laf03] and AspectDNG [Asp05] also define the weavings through an XML file; however, they only use the assemblies of the base code, the aspects and weavings to join the code without being available the source code. Loom.Net [Sch02] is another .NET approach for supporting AOP. It has a graphical interface that allows the addition of defined aspects by means of reusable code templates and allows the performance of weavings.

The approaches mentioned above clearly separate the base code, the aspects and the weavings in different entities. However, none of them supports mechanisms for dynamically adding or removing aspects. The Rapier-Loom.Net [Sch03] approach does allow dynamic addition and removal of aspects, but it defines the weavings inside the aspects thereby losing their reusability. SetPoint [Set05] also allows for dynamic addition and removal of aspects. Its weaving is based on the evaluation of logical predicates in which the base code is marked with meta-information that permits the evaluation of such predicates. EOS [Raj03] is another dynamic approach which is able to attach aspects at instance-level by means of events.

None of the approaches mentioned above takes into account the emerging relations that result from the aggregation of various aspects at the same point of the base code (joinpoint). However, JAsCo.Net [Ver03] provides an expressive language that permits the definition of relations among aspects. JAsCo.Net integrates AOP and CBSD. It introduces the concept of connectors for the weaving between the aspects and the base code which allows for a high level of aspect reusability. An inconvenience of this approach is that the dynamic weaving of aspects to the base code is referential but not inclusive. This requires an execution platform to intercept the application and insert it into the aspects at execution time.

The principal disadvantage of these approaches is that none of them integrates the needed properties at the same time to allow the mobility, the reusability and the evolution of aspect-oriented components. These properties are the dynamic weaving, the join of the base code and the aspects inside the same entity and the reusability of aspects. Therefore, the code mobility is limited because not all the properties of the object code can be moved. However, PRISMA defines a model that combines AOP and the dynamic reconfiguration of the CBSD models. The aspects are separately defined from the weavings and are highly reusable. The components are formed from aspects which are inclusively and can be dynamically aggregated. In addition, PRISMA permits the dynamic mobility of its components, and the concept of base code does not exist, so the component is solely formed by aspects. The implementation of the PRISMA model in .NET permits the dynamic addition and removal of aspects as well as the dynamic modification of the weavings without stopping the execution of the component.

5. CONCLUSIONS AND FURTHER WORK

In this paper, an innovative middleware called PRISMANET has been presented. This middleware is based on PRISMA model and in this way, it allows the implementation of complex, dynamic, distributed, aspect-oriented and component-based software systems using C# language. PRISMANET has been developed with C# language using the standard techniques that the framework provides, that is, without extending the development platform. As a result, PRISMANET can be executed in every computer that has the .NET framework installed.
PRISMANET offers extra functionalities for the .NET platform. It allows the execution of aspects, the reconfiguration of software architectures (local and distributed), the addition and removal of aspects from components at run-time, mobility, etc. As explained in the paper, these functionalities have been implemented using .NET mechanisms such as delegates, reflection, serialization, .NET Remoting and dynamic code generation.

PRIMANET has also been tested in industrial case studies such as the EFTCoR teleoperation system and the *TeachMover* robot. We are now working on the PRISMA model compiler in order to integrate the PRISMA graphical interface and the middleware in a CASE tool and to automatically generate code from the graphical diagrams.

6. ACKNOWLEDGMENTS

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, *DYNAMICA* project TIC2003-07776-C02-02. PRISMANET and industrial case studies developments are funded by the Microsoft Research Cambridge, *"PRISMA: Model Compiler of Aspect-oriented component-based software architectures"* Project [PRI05].

All the class diagrams shown in this work have been automatically generated from the source code of the PRISMANET middleware, by using the Enterprise Architect Modeling tool from Sparx Systems [Spa05]. This tool was selected because it supports the automatic generation of class diagrams from source code, as well as the traceability between the generated classes and the original source code.

7. REFERENCES

[Aos05] Aspect-Oriented Software Development, http://aosd net

[Asp05] AspectDNG Project, http://aspectdng.sourceforge.net

[EFT02] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794, 2002.

[Kic01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. In ECOOP 2001, (Budapest, Hungary), Springer, pp.327-355.

[Kim02] Kim, H., AspectC#: An AOSD implementation for C#. MSc. Thesis, Comp.Sci, Trinity College, Dublin, Dublin, 2002.

[Jac04] Jackson A., Clarke S., SourceWeave.NET: Cross-Language Aspect-Oriented Programming. In Proc. of Generative Programming and Component Engineering (GPCE). Vancouver, Canada, 2004.

[Laf03] Lafferty D., Cahill V., Language-Independent Aspect-Oriented Programming. In Proc. of Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003). Anaheim, California, USA, 2003.

[Mic05] Microsoft .Net Remoting, A Technical Overview, http://msdn microsoft.com/library/default.asp?url=/library/en-/dndotnet/html/hawkremoting.asp

[Per03] Perez J., Ramos I., Jaén J., Letelier P., Navarro E., PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. In proceedings of 3rd IEEE Int. Conf. on Quality Software (QSIC), Dallas, Texas, USA, November 2003, IEEE, pp. 59-66.

[PRI05] PRISMA, http://prisma.dsic.upv.es

[Raj03] Rajan, H., Sullivan, K., Eos: Instance-Level Aspects for Integrated System Design. In the proc. of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Helsinki, Finland, September 2003.

[Rob03] Robinson S. et al., Professional C#, 2nd Edition. Wrox Programmer to Programmer.

[Sch02] Schult, W. and Polze, A., Aspect-Oriented Programming with C# and .NET. In 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (Washington, DC, 2002), IEEE, pp.241-248.

[Sch03] Schult, W. and Polze, A., Speed vs. Memory Usage – An Approach to Deal with Contrary Aspects. In 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS). Boston, Massachusetts, 2003.

[Set05] SetPoint! Project, http://www.dc.uba.ar/people/proyinv/setpoint/

[Spa05] Sparx Systems: Enterprise Architect Modeling Tool, http://www.sparxsystems.com.au

[Szy98] Szyperski C., Component software: beyond object-oriented programming. *ACM Press and Addison Wesley*, New York, USA, 1998.

[Tea05] The *TeachMover* Robot, http://www.questechzone.com/microbot/teachmover.htm

[Ver03] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V., JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services. In Proc. of Second Nordic Conference on Web Services NCWS'03, Vaxjo, Sweden. Published in Mathematical modeling in Physics, Engineering and Cognitive Sciences, Vol. 8, November 2003.

A Conceptual Model for Distributed Aspect-Oriented Software Architectures[1]

Nour Ali, Isidro Ramos, José Ángel Carsi
Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera S/N
E-46022 Valencia Spain
{nourali, iramos, pcarsi}@dsic.upv.es

Abstract

Distribution has become a fundamental requirement for information systems and not an auxiliary. We believe it is important to consider distribution from early stages of the life cycle of software development and deal with distribution at a high abstraction level. This paper, presents a conceptual model for specifying software architectures of distributed information systems. The conceptual model is defined by combining the Aspect Oriented Software Development (AOSD) and the Component Based Software Development (CBSD). The distribution conceptual model is explained by presenting its AOSD perspective and the CBSD perspective. The AOSD perspective of the conceptual model describes the mobile and replication behaviour of an architectural element by introducing a distribution aspect and a replication aspect. The architectural perspective (CBSD perspective) of the distribution model allows specification of the distributed communication of the architectural elements of a software architecture. ***Keywords:*** *distribution, AOSD, software architecture, mobility, replication.*

1. Introduction

Nowadays, information systems are large and complex to develop. An important factor that influences in this complexity is that information systems are tending to be distributed with mobile components. Many technologies deal with distribution issues at an implementation level such as CORBA [1] and .Net Remoting [2]. While distribution of components affects on the decisions taken in the construction of an information system, distribution should also be taken into account at a high abstraction level. Thus introducing distribution at early phases of software development minimizes time and costs instead of only introducing it at implementation.

Moreover, considering distribution at the analysis phase of software development generates high quality distributed applications. Thus applications are prepared to support non-functional requirements analyzing how they affect on the different decisions of software construction. However if distribution is not considered from the beginning, applications should be changed when they arrive to lower phases, loosing in this way the traceability between phases of the life cycle.

The object-oriented approach has provided great improvements in the development of software at early phases of the software life cycle. However, it does not satisfy all the requirements necessary to develop complex systems due to its poor capability of supporting non-functional requirements.

The Component-Based Software Development (CBSD) and the Aspect-Oriented Software Development (AOSD) have emerged, promising to easily develop large and complex systems. The CBSD [3] promises to control the complexity of system construction by coupling entities that provide specific services. The AOSD [4] allows separation of concerns by modularizing crosscutting concerns in a separate entity: the aspect. Aspects can be reused and manipulated independently of the rest of properties of the system.

PRISMA [5] is a conceptual architectural model that combines the AOSD and CBSD to specify

[1] This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, DYNAMICA project TIC2003-07776-C02-02. The development and implementation of this work has been funded by Microsoft Research Cambridge, PRISMA: Model Compiler of Aspect-oriented component-based software architectures Project.

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)
0 7695 2315 3/05 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

Ali, N., Ramos, I., Carsí, J. A. A Conceptual Model for Distributed Aspect-Oriented Software Architectures. Proc. of the International Conference on Information Technology Coding and Computing, ITCC 2005. 2005.

software architectures of software systems. Thus, software architectures of software systems are constructed through the connection of architectural elements. Architectural elements can be components, connectors and systems (a set of connected components and connectors). Each architectural element uses AOSD to describe it's concerns. Thus an architectural element is described by weaving a set of aspect types such as a functional aspect, quality aspect and a coordination aspect.

This paper presents how PRISMA becomes a conceptual model for constructing software architectures of distributed systems by incorporating the necessary distribution primitives to its metamodel. Thus, the conceptual model proposed in this work is defined after an analysis [6] performed to possible approximations for adding distribution to PRISMA.

The paper is organized as follows. Section 2 presents some related work to ours that can be found in the literature. In section 3, we present the distribution model introduced to the PRISMA architectural model and how it combines the AOSD and the CBSD. Finally, section 4 draws some conclusions and highlights some future work.

2. Related Work

Although software architectures arose to simplify the construction of dynamic distributed systems, few Architecture Description Languages (ADLs) have explored the area of distributed systems. The first research that provided significant results in the area of distributed systems was carried out by Kramer and Magee at the Imperial College in London in 1995. This research was done in the Darwin ADL [7]. Darwin uses π-calculus to define the semantics of distributed message-passing. It builds architectures by defining composite components that consist of the binding of instantiated components that are given locations at instantiation time. Darwin has also been used in the CORBA environment to specify the overall architecture of component-based applications [8]. However, there have not been new advances to Darwin in constructing software architectures with mobile and replicable components. Moreover, Darwin only supports constrained dynamic manipulation of the architecture, i.e. the changes must be planned.

The work by Virginia C. de Paula [9] introduces features that an ADL should consider in order to specify dynamic architectures such as composition, reusability and configuration. The work presents a formal configuration language that describes a method for a reconfiguration model at run-time. However, the reconfiguration model is not formal and it neglects a distribution model for specifying distributed message-passing among components and connectors.

In the works of Mascolo and Ciancarini [10] [11], introduced MobiS a specification language that is based on a tuple-space based model which specifies coordination by multiset rewriting. The works show that MobiS can also be used to specify architectures containing mobile components. However, these works do not deal with the mobility concerns separately from the rest of the functionalities of the software architectures thereby losing reusability and proper adaptability to changes.

The works of Medvidovic in the ADL C2Sadel [12] adapts a style to support distribution and mobility. This style provides software connectors that are able to move components. They exploit the modeling and implementation infrastructure to support distributed and heterogeneous applications. However, this approach has the drawback that there is no separation between coordination and distribution. The components are the only architectural elements that are mobile while the connectors are static.

The recent works of Fiadeiro [13] on supporting distribution and mobility which describe the semantics of externalizing a distribution dimension. This distribution dimension is very similar to a connector. However, instead of containing the business logic, it controls the rules for mobility and locations. This achieves a separation between computation, coordination and distribution. Our approach is quite similar, however we use the aspect oriented approach which encapsulates distribution issues of components in entities called aspects. We use the aspect-oriented approach because it supports the independence of the aspects from the functionalities thereby, achieving reusability, adaptability and evolution. In addition, it is a considerable mature and tested approach at the implementation level.

The rise of aspect-oriented programming languages introduced the encapsulation of the distribution code in the distribution aspect separating it from the rest of the code. This has been done by extending well known programming languages as AspectJ [14, 15] in the case of Java. Whilst an inconvenient of these low level languages is the definition of the pointcuts inside the aspects decreasing reusability. Therefore, to solve these issues the aspect-oriented software development approach emerged. It incorporates aspect concepts at early stages of development, as in the architectural and analysis phases, and supports code generation to AspectJ.

In works as in [16], distribution and replication aspects are introduced at analysis and design phases. However, distribution issues as load balancing, caching, migration are not contemplated. Moreover, it

IEEE COMPUTER SOCIETY

does not use a general purpose language (i.e. the language used to specify the distribution behaviour is different from the one used to specify the functional behaviour of the architecture). In fact, when describing the functional entity (the object) Java is used, whereas for defining the distribution aspect another language is used and for composing the aspect with the object a third language is used.

3. Distribution Model for PRISMA

This section presents the distribution conceptual model of PRISMA. The distribution model combines the AOSD and the CBSD according to the PRISMA architectural model. The distribution conceptual model is explained by presenting the AOSD perspective presenting the distribution aspect and the replication aspect introduced to the PRISMA meta-model. Afterwards the architectural perspective (CBSD perspective) of the distribution model is shown.

3.1. AOSD Perspective of the Distribution Model of PRISMA

An aspect is a crosscutting concern i.e. a concern that is present in many components of the information system. Distribution is a clear crosscutting concern in a distributed system. Each distributed component has its properties that enable it to be distributed. Replication properties also crosscut different architectural elements of a software architecture. For this reason, another important aspect for large and complex distributed systems is replication which for instance can be used for solving fault tolerance problems. Therefore, in PRISMA a distribution and a replication aspect are defined in order to support the distribution and replication properties of an architectural element.

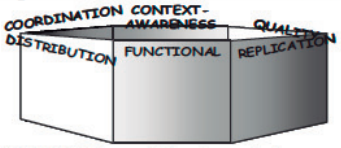


Figure 1 A PRISMA architectural element AOSD view with a distribution and replication aspect.

3.1.1. The Distribution Aspect.

A PRISMA distribution aspect should be added to the set of aspects types of the model in order to enable the definition of software architectures of distributed systems (see Figure 1). The distribution aspect specifies the features and strategies that manage the dynamic location of the instances of the architectural elements of a software architecture. Therefore, if the system is local a distribution aspect is not added to the set of aspects of the architectural element.

The distribution aspect deals with all the properties related to distribution and changes in location. To define the distribution aspect the general concepts of the PRISMA aspects are used. The distribution characteristics are defined through the *Attributes* and *Services*. The change of state of the architectural element is specified through the *Valuations* of the services. The restrictions associated to these changes are defined using *constraints* and *preconditions*. The consequences following certain conditions are specified by *triggers*.

The distribution aspect has to be added to the set of aspects of the meta-model. The distribution aspect includes both the general concepts of the meta-class *Aspect* (see Figure 2) and proper concepts. Therefore, a *Distribution* meta-class is defined which inherits from the meta-class *Aspect* of the meta-model.

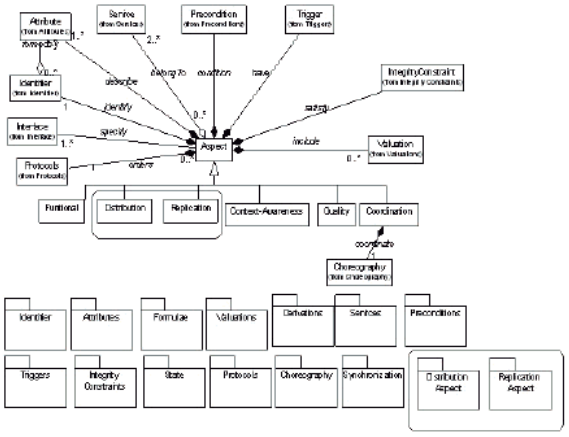


Figure 2 The Distribution and Replication aspect added to the Aspect Package of the PRISMA meta-model

Each architectural element with a distribution aspect must have a **location**. The **location** is a predefined PRISMA attribute of the distribution aspect that has as its abstract data type (domain) **loc**. This data type models the sites in which architectural elements of an architecture domain are located. Therefore, the values of a location attribute should be one of the values of the **loc** data type designed in the architecture domain. In this way, PRISMA can remain independent of any notation of locations (nodes) and can be used for designing different kinds of location mechanisms. For example, the location can be an http address or an IP address of a LAN network.

The distribution aspect can specify the ability of an architectural element to be mobile or the ability of an architectural element to affect on the locality of an

external architectural element through the PRISMA service **move**. Therefore, if a distribution aspect does not have the PRISMA service **move** it means that it is not mobile nor it can affect on the locality of other architectural elements of the architecture. The PRISMA service move can be **in**, **out** or both. If the move is an **in**, the instance of the architectural element type can be moved. If the move is an **out** then the architectural element instance which it is formed by the distribution aspect can move another architectural element of the software architecture.

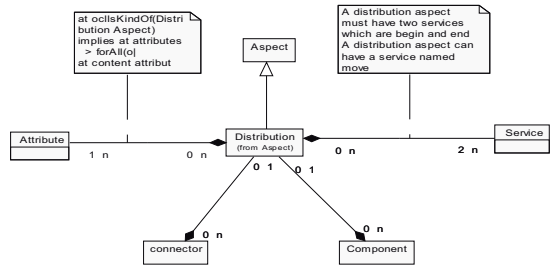


Figure 3 Distribution Aspect package of the PRISMA meta-model

In Figure 3, the package of the distribution aspect is shown. The distribution aspect is directly related with the meta-class *Attribute*, the meta-class *Service* and the meta-class *Component* and *Connector*. A distribution aspect must have one to many Attributes (1..n). This is due to the fact that a distribution aspect must have at least an attribute named *location* with **loc** as a data type. This is specified as a constraint using Object Constraint Language (OCL) [17]. Moreover, the distribution aspect must have at least two Services (2..n). This is because the initializing **begin** service and the finalization **end** service must exist. In addition, a proper distribution service **move** can optionally exist depending on the mobility requirements. In order to specify that the services **begin**, **end** and **move** are services of the distribution aspect a constraint is defined. A component or connector can have a distribution aspect depending if we are working on a distributed system or not. Therefore the cardinality of the aggregation between the meta-class distribution and the meta-class component is (0..1). The same applies to the connector. In addition, only a distribution aspect can exist for each component or connector.

3.1.2. The Replication Aspect.

Many models consider replication as the sequence of a copy and a move operation without considering it as a primitive. However, in our proposal we consider to take replication as a primitive and separate the replication properties from those of distribution. In this way, we achieve a higher level of modularization and maintenance such that we can change the replication properties without affecting on those of distribution and the distribution requirements can be tested without those of replication. For this reason, the replication can be independent of the distribution and mobility. Thus, architectural elements can be replicated independently of being in a distributed system or not.

The replication aspect, as the distribution aspect, is also included in the PRISMA meta-model by inheriting from the meta-class aspect (see Figure 2). The replication aspect as the distribution aspect uses the general concepts of an aspect to specify all the properties related to the replication of an architectural element. The replication characteristics are defined through the aspects *Attributes* and *Services*. The change of state of the replication aspect is specified through the *Valuations*. The restrictions associated to these changes are defined using *Constraints* and *Preconditions*. The consequences fulfilling certain conditions are produced by *triggers*.

The replication aspect must have the PRISMA service **replicate**. The replicate service has a replica's location as an argument therefore its domain should be of type **loc**. If the service **replicate** is an **in** then the aspect specifies that its architectural element can be replicated. If the service **replicate** is an **out** then the aspect specifies that the architectural element of the aspect can cause the replication of another architectural element of the architecture.

In Figure 4, the package of the replication aspect is shown. The replication aspect is related with three meta-classes: *Service*, *Component* and *Connector*. The aggregation between the meta-class *Service* and *Replication* has the cardinality (0..n-3..n). This is to indicate that a replication aspect must have at least three services: the **begin** service, the **end** service and the **replicate** service. To constraint the existence of these three services a constraint is defined. The aggregation relation between the meta-class *Replication* and the meta-classes *Component* and *Connector* have the cardinality (0..n-0..1). This is due to the fact that the replication aspect is an optional aspect for the architectural elements. Only a replication aspect can exist for each component or connector.

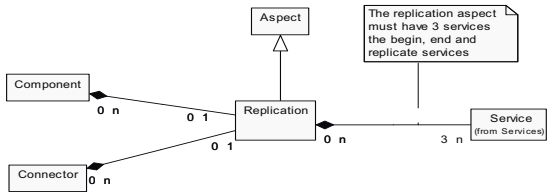


Figure 4 Replication Aspect package of the PRISMA meta-model

3.2. The architectural Perspective of Distribution

IEEE COMPUTER SOCIETY

An architectural model in PRISMA is built by attaching the instances of architectural elements through relationships called the **attachments.** The architectural elements can be simple (components and connectors) or complex architectural elements (systems). Systems are complex because they aggregate components, connectors and other systems through **attachments**. A system is connected with the elements it encapsulates by relationships called **bindings**.

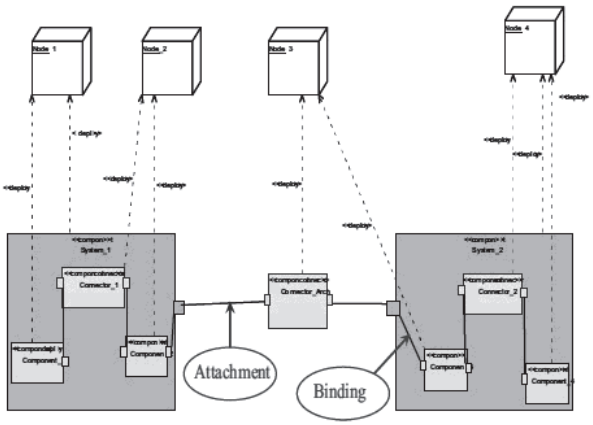


Figure 5 A distributed architectural model in PRISMA connected with attachments and binding links

As the attachments and bindings define the connections among the instances of the architectural elements, they can also define the connections between the locations of the connected instances. As a consequence, the attachments and binding links can be seen as the software artefacts in which architectural elements instances are registered once instantiated and given a location value in order to publish their services, that is, in order to offer and provide their services remotely to other instances of the architectural model. Therefore, the attachments and binding links have to be location aware of the architectural elements they connect.

In Figure 5, an example of a distributed architectural model is shown. There are 4 nodes (locations): *Node 1* where *component 1* and *system 1* are deployed, *Node 2* where *connector 1* and *component 2* are deployed, *Node 3* where *connector Arch* and *component 3* are deployed on and *connector 2*, *component 4* and *system 2* are deployed on *Node 4*.

The location attribute of the distribution package has been included in the attachments package (see Figure 6). So the location of the two connected architectural elements is added to the attachments. Moreover, the locations of the system and the element it encapsulates are also appended to the bindings package (see Figure 7).

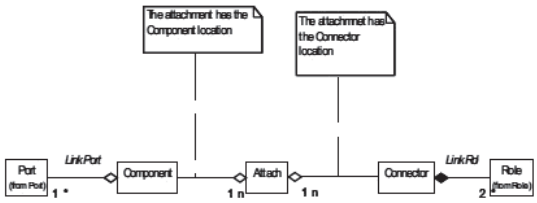


Figure 6 The attachments package of the PRISMA meta-model

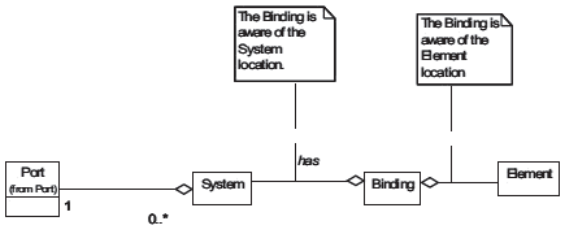


Figure 7 The bindings package the PRISMA meta-model

The attachments and the systems bindings form a distributed Domain Name Service (DNS) when performing distributed communication in the PRISMA model. This optimizes the performance and scalability of distributed systems specified in PRISMA due to the fact that problems resulting from having a centralized DNS (such as a bottle-neck effect) are reduced by balancing the load between the distributed DNSs. At the same time, this provides a description mechanism to establish location hierarchies which is crucial for mobile and ubiquitous systems.

The location hierarchies in PRISMA are easily described due to the fact that architectural models are hierarchical. Such that bindings represent hierarchical connections (connect an architectural element from a lower level of aggregation with the system which is a composite component) and attachments are the non-hierarchical connections. As a consequence, the locations of PRISMA software architectures can be represented in a tree-like structure. The root of the tree is the architectural model and its leafs represent the locations of the direct elements (systems, components or connectors) it contains. If the element is a system then its direct leafs are the locations of the architectural elements that it encapsulates (see Figure 8).

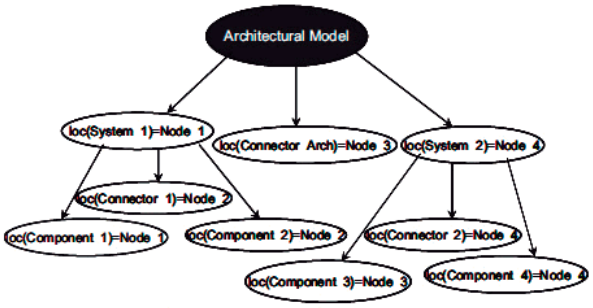


Figure 8 Representing the locations of the architectural model example of Figure 5 hierarchically.

4. Conclusions and Further Work

This paper has presented a conceptual model for describing software architectures of distributed systems. Primitives for mobility, replication and distributed communication have been defined. The conceptual model has been defined by combining the AOSD and the CBSD.

We have implemented this model in C#.Net using .Net Remoting. We also have used the teleoperation systems as a case study to apply our model.

In the medium term, we are building a model compiler to automatically generate distributed applications. A case tool is going to be developed to incorporate textual and graphical notation of PRISMA and distributed applications into different platforms and programming languages.

5. References

[1] CORBA Official Web Site of the OMG Group, http://www.corba.org/

[2] Microsoft .Net Remoting: A Technical Overview, http://msdn.microsoft.com/library/default.asp?url /library/en us/dndotnet/html/hawkremoting.asp

[3] Szyperski, C., Component software: beyond object oriented programming, ACM Press and Addison Wesley, New York, USA, 2002.

[4] Aspect Oriented Software Development, http://aosd.net

[5] J. Pérez,, I. Ramos, J. Jaén, P. Letelier, E. Navarro. "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", Proceedings of 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 2003, p 59 66.

[6] N. Ali, J.A. Carsi, I. Ramos. Analysis of a Distribution Dimension for PRISMA. Proceedings IX Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2004, ISBN 84 688 8983 0, Malaga, Spain, 2004, p 467 474.

[7] Magee, J., Dulay, N., Eisenbach S. and Krammer, J. "Specifying Distributed Software Architectures". Proceedings of the 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, 1995, p 137 153.

[8] Magee, J., Tseng, A., Kramer, J. "Composing Distributed Objects in CORBA", Proceedings of the Third International Symposium on Autonomous Decentralized Systems, Berlin Germany, 1997, p 257 263.

[9] Virginia C. de Paula, G. R. Ribeiro Justo and P. R. F. Cunha, "Specifying Dynamic Distributed Software Architectures", XII Brazilian Symposium on Software Engineering, BCS Press, October, 1998.

[10] Ciancarini, P., Mascolo, C. "Software Architecture and Mobility". Proceedings of 3rd Int. Software Architecture Workshop (ISAW 3), November, 1998.

[11] Mascolo, C. "MobiS: A Specification Language for Mobile Systems". Proceedings of the 3rd International Conference on Coordination Models and Languages, 1999.

[12] Medvidovic, N. and Rakic, M. "Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility". Proceedings of the Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.

[13] Lopes, A., Fiadeiro J.L. and Wermelinger, M. "Architectural Primitives for Distribution and Mobility", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, p 41 50.

[14] Soares, S. and Borba, P. "PaDA: A Pattern for Distribution Aspects". Proceedings of the Second Latin American Conference on Pattern Languages Programming SugarLoafPLoP, Itaipava, Rio de Janeiro, Brazil, 2002, p 87 99.

[15] Soares, S., Laureano, E. and Borba, P. "Implementing Distribution and Persistence Aspects with AspectJ". Proceedings of the 17th ACM Conference on Object Oriented programming systems, languages, and applications, OOPSLA'02, Seattle, WA, USA,2002, p 174 190.

[16] Herrero, J.L. Proposal of a Platform, Language and Design, for the Development of Aspect Oriented Applications. (In Spanish) P.h.d Thesis, Extremadura, Spain, 2003.

[17] J. Warmer. *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition,* Addison Wesley Pub Co, 29 August, 2003.

Separación de "Crosscutting Concerns" en el Desarrollo Orientado a Aspectos a Nivel de Diseño Arquitectónico*

Rogelio N. Limón Cordero*, José Torres Jiménez[O], y Isidro Ramos Salavert*

* Universidad Politécnica de Valencia,
Departamento de Sistemas y Computación
[O]Centro Nacional de Investigación y Desarrollo Tecnológico,
Departamento de Computación
E-mail: limon@itoaxaca.edu.mx, jose.torres.jimenez@orgacm.org, iramos@dsic.upv.es

Resumen

Actualmente el desarrollo orientado a aspectos es un paradigma novedoso en la ingeniería de software, principalmente en lo que se refiere a la separación de los "*crosscutting concerns*". La ventaja de identificar y separar los *crosscutting concerns* en las fases iniciales del desarrollo del software permite conciliar intereses entre los usuarios y los diseñadores. En este artículo se propone un método para realizar una descomposición modular descendente e iterativa de los *concerns* que permite detectar y separar los *crosscutting concerns*, y posibilita la obtención de las vistas arquitectónicas más importantes.

1. Introducción

El desarrollo orientado a aspectos (AOSD: Aspect Oriented Software Development) es un paradigma novedoso dirigido a mejorar la separación de los asuntos o materias de interés conocidos como *concerns* [6], los *crosscutting-concerns* son *concerns* que al momento de diseñarlos o implementarlos se llegan a expandir a través de varias unidades modulares (atravesando los límites de dichas unidades *crosscutting*). Esta situación complica las actividades de desarrollo, evolución y mantenimiento del software. Las unidades modulares ocupadas para contener a los *crosscuting concersn* son conocidas como *aspectos* [6].

Con relación a la arquitectura los *aspectos* involucrados son los llamados *aspectos tempranos* [10] que posibilitan la toma de decisiones en las primeras etapas del desarrollo del software. Sin embargo las técnicas de separación de *concerns* apenas comienzan a ser usadas de manera extensiva (aunque no siempre de manera efectiva) en los niveles de abstracción de determinación de requisitos y elaboración del diseño [10].

El método aquí propuesto proporciona un mecanismo de separación de *concerns* a nivel de arquitectónico para cubrir la necesidad de manejar a los aspectos tempranos del software [3]. Primero se hace una propuesta de la meta-estructura arquitectónica que permite expresar la arquitectura mediante vistas estructurales y de componentes-conectores. En segundo término se

* Este trabajo ha sido financiado por el Proyecto CICYT PRISMA

plantea un método que tiene como entradas los requerimientos funcionales, los requerimientos no funcionales y los escenarios. En tercer término se plantea una propuesta preliminar para la arquitectura utilizando patrones arquitectónicos, esta propuesta se va refinando mediante un proceso iterativo de descomposición modular descendente y de composición, esto permite separar los *concerns* y los *aspectos*. En último término se obtiene la arquitectura aspectual del sistema expresada a través de sus vistas.

2. La Separación de Concerns

Los *concerns*, de acuerdo a [8] son "…aquellos intereses que pertenecen al desarrollo del sistema, sus operaciones o algunos otros aspectos que son críticos o de alguna manera importantes para uno o más actores involucrados…". La tarea de separación de *concerns* tiene como intención reducir la complejidad de un sistema de software durante todo el ciclo de vida de su desarrollo (mejorando la compresibilidad, su administración, promoviendo la reutilización y facilitando su evolución). La descomposición ha sido una de los elementos claves para la separación de *concerns*, sin embargo metodologías como la estructural y la orientada a objetos han centrado dicha descomposición en un solo *concern* dominante (función o clase) lo cual no ha permitido una completa y adecuada separación de los *concerns*. de ahí que hayan surgido trabajos como el de [11] que pretendan una descomposición multidimensional separando a los *concerns* mediante una mecanismo llamado hiper-espacios dando libertad para identificar y separar diferentes tipos de artefactos de software. Con la aparición de la AOP (Aspect Oriented Programming) se intensificó el tratamiento de los *concerns* a nivel de programación [6], donde los *crosscutting-concerns* se limitaron a *concerns* propios de la implementación. Sólo recientemente se ha planteado una estrategia de descomposición descendente [5]. Contribuciones a nivel de proceso del AOSD (Aspect Oriented Software Development) proponen una metodología novedosa para separar y representar aspectos combinando su representación con UML sin involucrarse a nivel arquitectural [1], en [3] se trabaja a nivel arquitectural pero sin usar el contexto de un proceso.

El método propuesto en este artículo es parte inicial del proceso AOSD, usa una estrategia de descomposición descendente, trabaja a nivel arquitectural y usa la notación UML 2.0.

3. Especificación de la Arquitectura de Software Aspectual Propuesta

La arquitectura de software (AS) considerada en [2] como el conjunto de estructuras del sistema, comprende componentes de software, propiedades externamente visibles de estos componentes e interrelaciones entre ellos a través de conectores. La arquitectura puede ser representada ya sea en forma esquemática o mediante algún lenguaje de descripción arquitectural (ADL: Architecural Description Language) para fines de análisis y evaluación.

Dentro del proceso de desarrollo del software la AS se sitúa entre los requerimientos y el diseño, y sirve como un puente entre estas dos áreas, permitiendo tomar decisiones tempranas que rigen al diseño e implementación de ahí que en [2] sea considerada como la estructura de estructuras.

En el AOSD la AS tiene una doble contribución: a) abordar los aspectos tempranos enlazando los aspectos que se presentan en los requerimientos con los que se manifiestan en el diseño, i.e. aspectos de una fase que pueden repercutir en otra; y b) abordar los propios aspectos de la AS y los posibles efectos de los aspectos de nivel superior sobre los aspectos de niveles inferiores [3]. Para poder contener los aspectos a nivel arquitectural se tiene que construir una estructura que permita representarlos y manejarlos.

Existen por lo menos tres estrategias a nivel de la AOP [6]: a) uso de un sistema reflexivo que incorpora estructuras para representarse a sí mismas; b) uso de un modelo basado en patrones de código; y c) uso de filtros de composición. El método propuesto usa la primer estrategia extendiendo el modelo propuesto en [9].

La figura 1 muestra las meta-estructuras propuestas fundamentadas en el modelo de descripción arquitectónico de [8], en este caso representada mediante las vista modular y la vista de componentes-conectores [2], La primer vista indica la estructura estática que contienen los tipos de módulos representadas por los estereotipos: funcionales y aspectuales. La segunda vista indica el dinamismo que se tiene a nivel de componentes, son la realización de los módulos de la vista estática, incorporando al conector para realizar la interacción entre los mismos módulos y de éstos con los *aspectos*.

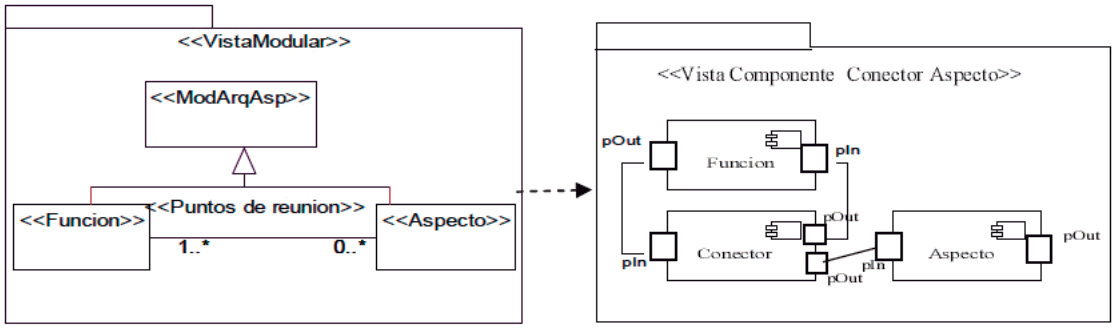


Figura 1. Meta-Estructuras para las Dos Vistas Propuestas para una Arquitectura Aspectual y su Relación (Notación: UML 2.0).

El propósito de cada elemento de la vista-modular es el de describir los requerimientos funcionales, los módulos de esta vista pueden aceptar relaciones propias de esa estructura: descomposición ("es un sub-módulo de"), usos, capas y generalización [3]. Los módulos de *aspectos* se ocupan como contenedores de los *crosscuting concerns* los que se vinculan con las funciones por medio de una relación de prototipo *puntos de reunión*, cuya correspondencia (o cardinalidad) indica que puede haber módulos funcionales que no tengan que ver con ningún *aspecto*, pero un *aspecto* por lo menos debe de relacionarse con un módulo funcional.

La vista componente conector aspecto (C-C-A), está representada por tres clases de componentes: función, conector y aspecto. Para la especificación conector-función se usó la estrategia tercera propuesta en [7] con base a la notación de UML 2.0, en esta estrategia se usa un componente tipo conector para vincular a las funciones a través de sus puertos de entrada (pIn) y salida (pOut), de lo cual se hizo una extensión para incorporar a los aspectos representados por

medio de la clase *aspecto* que se vincula con las funciones por medio de los conectores. Esta última clase de componente es la realización de la relación de la asociación *puntos de reunión* de la vista modular, que de acuerdo a su cardinalidad puede tener tantos puertos como sean necesarios dependiendo de los componentes y *aspectos* que relacione.

4. Método Planteado para la Separación de Concerns a Nivel Arquitectónico

Las características relevantes del método se fundamentan en tres puntos: en la estrategia propuesta para la descomposición con base a la meta-estructura arquitectónica aspectual descrita anteriormente, en el uso de patrones arquitectónicos, y en el empleo de herramientas de análisis de comportamiento. Los dos últimos son tomados de las contribuciones que otros autores han hecho al respecto.

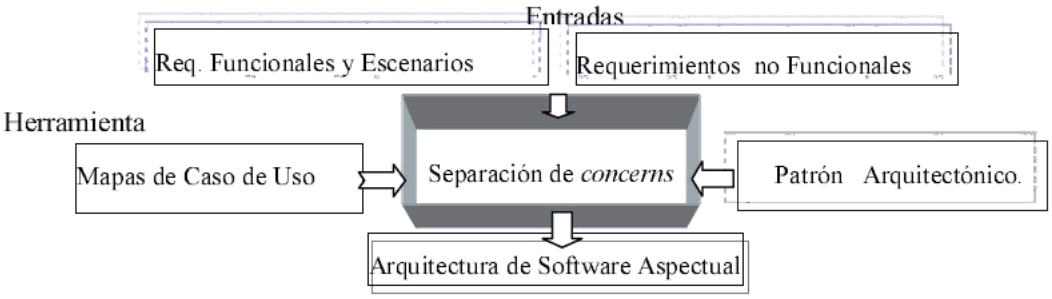


Figura 2. Elementos que Participan en la Separación de Concerns.

Los principales elementos que intervienen en el método se muestran en la figura 2. Las entradas proceden de los usuarios que aportan los requerimientos y escenarios y de los diseñadores que aportan los requerimientos no funcionales. En este trabajo los requerimientos funcionales y los escenarios pueden ser expresados en forma textual o por casos de usos.

La herramienta de mapas de caso ayuda a encontrar los *crosscutting concerns*, a especificar el comportamiento de los módulos, y facilitan la compresión del comportamiento de sistemas complejos [4].

El uso de patrones arquitectónicos posibilita la reutilización de una arquitectura [2], simplificando y agilizando su diseño. Se han identificado algunos patrones genéricos dentro del área llamados estilos arquitectónicos como tuberías y filtros, capas, flujo de datos, cliente servidor y pizarra, los cuales sirven de base para la arquitectura de muchos sistemas.

La separación de *concerns* propuesta consiste en un proceso de descomposición iterativo descendente que permite una división funcional del sistema estableciéndo relaciones de jerarquía, de asociación y de composición entre los módulos, vinculando a las funciones con los aspectos. El resultado final de este proceso consiste en la arquitectura aspectual del sistema en sus vistas modular y de componentes-conectores expresada en notación de UML 2.0.

En la figura 3 se muestra la secuencia de actividades implicadas en el método y en la siguiente sección se describen e ilustran a través del ejemplo propuesto.

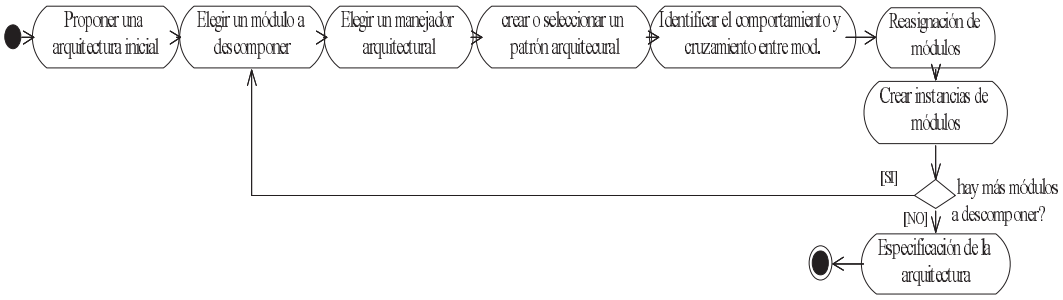


Figura 3. Actividades del método de descomposición de concerns.

5. Despliegue del Método

5.1. Caso de Ejemplo

Se pretende diseñar un sistema para la reservación y venta de boletos de pasajeros que comprende a diversas líneas de autobuses que recorren varias ciudades. Un presunto pasajero puede hacer su itinerario, eligiendo origen, destino, línea de autobús, fecha y horario, de acuerdo a la disponibilidad de espacio, a través de una pagina web o bien comprar el boleto en cuyo caso debe poseer una tarjeta de crédito. También puede comprarlo directamente en taquilla pudiendo pagar también en efectivo. Por sencillez consideremos a un requerimiento no funcional "la seguridad", la cual involucra tanto al acceso por los usuarios, posibles pasajeros y encargados de taquilla, como al acceso a los bancos para realizar consulta y cargo correspondiente en caso de confirmar la compra de boletos

5.2. Entradas del Método

Las entradas del método constituyen los requerimientos y los escenarios, los primeros están descritos en la Fig 4 como casos de uso, donde se incluye al requerimento no funcional REGS/AUTEN correspondiente al de seguridad. En cuanto a los escenarios solo se consieran a dos, representados en la tabla 1.

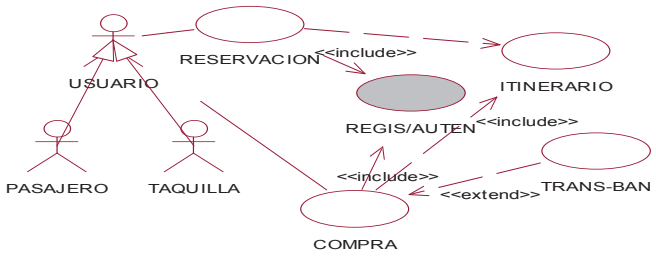


Figura 4. Caso de Uso: Reservación-Compra de Boletos de Pasaje de Autobús.

TABLA 1. ESCENARIOS CONSIDERADOS DEL EJEMPLO.

ESCENARIO	
Compra de boleto	Crear itinerario/Registrar y autentificar a usuario/Realizar pago/Validar/reservar/imp. boleto
Reservación de boleto	Crear Itinerarios/Registrar y autentificar a usuario/Reservar asiento-camión

5.3. Actividades del Método

Proponer una arquitectura inicial: Como punto de partida se hará un boceto de una posible arquitectura de acuerdo a las especificaciones de los requerimientos funcionales y no funcionales al más alto nivel. Para el caso del ejemplo pueden ser solo los módulos que se tienen a nivel del paquete estereotipo <<función>> reservacion, compra. También se incluye el requerimiento no funcional de registro autentificación (reg/auten), haste este punto son los primeros *concerns* de interes detectados.

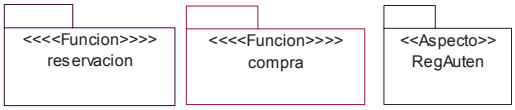


Figura 5. Arquitectura Inicial.

Elegir un módulo a descomponer: En el primer caso se selecciona cualquiera de los módulos de la arquitectura propuesta inicialmente puesto que solo están a nivel de paquete. Para los casos siguientes se hace en función de la prioridad mayor de los requerimientos que aún no se hayan satisfecho. Para nuestro ejemplo se elige tanto a reservar y comprar, y tienen la misma prioridad.

Elegir un manejador arquitectural: Aquí se considera a un manejador arquitectural de acuerdo a la definición dada en [2]. "*..es la combinación de requerimientos funcionales y de calidad que forman la arquitectura de un módulo considerado...*" Esto hace necesario priorizar los requerimientos específicos para el módulo. Para el ejemplo en cuestión considere los requerimiento asociado con la compra y reservación del boleto, cuyos escenarios se muestran en la tabla 1, los dos son considerados como manejadores arquitecturales.

Crear o seleccionar un patrón arquitectural: Determinar si existe un patrón arquitectónico que cubra o se ajuste a las expectativas del manejador arquitectónico. El cual servirá de base para refinar el módulo. Para el ejemplo de los manejadores arquitecturales mostrados se puede usar un patrón cliente-servidor entre reservar y comprar boletos con respecto a itinerario, en donde los primeros son los clientes y el segundo el servidor. Ahora se puede descomponer el módulo compra y reservación boletos en los sub-módulos (clases) Compra y Reservación y se crea una relación de <<uso>> con la clase Itinerario.como se aprecia en la figura 6. Por su parte clase Compra se realiza(<<realice>>) mediante la clase TransBan.



Figura 6. Descomposición a Nivel de Módulo y Aplicación de un Patrón.

Identificar el comportamiento y cruzamiento entre módulos: Para poder hacer esto se emplean los mapas de casos de uso con los escenarios especificados y los módulos hijos identificados.

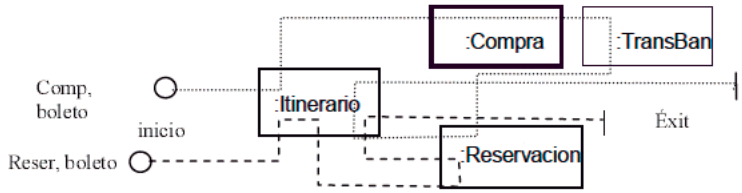


Figura 7. Mapas de Caso de Uso para los Escenarios Reservación y Compra de Boletos

De este análisis se detecta un *crosscutting* para los dos escenarios descritos, ya que el *concern* Intinerario está esparcido en dos módulos, éste es tocado más de una vez por los escenarios vinculándose con clases de diferentes unidades modulares, por lo que se convierte en un *crosscutting concern* , lo cual hace que los módulos arquitectónicos en cuestión tengan que ser reestructurados.

Reasignación de módulos: Habiendo detectado los módulos que presentan cruzamiento se reasignan las clases causantes de estos y se refina la relación entre ellos. Para el ejemplo, la clase Itinerario se reasigna a un módulo <<aspect>>, como se puede ver en la figura 8.a.

Crear instancias de módulos: Una vez que se han identificado las relaciones entre los módulos se procede a crear instancias de éstos a nivel de componentes. La interconexión se deriva de la relación y se lleva a cabo mediante un elemento conector en este caso llamado sesión. Esto se ilustra en nuestro ejemplo en la figura 8.

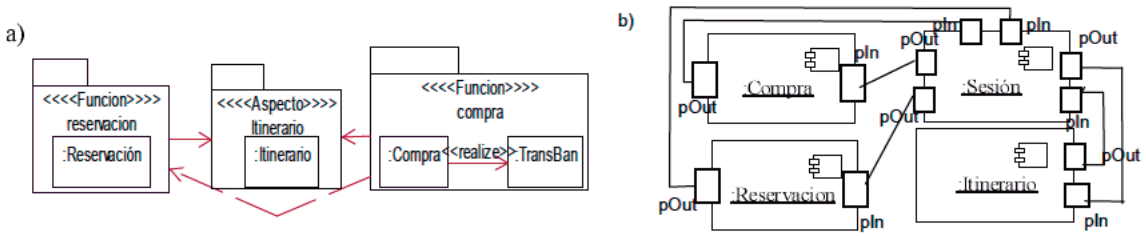


Figura 8. Arquitectura Reestructurada: a) Vista Modular b) Vista de Componente-Conector.

El proceso continúa repitiendo las actividades antes indicadas (salvo la primera) hasta que se tienen todos los módulos refinados, como la mostrada para el ejemplo en la figura 9.

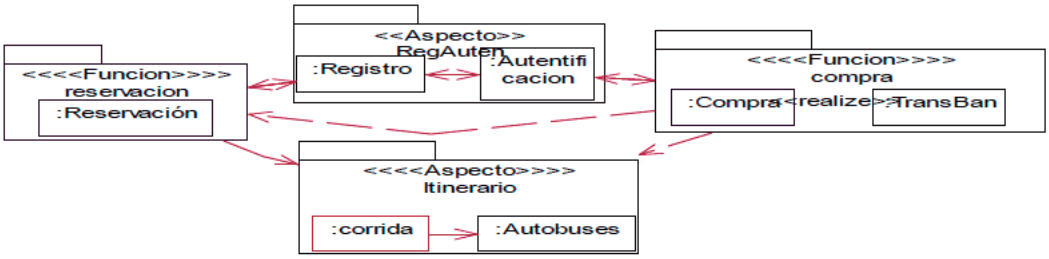


Figura 9. Vista Modular de la Arquitectura Resultante para la Compra-Venta de Boletos de Autobuses.

6. Conclusiones

Al abordar a los aspectos a nivel arquitectónico se contribuye a la consolidación del proceso de desarrollo orientado a aspectos en sus fases iniciales. Con el método propuesto no solo se consigue separar a los *concerns* y a los *crosscutting concerns* , sino que se logra enfrentar la complejidad dentro del espacio del problema mediante una estrategia de descomposición modular descendente e iterativa que a la vez va conformando las vistas de arquitectura modular y de componente conector, guiando la construcción del diseño detallado y a la implementación. La estructura de las vistas arquitectónicas aspectual que se elaboró posibilita la separación de los *crosscutting concerns* en forma estructural al mismo tiempo que describe su comportamiento para un análisis dinámico de la arquitectura El método planteado consigue varias aportaciones: posibilita el uso de patrones que hacen a la arquitectura reutilizable, lo cual redunda en una mayor rapidez en su diseño; aprovecha las ventajas de la descomposición descendente e iterativa haciendo una separación multi-dimensional, y espera ser un eslabón más dentro de la cadena de tareas del AOSD.

La separación de *concerns* a un nivel arquitectónico es un primer paso para la conformación de un proceso integral en el desarrollo orientado aspectos, aunque se reconoce que todavía hace falta consolidar esta propuesta mediante mecanismos que permitan su automatización por un lado y por otro vincularlo con las técnicas que se manejan a niveles de implementación.

Referencias

[1] Baniassad, E. and Clarke, S., Theme: An Approach for Aspect-Oriented Analysis and Design, IEEE, 26th International Conference on Software Engineering (ICSE'04) , pp. 158-167

[2] Bass L., Clements P., Kazman R., Software architecture in practice, Addison-Wesley, 2ª Ed. , ISBN 0-321-14595-9, 2003

[3] Bass L., Klein M. and Northrop L.., Identifying Aspects Using Architectural Reasoning, Proceedings Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Lancaster, 2004, pages 51-57

[4] Buhr R.J.A., Use Case Maps as Architectural Entities for Complex Systems IEEE Transaccions on SoftwareEngineering. (Vol. 24, No. 12) pp. 1131-1155, 1998

[5] Chang Carl K. and Kim Tae-hyung, Distributed Systems Design using Function-Class Decomposition with Aspects, Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004.

[6] Filman et al, Aspect-Oriented Software Development, Addison-Wesley, ISBN 0321-21976-7, 2005.

[7] Ivers, J. et al, Documenting Component and Connector Views with UML 2.0, School of Computer Science, Carnegie Mellon University, TECHNICAL REPORT CMU/SEI-2004

[8] IEEE Architecture Working Group, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[9] Perez J., Ramos I, et al., PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures, IEEE,Proceedings of the Third International Conference On Quality Software (QSIC'03).

[10] Rashid, A., Sawyer, P., Moreira, A., Araújo J., Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, IEEE Joint International Conference on Requirements Engineering , pp. 199, 2002

[11] Tarr, P. And Sutton, S.M. Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns, IEEE, Proceedings of the 21st international conference on Software engineering table of contents,Pages: 107 - 119 , 1999

Integrating Expressiveness of Modern Requirements Modeling Approaches ♦

Elena Navarro
Department of Computer Science, UCLM
Avda. España S/N
Albacete, Spain
enavarro@info-ab.uclm.es

Patricio Letelier
Dept. of Information Systems and Computation, UPV
Camino de Vera s/n
Valencia, Spain
letelier@dsic.upv.es

Isidro Ramos
Dept. of Information Systems and Computation, UPV
Camino de Vera s/n
Valencia, Spain
iramos@dsic.upv.es

Abstract

Currently, Uses Cases are the most widely accepted notation despite their restricted expressiveness and overloaded semantic. Other more sophisticated modeling techniques have been developed independently of any other ones and/or for specific application domains. These techniques often provide broad expressiveness but do not include any scalability or any integration with other approaches. In this work, we present a proposal for requirements modeling that allows the integration of the expressiveness of some of the most relevant approaches in the Requirements Engineering arena. By focusing on the scalability with respect to expressiveness and adaptability to the application domain, we have established some basic guidelines and extension mechanisms that give coherence and semantic precision to our approach.

1. Introduction

The drawbacks in Requirements Engineering (RE) and their negative impact on the success of the software development projects have led to a great deal of research in this field. However, current trends in industrial development have not been related to supposed advances in research. A point has been reached where is absolutely necessary to integrate knowledge and techniques that have been already developed. This integration must consider scalability mechanisms for use at different levels of sophistication and must also provide adaptability in order to smooth their application in specific domains.

This work focuses on five approaches that we consider to be representative in the current state of the art in Requirements Engineering: *traditional* (based on the IEEE 830-1998 [18]), *Use Cases* [6], *Goal Oriented* [26], *Aspect Oriented* [38] and *variability management* [15]. The aim of this work is to present a proposal for integrating these approaches. We have not explicitly included all the details found in each approach. We start from a reduced set of concepts, so that it will be simple and easy to reach a consensus. Additionally, we provide some guidelines to extend this set of concepts, in such a way that a proper semantic coherence can be maintained.

This work is organized as follow. In section 2, we introduce the most relevant approaches that we have integrated in our approach. In section 3, the core concept and the developed Metamodel are presented. Some guidelines to extend the core along an example of its use are described in section 4 and 5, respectively. Eventually, related works and the conclusions round up the work.

2. Approaches to RE

As stated above, we have selected five approaches to Requirement Engineering focusing mainly on the integration of their expressiveness. They have been selected because they are some of the most relevant in the state of the art of this field and have had great influence on practice and/or research. Each of them is briefly described in this section.

2.1 Standard IEEE 830-1998 Approach

The recommendations of this standard are a reference point with regard to the content expected in a Software Requirement Specification. The template

♦ This work has been funded by the Spanish CICYT project DYNAMICA (TIC2003 07776 C02 01)

Proceedings of the 2005 Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05)
0-7695-2297-1/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

Navarro, E., Letelier, P., Ramos, I. Integrating Expressiveness of Modern Requirements Modeling Approaches. Proc. of the 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005). 2005.

proposed in this standard identifies several requirements artifacts, where the principal ones are included in a section called "Specific Requirements".

Unfortunately, this standard does not propose details with regard to the attributes that each of these artifacts should have. Furthermore, apart from a briefly indication about a possible decomposition in sub-functions for the "*function*" artifact, no specific refinement mechanisms are mentioned. Nevertheless, the standard does include some recommendations to deal with potential organizations of specific software requirements that are based on *operation modes of the system*, *user classes*, *objects*, *etc*. However, in practice, when the number of requirements and/or the level of complexity (due to the high number of relationships between them) is significant those recommendations are not enough. Finally, IEEE 830-1998 do not provide any assistance to the process of elaboration or analysis of the requirements..

2.2. Use Case Approach

Uses Cases are perhaps one of the most popular approaches to requirements specification. They have been widely embraced by the industrial community due to their straightforward notation and application. These properties allow stakeholders to easily understand them, and this contributes to the elicitation and validation of the requirements. In a Use Case diagram we mainly distinguish the following elements:

- **Use Cases** represent an atomic functionality (there is no hierarchical refinement when Uses Cases are specified or identified) that the system offers to the environment. Basically, the detailed specification of a Use Case shows the dialogue between the environment and the system to obtain a desired service.
- **Actors** represent the environment of the system and can be users, devices or any other system that interacts with the system under development.
- **Relationships** which include **Communication** (to represent the interaction between the Actor and the Use Case); **Generalization** (applicable both to Uses Cases and Actors to establish specialization hierarchies); **Include** (permits a Use Case to be reused) and **Extend** (simplifies a Use Case).

Additionally, as stated above, Uses Cases do not allow for hierarchical refinement which implies a lack of consensus related to both the proper granularity level of functionality that a Use Case should have and the precise exploitation of *Extend* and *Include* relationships. Consequently, in situations where modeling has to be rigorous and/or precise, Uses Cases usually exhibit problems with regard to their interpretation because of their overloaded semantics and lack of consensus.

2.3. Goal Oriented Approach

Studying the requirements means to determine conflicts, dependencies alternatives to satisfy the construction of software that meets the client's needs. This aim has inspirited most Goal-Oriented proposals such as KAOS [7] or the NFR Framework [4]. Following this approach, there is a wide number of proposals ranging from elicitation to validation activities in the RE process (see [24] for an exhaustive survey). Some concepts are common to all of them:

- **Goal** describes why a system is being developed, or has been developed, from the point of view of the business, organization or the system itself. In order to identify it, both functional goals (expected services of the system) and non functional goals (quality of service, constraints on the design, etc.) should be determined.
- **Agent** is any active component, either from the system itself or from the environment, whose cooperation is needed to operationalize a goal.
- **Refinement Relationships**: AND/OR/XOR relationships allow the construction of the goal model as an acyclic directed graph. These relationships are applied by means of a refinement process (from generic goals towards subgoals) until they have enough granularity to be assigned to specific agents.

This approach offers two advantages that make it especially appropriate for analyzing the specification. The first advantage is related to its ability to specify and manage positive and negative interactions among goals [4], which allow the analyst to reason about different design alternatives and to validate the Goal Model by means of its animation [43]. The second advantage is related to its capabilities concerning traceability from low-level details to high-level goals (*concerns*) [7], which make it especially suitable to bridge the gap between architectural and requirements models.

2.4. Aspect Oriented Approach

The Aspect-Oriented approach has emerged from the implementation level [25], [11] as a mechanism to

manage the complexity of code. With this aim, a set of techniques has been provided that allows the analyst to specify each concern of the system separately. Afterwards, these concerns are weaved according to the expected behavior of the system. The concerns of the system can refer to either its functionality or any other issue such as security or distribution. In this way, the code that would arise tangled in many locations can be managed by modularizing it as an aspect.

There are several recent proposals that promote the detection and description of aspects at early stages of development (such as the design [42] and requirements phases [38]) in order to satisfy the closure property [11]. This is how the term Aspect Oriented Requirements Engineering (AORE) emerged. AORE does not have its own notation or expressiveness Current proposals have developed notations that are based on other techniques in order to accomplish the identification of early aspects. Nonetheless, we can enumerate a set of common concepts that is common to all of them:

- **Concern** refers to the interests of the system, which can be either functional or non-functional.
- **Crosscuting** is a relationship among concerns that arises whenever a given concern interacts with other concerns (either by constraining, extending, etc.). A more detailed description of potential crosscutting relationships can be found in [37].

Some requirements models that are based on the Aspect-Oriented approach include the concept of *early aspect* as a constructor of the model. However, this is not mandatory because, when applying AORE, it is more important to detect and manage the composition relationships between early aspects (along with their possible traceability to other aspects) than to explicitly specify them as early aspects. The main purpose of AORE is to improve the crosscutting management and to establish the composition relationships between specifications. This is done by encouraging the separation of *concerns* in a similar way to the role played by *weaving* in Aspect-Oriented Programming.

2.5. Variability Management Approach

This approach helps the analyst to delay the decision of what functionality or quality aspects will be incorporated in the final system as long as possible. This approach has been successfully applied in two areas: Software Product Lines (SPL) [5] and Dynamic Software Architectures (DSA) [35]. SPL allow the analyst to identify and specify shared product line assets so that the products implemented can be varied. DSA allows the system to evolve in both composition and configuration, by supporting ad-hoc features at run-time. Traditionally, most of these proposals have dealt with the management of variability at the architectural level by establishing a central architecture along with a set of components that can be evolved or integrated according to the system needs. When variability identification is performed at the architectural level a problem related to product lines and dynamic architectures arises because the number of potential systems and the capability of adaptation, respectively, are more limited [15]. Thus, the early identification of the variability which is performed in the requirements phase (*early variability*) is a great advantage.

As in AORE, there is no a widely accepted notation to specify the variability in the requirement phase. However, other approaches such as Use Cases have been extended to support the variability management. Following, we present the necessary concepts as designated by Maßen and Lichter [30] as follows:

- **Representation of common and variable parts**. The notation should allow the analyst to express those assets that are shared among different products of a product line, or among different instances of software architecture and those that are specific to a specific product or instance. The notation should be able to represent both variation points and variants. A variation point indicates a specific variability in the specification. A variant is the specific realization of variability in a specific variation point. Each variation point has a time link, i.e., when the variability is removed during the development process: design, analysis, running, etc. It is also important to define the multiplicity (both maximum and minimum) in each variation. The multiplicity determines how many variants must exist at the same time in a product or architecture when the variability is removed.
- **Distinction between types of variability**. The notation must allow the analyst to express the different types of variability in the following way: a) **option -** when some specific variants can be selected in the instantiation process, b) **alternatives -** when only a single variant can be selected and c) **optional alternative -** when either cero or one alternative can be selected from those available.
- **Representation of dependencies between variable parts**. The variants frequently have

dependencies among them that have to be represented. For instance, some the dependencies that are used are *require* (when a variant must be selected if another variant is present) or *exclude* (when the selection of a variant implies another variant can not be selected). A more extended set of relationships in the requirements stage is presented in [3].

3. Core Concepts in Requirements Engineering Modeling

The first obstacle to overcame in integrating these approaches and their notations is the wide diversity of terms and concepts. We have realized immediately that a consensus metamodel would be impossible to describe if we attempted to cope with the whole expressiveness of all the approaches. Therefore, we decided to organize our work in two parts. The first part, described in this section, defines a metamodel for the essential concepts that deal with generic expressiveness. By doing so, we could get a consensus more easily. The second part (see section 4) describes an extension of the metamodel that has been developed and validated in a specific application domain.

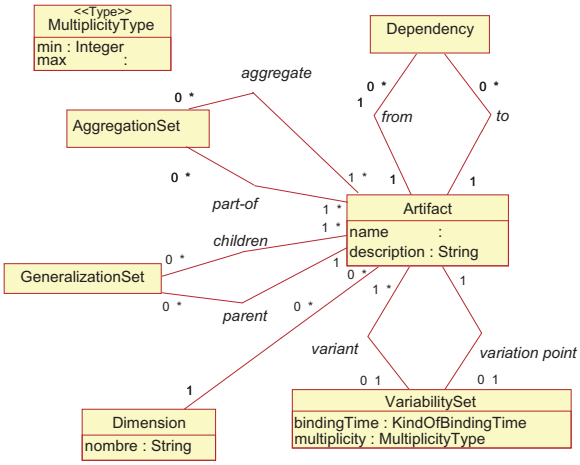


Figure 1. Metamodel for the Core Concepts

The core concepts and their relationships, which are taken from these approaches, are shown in Figure 1. The first topic that we have taken into account to define the metamodel is the multidimensional nature of a SRS. This has led us to organize some of the elements in the specification in several dimensions such as: requirements, features, stakeholders, actors, data, etc. In section 4.1, we provide some guidelines to decide when it is appropriate to establish a new dimension. We have introduced the metaclass *Dimension* in the metamodel to associate each artifact with one dimension.

In addition to the artifact description, it is also necessary to establish the artifact relationship with other artifacts of the SRS. Therefore, we have identified several kinds of relationships which are described in below.

An artifact can be refined through other artifacts, forming a hierarchy structure. Three basic kinds of refinements are included: *VariabilitySet*, *GeneralizationSet* or *AggregationSet*. The *VariabilitySet* refinement can be used to both specify an artifact as a variation point and to establish the relation with its variants. This means that a starting point for a variability hierarchy can be determined from an artifact. During the specification of *VariabilitySet*, the attribute *multiplicity,* which constrains the number of variants to be simultaneously included in the product, must be specified. The attribute *bindingTime* must also be specified in order to determine the maximum delay allowed to decide which variants will be included in the product or in the instanced architecture. Typically, *bindingTime* is assigned as: design time, compile time and runtime.

In the case *GeneralizationSet* or *AggregationSet* refinements, the same artifact can belong to several hierarchies either as parent/aggregated or as child/component, respectively. If *GeneralizationSet* is used, it allows the analyst to define hierarchies of alternative specializations from the same parent and to relate one child to more than one parent by multiple inheritance. If *AggregationSet* is employed it means that an artifact can be part of several aggregate artifacts. Although, these refinements criteria are mutually exclusive, some special cases may arise. For instance, when an established *VariabilitySet* relationship is not only based on a refinement but also on a specialization criterion, this refinement would be considered as a *GeneralizationSet* and a *VariabilitySet* simultaneously.

In addition to these three refinement relationships between artifacts, the dependency relationship has also been included in the metamodel. Perhaps this is the most conflictive relationship for consensus. For this reason, we have preferred to represent it in its most generic form, i.e., by means of *Dependency* which is applicable to artifacts in the core concepts. Several kinds of dependency relationships are allowed between artifact of different hierarchies and/or dimensions; for example, non-functional requirements constraining functional requirements, conflicting goals, data used for requirements; interacting actors with use cases; excluding variants between them; traceability between requirements; etc. Some kinds of dependency between

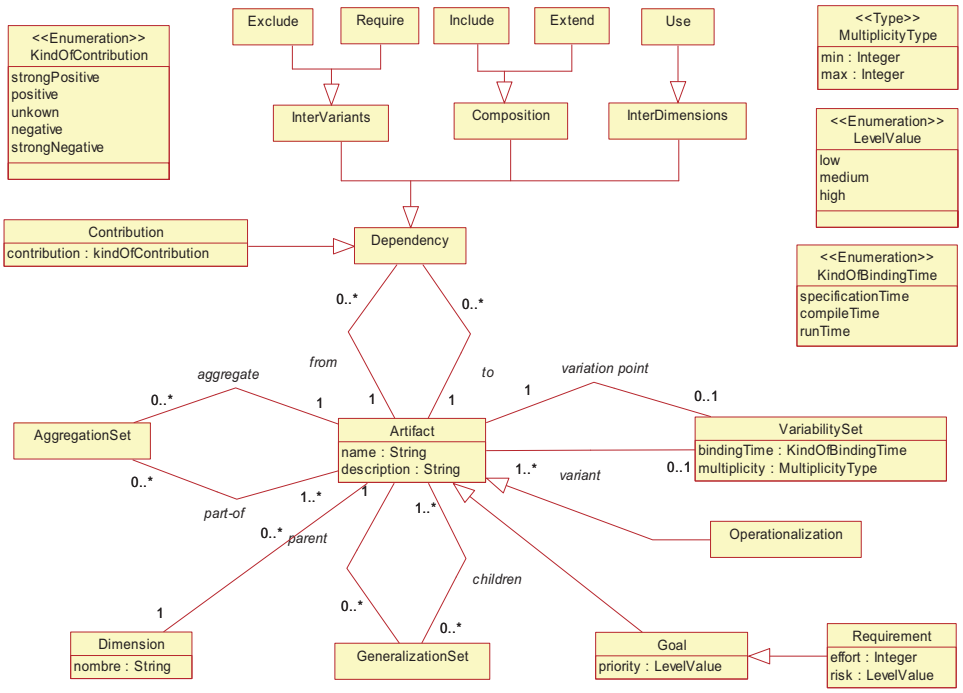


Figure 2. Metamodel of the ATRIUM Goal Model

artifacts are bidirectional, such as those used to specify conflicting goals or mutual exclusion between variants. However, others are unidirectional, such as «*Extension*» dependencies between Use Cases or «*Requires*» dependencies between variants. Therefore, although, in general terms, we consider that dependencies are unidirectional, the metamodel also permits the specification of dependencies that in some cases may imply their inverse.

We have analyzed some current tools for the management of requirements such as DOORS [8], IRqA [20] and RequisitePro [39]. Even though they have a great potential for defining artifacts, they only offer one type of dependency (the traceability specified between artifacts and characterized as *from* or *to*) and only one refinement (using the *AggregationSet* semantic).

4. Customizing the Core

In this section, we will illustrate how the established concepts can be extended and customized according to the particular needs of expressiveness. We present a requirements model that we have developed and tested in a specific application domain. A real system carried out within the European project *Environmental Friendly and cost-effective Technology for Coating Removal* (EFTCoR) [9]. The aim of this project is related to the maintenance operations for ships hull. The system includes operations such as coating removal, cleaning and re-painting of the hull, these operations are performed by a family of robots.

Our requirements model is defined in the context of ATRIUM [34], a methodology that guides the analyst through an iterative process, from a set of user/system needs to the instantiation of a Software Architecture, expressed by means of the ADL PRISMA [36]. This requirement model is a Goal Model [32] which is based on the Dardenee and Lamsweerde's [7] and Chung's et al. [4] proposals. This Goal Model has been extended [31] by integrating the aspect-oriented approach, in order to achieve both the efficient management of *crosscutting* and the correct organization of the SRS. We have also established a strategy for the separation of concerns based on the standard ISO/IEC 9126 and on the IEEE 830-1998 related to the organization of the SRS. These two standards facilitate the incorporation of multidimensional requirements specification. We have also incorporated the variability management in order to apply the model to the definition of product lines and dynamic architectures. This was done to support the specification of a family of robots which was required by EFTCoR project.

4.1 Tasks to adapt or to extend the Metamodel

Based on our previous experience defining the ATRIUM Goal Model, we suggest the following tasks to adapt and/or extend the essential Metamodel described in section 3:

i) *Define the types of artifacts that are relevant to the model*. This permits the inclusion of the needed artifacts from the metaclass *artifact* by means of specialization.
ii) *Establish the refinement relationships of interest*. The essential metamodel provides three forms of refinement: *AggregationSet*, *GeneralizationSet* and *VariabilitySet*. If necessary, an additional refinement type could be added as a specialization of any of the existing ones or as a new and independent one.
iii) *Establish the types of dependency relationships between artifacts*. This allows the definition of the relevant relationships between artifacts from the metaclass *Dependency* by means of specialization.
iv) *Include the attributes considered relevant* to the types of artifacts, types of refinement and types of dependency.
v) Determine the needed dimensions. This task is detailed below.

4.2 Establishing the dimensions of the SRS

The definition of dimensions does not imply a change in the metamodel but rather an instantiation of the metaclass *Dimension*. In our Metamodel, *FNF* (Functional-NonFunctional) refers to an instance (by default) of *Dimension* that is used to specify goals and requirements, that are either functional or non functional. To organize this dimension, we have used the structure of ISO/IEC 9126 (this standard describes the characteristics and subcharacteristics for the quality of software products) as an initial framework for *concerns*. Every *characteristics*/*subcharacteristic* of the ISO/IEC 9126 acts as a *concern* that could be either refined into goals and requirements or simply discarded if it is not relevant. In this way, the goals and requirements can be classified, organized and, possibly, refined, as they are gathered. The taxonomy defined by ISO/IEC 9126 goes beyond the traditional classification of requirements into functional and non functional requirements. Therefore, a wide range of the functional goals and requirements can be organized according to the subcharacteristic *suitability*.

In general, the addition of a new dimension can improve the organization of the SRS by supplying it with an additional perspective. However, an overexploitation of the definition of new dimensions can affect the legibility of the SRS. A new dimension would be appropriate if the *crosscutting* between artifacts in the same dimension was significant and these artifacts did not belong to the same refinement process. For instance, the section "logic requirements of the database" (included into the IEEE 830-1998) is a right candidate for an additional dimension which could be called *Data Entities*. The justification for this is the fact that data requirements are a type of artifact that shows a fair amount of *crosscutting* with artifacts in the dimension FNF. In this case, it is appropriate to establish a new dimension and to specify dependencies *InterDimension* in order to specify relationships between artifacts appearing in the dimension FNF.

However, if there is not much crosscutting between artifacts in a dimension, or those artifacts belong to the same refinement process, it would be better do not separate those types of artifacts into different dimensions. For example, it could be appropriate to keep both goals and requirements in the same dimension when their refinement follows a Goal-Oriented approach.

In order to improve the organization and presentation of the SRS, without using the definition of a new dimension, it would be enough to introduce an attribute on some types of artifacts. For example, if it was necessary to know the person responsible for a specification, instead of defining a new dimension for him, we would only establish the attribute *responsiblePerson* on artifacts requiring it.

5. Goal Model of ATRIUM

Figure 2 shows the metamodel of the ATRIUM Goal Model. The artifacts and the identified relationships shown in the figure are the results of tasks described in section 4.1. There are three kinds of artifacts that are usually presented in a Goal-Oriented approach: *goal*, *requirement* and *operationalization*. A requirement is a specialization of a goal, but a requirement must also be verifiable and assignable to an agent by means of an operationalization. *Goals*, *requirements* and *operationalizations* have been defined by inheriting from *Artifact* (task (i) in section 4.1).

We have defined a complete and disjoint hierarchy of types of dependencies by performing task (iii). On this hierarchy, several specializations were identified: *Intervariant*, *Composition* and *InterDimension*. *Intervariant* expresses a dependency between variants. This dependency can be either *Require* or *Exclude*,

IEEE COMPUTER SOCIETY

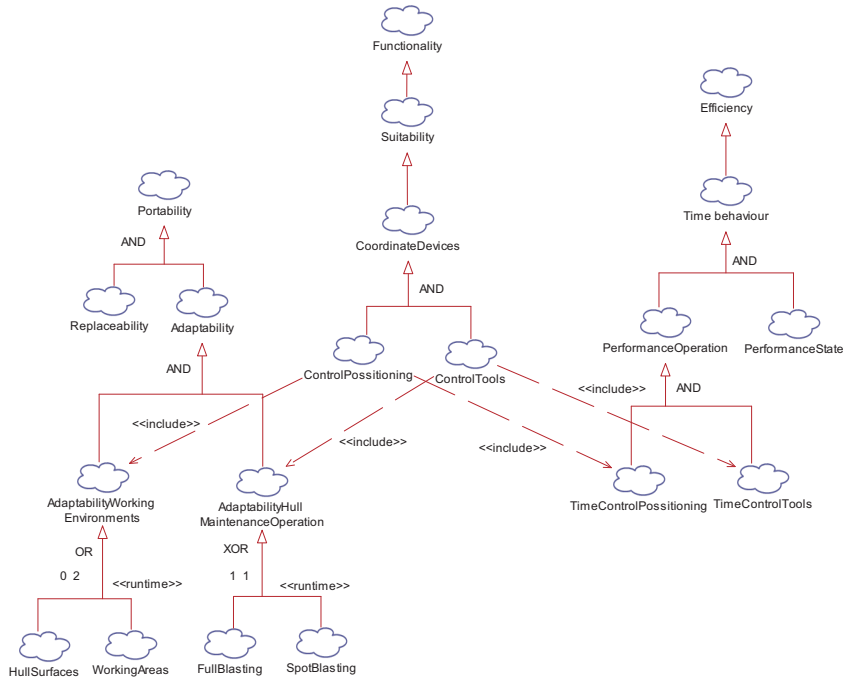


Figure 3. A part of the Goal Model of the EFTCoR system

using the semantic described in section 2.5. *Composition* allows the establishment of a composition dependency between artifacts. To do this, we have defined the same dependencies as those applied to Use Cases, i.e., *Include* or *Extend*; however, these relationships will model composition dependencies for crosscutting requirements. Finally, *InterDimension* allows the definition of dependencies between artifacts that were specified in different dimensions. *Uses* have been defined by inheriting from *InterDimension* in order to specify a use dependency between artifacts specified in different dimensions.

Additionally, we have established a contribution dependency from one artifact to another because this kind of dependency can be specified along with any other kind of dependency, according to the hierarchy defined in Figure 2. A special case arises when two artifacts affect each other in a negative or positive way, which generates a conflicting situation that is modeled by setting *strongNegative* as a value for the attribute *contribution*.

Due to space limitations, we have not indicated the integrity constraints of the model when task (iii) is performed; these constraints are applied in addition to those determined by associations and multiplicities defined in the metamodel of Figure 2. In the ATRIUM Goal Model, these constraints refer to the types of artifact that can be related by means of a specific kind of dependency. For example, *Exclude* or *Require* dependencies are applied only to variants. A more typical restriction is that the three kinds of refinement hierarchies (described above) do not admit cycles.

In task (iv), the relevant attributes should be defined. In our case study, we have established the following attributes: *priority* for the metaclass *Goal*; *risk* and *effort* for the metaclass *Requirement*; and *Contribution* for the metaclass *Contribution*. Their possible values have been declared at the corresponding enumeration showed in Figure 2.

In task (v), in addition to the dimension FNF included by default, we have added the dimension *Data Entities* to specify artifacts that represents data entities. Connecting artifacts from the dimension FNF to the dimension *DataEntities* was achieved by means of the dependency *Use*.

In order to achieve a graphical representation for the Goal Model, we defined a UML profile, from the established metamodel, which was introduced in [33]. A Goal Model could then be elaborated in a straightforward way by using any CASE tool that supports UML. Figure 3 illustrates a goal model performed with the editor of Use Case diagrams provided by Rational Rose [40].

Figure 3 only shows an outline of the FNF dimension of the model: a goal refinement. We started from the characteristic/subcharacteristic *Portability/ Adaptabilty, Functionality/Suitability* and *Efficiency/ Time Behaviour* of the *ISO/IEC 9126* standard. The

AggregationSet refinements labeled with AND indicate that the satisfaction of an aggregated goal is achieved if every one of its component goals is satisfied. The refinement OR under the goal *AdaptabilityWorking Environment* indicates that it is a variation point, which is defined by means of a refinement *VariabiltySet.* Thus, in our example, either the goal *Adaptability to Hull Surfaces* or the goal *Adaptability to Working Areas* had to be satisfied in order to achieve a system that was self-adaptive to different working environments. In this case, both subgoals were satisfied by means of characteristics added at runtime, as the label specified. The *0..2* multiplicity indicates that the goal *AdaptabilityWorkingEnvironments* is optional, due to the fact that it can have *0* as the minimum satisfied goals. *AdaptabilityHul-Maintenace-Operation* is also a variation point that should have at least one variant at runtime (minimum multiplicity 1). It was labeled as XOR to denote that only a subgoal can be satisfied as maximum (maximum multiplicity 1). The functionality of coordinating devices (*CoordinateDevices*) was refined into two subgoals: positioning control (*ControlPositioning*) and tools control (*ControlTools*) of the robot. The specification of these subgoals can be composed into other non-functional goals acquired from the refinement of *Adaptability* and *Efficiency.* The refinement of the goal *Efficiency* establishes time constraints to the control and tools positioning according to those established by *TimeControlPositioning* and *TimeControlTools.* This composition of specifications was modeled by means of dependencies that were stereotyped as *Include*.

6. Related works

In this section, we focus on describing works that include some of the requirements engineering approaches described in section 2. It is important to point out that their purpose is not the integration of approaches but the extension of an existing notation to achieve some specific expressiveness. This is the case of the Use Case approach which has been extended to model variability [13], [16], [30].

Within the scope of the Aspect-Oriented approach, the proposals included in [31] and [44] exploit a Goal Model to identify and specify aspects in early stages. Navarro et al. [31] have extended the expressiveness of the Goal Model by incorporating a new type of relationship which allows specifying the crosscutting between goals. Yu et al. [44] identify the crosscutting whenever an operationalization contributes to the satisfaction of several goals.

Brito et al [2] present an extension of the Use Case diagram for its application to the AORE context. They have specified «*wrappedBy*» and «*overlapBy*» relationships by stereotyping a dependency. In this way, if a non functional requirement implies a constraint on a functional one, this would appear as a Use Case that is stereotyped as a *crosscutting concerns* and would present a weaving relationship with the other Use Cases. Other proposals, such as [1] and [41], consider it more appropriate to employ two different models to specify functional and non functional requirements, using a Use Case model and a Goal Model, respectively. The main problem with these approaches is that specification can hardly be analyzed, due to the fact that they use two separate models.

In [13], [16] and [30] extensions to Use Case diagrams are used to manage the variability. All of them take advantage of the easy communication with the user; however there are drawbacks whenever a specific expressiveness is required to specify the variability. Basically, these proposals have increased the expressiveness of the Use Case diagrams by means of the introduction of stereotypes that allow the modeling of variability concepts. Thus, for example, when a Use Cases specifies a functionality that some products of the family have, it is stereotyped as «*variant*». The proposal presented by Halmans and Pohl [16] is perhaps the more extensive because it is the only one to incorporate its own graphical notation to specify the variation points and the maximum and minimum multiplicities. However in this case, the formulated extension present some drawbacks that are inherent to the Use Case approach, because it does not provide by default any support to analyze or validate the model.

Similarly, the Goal-Oriented Approach has been recently applied to modeling and analyzing variability. Meaningful works in this area are those presented in [14] and [17]. When the Goal-Oriented approach is applied, AND/OR/XOR relationships appear as a consequence of the construction process of the model and are recorded in the refinement graph. Thus, these goals that have been refined by means of OR and XOR relationships are capable of representing variation points in the specification of a product line or a dynamic architecture, in a natural way. However, these proposals have some deficiencies from the point of view of the required expressiveness for variability; for example, they lack the concept of multiplicity or specific dependencies between variants. However, they do provide sophisticated mechanism for analysing alternatives that have been used to select the optimum

variant for each product [14] or for each profile/ability of the user [17].

7. Conclusions

We have analyzed the expressiveness of five approaches for specifying requirements. Although these approaches are usually alternatives in this process, the expressiveness that they provide is complementary. Therefore, a metamodel that integrates the expressivenesses of several approaches, could be useful in many cases. However, one of the main problems when defining highly expressive models (which can handle a wide range of types of artifacts and/or dependencies) is achieving consensus with regard to their semantic. Additionally, the state of the art of the application of these approaches indicates a trend towards the simplest ones, specifically, those based on IEEE 830-1998 or on Use Cases; other more advanced approaches are applied only to specific domains. Hence, a metamodel must provides proper integration as well as scalability from simple towards other more sophisticated techniques. We consider that our proposal can contribute to solving the problems stated above. Therefore, we would like to point out the following features of our work:

- Definition of a metamodel which includes a core of concepts that corresponds to the essential expressiveness of some of the more popular and/or advanced approaches in requirements engineering.
- Establishment of guidelines for adapting the Metamodel to specific needs, according to the expressiveness. We are currently validating both the core and the presented extension of the metamodel by means of a UML profile and its application to the EFTCoR system.
- This metamodel is the first step towards a tool that supports modeling requirements by integrating expressiveness from different approaches. The demands for scalability and expressiveness, adaptability to specific application domains and the flexible organization of the SRS only can be efficiently satisfied by means of a software tool. The proposed metamodel is being used to define a repository of the tool.

In addition, we want to point out two topics to deal with. The first one is related to study other interesting approaches in Requirements Engineering area in the context of our proposal. We want to study: *View Points* [12], *Problem Frames* [21] or *Tasks Model* [27]. Based on the results of our case study, we think it will be easy to include the additional required expressiveness.

The second one is related to a formal framework. It is necessary to specify artifacts and to provide a precise semantic for the expressiveness provided by the model. The works by Letier and Lamsweerde's [28] or Katz and Rashid's [23] can be useful as a reference to achieve this aim.

References

[1] Brito, I., Moreira, A. "Integrating the NFR framework in a RE model", *Early Aspects 2004: Aspect Oriented Requirements Engineering and Architecture Design Workshop*, March 22, 2004 Lancaster, UK in conjunction with 3rd AOSD Conference.

[2] Brito, I., Moreira, A. "Towards a composition process for aspect oriented requirements", Early Aspects 2003: Aspect Oriented Requirements Engineering and Architecture Design Workshop, March 17, 2003 Boston, USA in conjunction with 2nd AOSD.

[3] Bühne, S., Halmans, G. and Pohl, K. "Modeling Dependencies between Variation Points in Use Case Diagrams", Ninth International Workshop on Requirements Engineering: Foundation for Software Quality. Collocated to CAiSE'03, 16 17 June 2003, Klagenfurt/Velden, Austria

[4] Chung, L., Nixon, B. A., Yu, E. and Mylopoulos, J., *Non Functional Requirements in Software Engineering*, Kluwer Academic Publishing, 2000.

[5] Clements, P. and Northrop, L. *Software Product Lines Practices and Patterns*, Addison Wesley, 2001.

[6] Cockburn, A. *Writing Effective Use Cases,* Addison Wesley Professional, 2000.

[7] Dardenne, A., van Lamsweerde, A. and Fickas, S. "Goal directed Requirements Acquisition", *Science of Computer Programming*, 20, (1993), pp. 3 50.

[8] DOORS, www.telelogic.com/products/doorsers/doors/

[9]EFTCOR: Environmental Friendly and cost effective Technology for Coating Removal. 5th Framework Program European Project (GROWTH G3RD CT 00794), 2003.

[10] Elrad, T., Filman, R. E., Bader, A. "Aspect oriented programming: Introduction", *Communications of the ACM*, 44(10), October 2001, pp. 29 32.

[11] Elrad, T., Aksits, M., Kiczales, G., Lieberherr, K. and Ossher, H. "Discussing aspects of AOP", *Communications of the ACM,* 44(10), October, 2001, pp. 29 32.

[12] Finkelstein A. and Sommerville I., "The Viewpoints FAQ." BCS/*IEE Software Engineering Journal,* 11(1), 1996.

[13] Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern based Software Architectures, ed. Addison Wesley Professional, 2004.

[14] Gonzales Baixauli, B., Sampaio, J. C., Mylopoulos, J. "Visual Variability Analysis with Goals Models", Proc. of the International Conference on Requirements Engineering, IEEE Computer Society Press, 2004, pp. 198 207.

[15] van Gurp, J., Bosch, J. and Svahnberg, M. "On the notion of variability in software product lines", Proc.

WICSA'01, Amsterdam, The Netherlands, August 28 31, 2001, IEEE Computer Society Press, pp 45 54.
[16] Halmans, G., Pohl, K. "Communicating the Variability of a Software Product Family to Customers". *Software and Systems Modeling*, Vol. (2): 15 36, Springer, Hamburg, 2003.
[17] Hui, B., Liaskos, S., Mylopoulos, J. "Requirements Analysis for Customizable Software Goals Skills Preferences Framework", Proc. 11th IEEE Int. RE Conference Monterey Bay, California, USA, pp. 117, September 08 12, 2003.
[18] IEEE Std 8301998. IEEE Recommended Practice for Software Requirements Specifications, In Volume 4: Resource and Technique Standards, The Institute of Electrical and Electronics Engineers, Inc. IEEE Software Engineering Standards Collection.
[19] ISO/IEC Standard 9126 1 Software Engineering Product Quality Part1: Quality Model, ISO Copyright Office, Geneva, June 2001
[20] IrqA, www.irqaonline.com
[21] Jackson, M. *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison Wesley Pub, 2000
[22] John, I. and Muthig, D. "Tailoring use cases for product line modelling", *Int. Workshop on Requirements Engineering for Product Lines* (REPL), Essen, Germany, 2002.
[23] Katz, S. and Rashid, A. "From Aspectual Requirements to Proof Obligations for Aspect Oriented, Systems", Proc. 12th IEEE Int. RE , September 06 10, 2004, Kyoto, Japan,.
[24] Kavakli, E. Loucopoulos, P. Goal Modeling in Requirements Engineering: Analysis and Critique of Current Methods. Information Modeling Methods and Methodologies, John Krogstie, Terry Halpin and Keng Siau (eds), IDEA Group, pp 102 124, 2005.
[25] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. Loingtier, J. M. and J. Irwin, "Aspect Oriented Programming". Proceedings of ECOOP, Finland. Springer Verlag LNCS 1241. June 1997.
[26] van Lamsweerde, A. "Goal Oriented Requirements Engineering: A Guided Tour", Invited Paper for 5th IEEE Int. Symp. on RE, Toronto, August, 2001, pp. 249 263.
[27] Lauesen S., "Task Descriptions as Functional Requirements", *IEEE Software*, 20(2), 2003, pp. 58 65.
[28] Letier, E.and van Lamsweerde, A., "Deriving Operational Software Specifications from System Goals", Proc. SIGSOFT FSE 2004, Charleston, November 2002.
[29] Letelier, P., Navarro, E., Anaya, V. "Customizing Traceability in a Software Development Process", Information Systems Development. Advances in Theory, Practice, and Education, Vasilecas, O.; Caplinskas, A.; Wojtkowski, G.; Wojtkowski, W.; Zupancic, J.; Wryczа, S. (Eds.), Kluwer/Plenum Press (to appear).
[30] von der Maßen, T. and Lichter, H. "Modeling Variability by UML Use Case Diagrams", *International Workshop on Requirements Engineering for Product Lines*, collocated to IEEE Int. Conf. on RE, Germany, September 9, 2002.
[31] Navarro, E., Letelier, P. and Ramos, I., "Goals and Quality Characteristics: Separating Concerns", *Early Aspects 2004: Aspect Oriented Requirements Engineering and Architecture Design Workshop*, collocated to OOPSLA 2004, Vancouver, Canada, October 25, 2004.
[32] Navarro, E., Ramos, I. and Pérez, J. "Goals Model Driving Software Architecture", *Proc. of the 2nd Int. Conf. on Soft. Eng. Research, Management and Applications*, Los Angeles, USA, May 5 8, 2004, pp. 205 212.
[33] Navarro, E., Letelier, P. and Ramos, I., "UML Visualization for an Aspect and Goal Oriented Approach", *5th Aspect Oriented Modeling Workshop*, collocated to UML 2004 Conference, Lisbon, Portugal, October 11, 2004.
[34] Navarro, E., Ramos, I. and Pérez, J., "Software Requirements for Architectured Systems", *Proc. of 11th IEEE Int. RE Conference*, Monterey, California, USA, September 8 12, 2003, IEEE Computer Society 2003, pp. 365 366.
[35] Oreizy, P., Medvidovic, N., Taylor, R. N. "Architecture Based Runtime Software Evolution", Proc. 20th Int. Conf. on Software Engineering, Kyoto, April 1998.
[36] Perez, J., Ramos, I., Jaén, J., Letelier, P., Navarro. E. "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", *Proceedings 3rd IEEE Int. Conf. on Quality Software*, Dallas, USA, Nov. 6 7, 2003, pp. 59 66.
[37] Rashid, A., Moreira, A., Araújo, J. "Modularisation and composition of aspectual requirements", *Proc. 2nd Int. Con. on AOSD*, Boston, , March 17 21, 2003
[38] Rashid, A., Sawer, P., Moreira, A. and Araújo, J. "Early Aspects: a Model for Aspect Oriented Requirements Engineering", *Proc. Int. Conf. on Requirements Engineering*, September 9 13, 2002, Essen, Germany, pp. 199 202.
[39] RequisitePro, http://www.306.ibm.com/software/awdtools/reqpro/
[40] Rational Rose, http://www 306.ibm.com/software/awdtools/developer/datamodeler/
[41] Sousa, G., Soares, S., Borba, P., J. Castro, "Separation of Crosscutting Concerns from Requirements to Design: Adapting the Use Case Driven Approach", *Early Aspects 2004: Aspect Oriented Requirements Engineering and Architecture Design Workshop*, March 22, 2004 Lancaster, UK, collocated to 3rd AOSD Conference.
[42] Suzuki, J. and Yamamoto, Y. Extending UML with Aspects: Aspect Support in the Design Phase. *Aspect Oriented Programming Workshop*, Lisbon, Portugal, June, 1999, Collocated to ECOOP'99, Springer LNCS 1743.
[43] Tran Van, H., van Lamsweerde, A., Massonet, P. and Ponsard, C. "Goal Oriented Requirements Animation", *Proc. Int. Conf. on Requirements Engineering*, September 06 10, 2004 Kyoto, Japan, IEEE Computer Society 2004.
[44] Yu, Y., Sampaio do Prado Leite, J.C. and Mylopoulos, J. "From Goals to Aspects: Discovering Aspects from Requirements Goal Models", *Proc. Int. Conf. on RE*, Sep. 06 10, 2004, Kyoto, Japan, IEEE Computer Society 2004.
[45] Yu, E., Liu, L., & Li, Y. "Modelling Strategic Actor Relationships to Support Intellectual Property Management", *Proc. Int. Conf. on* Conceptual Modelling, November 27 30, 2001, Yokohama, Japan.

IEEE COMPUTER SOCIETY

PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos

Cristóbal Costa, Jennifer Pérez, Nour Ali, José A. Carsí, Isidro Ramos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
46071 Valencia, España
{ccosta | jeperez | nourali | pcarsi | iramos} @dsic.upv.es

Resumen

La evolución se ha convertido en un requisito fundamental en el desarrollo de software de sistemas complejos. PRISMA proporciona un modelo que permite el desarrollo de este tipo de sistemas combinando el Desarrollo de Software Basado en Componentes y el Desarrollo de Software Orientado a Aspectos. En este trabajo, se presenta cómo PRISMA da soporte a la evolución de arquitecturas software basadas en componentes y orientadas a aspectos siguiendo la propuesta MOF. PRISMA define un metanivel que permite tratar los modelos como datos y hacerles evolucionar. Además, proporciona servicios de evolución para posibilitar el cambio de la aplicación en tiempo de ejecución sin detenerla. El soporte a la evolución del modelo se ha materializado en un middleware para la plataforma .NET, el cual se presenta en detalle en este trabajo. Dicho middleware, llamado PRISMANET, ha sido validado mediante un caso de estudio real, demostrando cómo PRISMA ofrece la evolución de elementos arquitectónicos y aspectos de forma estática y dinámica.

1. Introducción

Los requisitos de los sistemas software están sometidos a continuas variaciones, bien por cambios en su entorno o por cambios en el propio sistema. Junto a los sistemas, sus aplicaciones han de evolucionar para que cumplan los nuevos requisitos.

En la actualidad los sistemas software cada vez son más complejos y lo que antes eran mejoras adicionales como la distribución, reutilización y evolución, ahora se han convertido en requisitos primordiales para ellos. Esta complejidad y el avance constante de las nuevas tecnologías hace que los cambios en las aplicaciones sean más frecuentes.

Dichos constantes cambios producen un incremento de tiempo y coste en el proceso de mantenimiento de las aplicaciones; así como costes adicionales para aquellos sistemas en los que la parada de la aplicación, para la incorporación de los cambios, supone pérdidas económicas importantes.

La forma de paliar estos inconvenientes es proporcionando mecanismos de evolución, tanto en las plataformas de desarrollo, como en los modelos a los que dan soporte. Una aportación importante en este sentido es la propuesta MOF de la OMG [6] que propone una arquitectura de cuatro niveles para la gestión de modelos (M3: meta-meta modelo, M2: meta-modelo, M1: modelo, M0: datos). MOF define una arquitectura que permite el almacenamiento de descripciones de modelos a distintos niveles de abstracción y la manipulación de estas descripciones como metadatos, permitiendo su evolución de forma estática.

Sin embargo, no sólo es importante la evolución de la descripción almacenada de los modelos, también es necesaria la evolución en tiempo de ejecución de una aplicación sin tener que ser detenida. En este trabajo se presenta cómo el modelo PRISMA [9] permite dar un soporte completo a la evolución de los sistemas software complejos mediante la definición e implementación de un metamodelo. De este modo PRISMA sigue la aproximación MOF (ver Tabla 1).

PRISMA proporciona un modelo que permite el desarrollo de arquitecturas software complejas

Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos, JISBD'2005, pp.27-34
ISBN: 84-9732-434-X **© 2005 Los autores, Thomson**

Costa, C., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005.

mediante la combinación del Desarrollo de Software Orientado a Aspectos (DSOA) y el Desarrollo de Software Basado en Componentes (DSBC).

MOF	PRISMA
M2	Metamodelo PRISMA
M1	Modelos PRISMA
M0	Aplicación PRISMA

Tabla 1. Equivalencia entre MOF y PRISMA

En este trabajo se presenta cómo el metamodelo de PRISMA se ha implementado para dar soporte a la evolución de los modelos almacenados siguiendo la propuesta MOF, así como de las aplicaciones en ejecución sin necesidad de ser detenidas. La implementación del metamodelo y sus capacidades de evolución se han materializado en un middleware, llamado PRISMANET. PRISMANET extiende la funcionalidad de la tecnología .NET, tanto para dar soporte a la evolución de los tipos como a la reconfiguración dinámica de las arquitecturas. Asimismo, dicho middleware permite el desarrollo de software orientado a aspectos sobre tecnología .NET, evolución de aspectos y componentes, movilidad y comunicación distribuida.

La estructura del trabajo es la siguiente: en primer lugar se presenta brevemente el modelo PRISMA. En el apartado 3, se presentan las características de evolución de PRISMA y su metamodelo. En el apartado 4 se detalla la implementación del middleware PRISMA centrándose en las características de evolución. En el apartado 5 se presenta un caso de estudio real y la incorporación de nuevos requisitos mediante los mecanismos de evolución de PRISMA. En el apartado 6 se presentan los trabajos relacionados. Finalmente, en el apartado 7 se presentan las conclusiones y trabajos futuros.

2. PRISMA

El modelo PRISMA permite definir arquitecturas de sistemas software complejos. Principalmente se caracteriza por la integración que realiza del DSOA y del DSBC y el soporte completo que ofrece para la evolución de sus modelos arquitectónicos. La integración del DSOA y el DSBC se realiza definiendo las características relevantes (distribución, coordinación, seguridad, etc.) de un elemento arquitectónico como aspectos. Al definir las propiedades de interés del sistema separadas en aspectos, se evita que el código de los elementos arquitectónicos esté mezclado y se mejora su mantenimiento. Por otro lado, el soporte a la evolución que ofrece el modelo es posible gracias a la definición del metanivel y a sus propiedades reflexivas.

2.1. Modelo

Un componente es un elemento arquitectónico que captura la funcionalidad del sistema y no actúa como coordinador de otros elementos arquitectónicos. Sin embargo, un conector es un elemento arquitectónico que actúa como coordinador de elementos arquitectónicos. Es importante destacar que los conectores no tienen las referencias de los elementos que conectan y viceversa. Estas referencias son almacenadas en los canales de comunicación que se establecen entre ellos (*attachments*). De este modo, todo elemento arquitectónico definido en el modelo es completamente reutilizable.

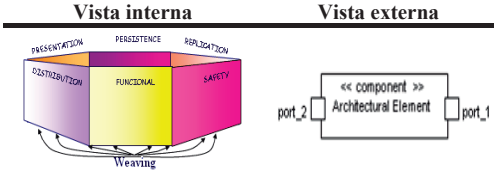


Figura 1. Diferentes vistas de un componente

Los elementos arquitectónicos pueden ser vistos desde dos perspectivas distintas, interna (*white box view)* y externa *(black box view).* La vista interna muestra un elemento arquitectónico como un prisma, de forma que cada lado del prisma se corresponde con un aspecto. La vista externa encapsula la funcionalidad del elemento teniendo sólo la visibilidad de los servicios que éste publica a través de sus puertos (ver Figura 1).

Un aspecto PRISMA es una propiedad de interés del sistema (*concern)* que es común y compartido por el conjunto de tipos del sistema (*crosscutting concerns*). Los tipos de aspectos (seguridad, distribución, coordinación, etc.) que forman un elemento arquitectónico varían dependiendo del sistema software.

Un elemento arquitectónico PRISMA está formado por un conjunto de aspectos de distinto tipo, las relaciones de sincronización entre estos aspectos (*aspects weaving*) y uno o más puertos o roles. Los puertos/roles representan los puntos de

interacción entre los elementos arquitectónicos. Los puertos se utilizan para definir componentes y los roles para definir conectores.

El weaving establece las sincronizaciones necesarias entre los distintos aspectos que conforman un elemento arquitectónico. El weaving define que la ejecución de un servicio de un aspecto puede generar la invocación de un servicio de otro aspecto.

2.2. Lenguaje de Descripción de Arquitecturas

PRISMA proporciona un Lenguaje de Descripción de Arquitecturas (LDA) para la especificación de sus modelos. El LDA de PRISMA define los elementos arquitectónicos a dos niveles de abstracción: nivel de definición de tipos y nivel de configuración. En el nivel de definición de tipos se definen los tipos de elementos arquitectónicos (interfaces, aspectos, componentes y conectores) de un sistema software determinado. Dichos tipos se almacenan en un repositorio para que puedan ser reutilizados por cualquier modelo arquitectónico. En el nivel de configuración se definen arquitecturas específicas para un determinado sistema software mediante la instanciación de los tipos que se definen en el nivel anterior y la interconexión de dichas instancias. De este modo se establece la topología de la arquitectura de un sistema software específico.

3. Evolución en PRISMA

PRISMA permite la evolución de arquitecturas software orientadas a aspectos y basadas en componentes. Por este motivo, permite tanto la reconfiguración dinámica de las arquitecturas software como la evolución de los aspectos que la constituyen.

3.1. Características

Debido a que la definición de las arquitecturas PRISMA se define a dos niveles de abstracción (nivel de definición de tipos y nivel de configuración), el soporte a la evolución también se proporciona a estos dos niveles. La evolución de tipos permite modificar la estructura y el comportamiento de los tipos PRISMA: las interfaces, los aspectos, los componentes y los conectores. La evolución de tipos es llevada a cabo mediante la invocación de servicios de evolución.

Por otra parte, la evolución a nivel de configuración permite la invocación de servicios de evolución que afectan a la vista externa de los elementos arquitectónicos. Estos servicios pueden modificar el número de elementos arquitectónicos, la comunicación entre ellos o la estructura de la arquitectura.

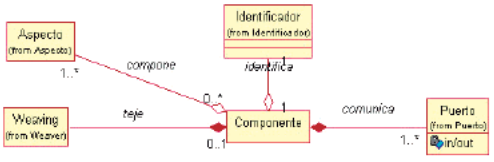
La ejecución de servicios de evolución puede desencadenar la invocación de otros servicios de evolución, con el objetivo de mantener la consistencia del modelo. Por ejemplo, la modificación de los puertos de un componente también afectará a los attachments que estén relacionados con dichos puertos y también requerirán su modificación. Por su parte, la creación o destrucción de la instancia de un componente también requerirá la invocación de los servicios de evolución de los puertos y de los attachments.

Finalmente, es importante señalar que la evolución de las arquitecturas PRISMA es proporcionada tanto en tiempo de especificación del modelo como en tiempo de ejecución. Esto permite que tanto la definición de los tipos como la reconfiguración de las arquitecturas pueda realizarse de forma dinámica, y la evolución sea desencadenada a consecuencia de las reglas de negocio o de requisitos no funcionales. De esta forma, pueden añadirse nuevos aspectos a una arquitectura software ya definida para ajustarse a nuevas especificaciones o definirse reglas para ajustarse a requisitos no funcionales. Por ejemplo, podrían establecerse reglas para equilibrar la carga en la comunicación distribuida: si un componente detecta que las comunicaciones que establece con otro componente remoto tienen un rendimiento excesivamente bajo, puede invocar servicios de evolución para solicitar la movilidad a otra máquina en la que el rendimiento en las comunicaciones sea mayor. En este caso estaríamos hablando de reconfiguración dinámica.

3.2. Metamodelo PRISMA

Todas las características de evolución del modelo forman parte del metanivel de PRISMA, nivel M2 de la propuesta MOF. Este metanivel se ha representado mediante un metamodelo. Dicho metamodelo contiene una metaclase por cada tipo PRISMA (aspectos, puertos, componentes, conectores, etc.). Cada una de las metaclases definen el conjunto de propiedades y servicios de evolución necesarios para el tipo que representa.

La Figura 2 muestra como ejemplo el paquete *componente* del metamodelo y algunos de los servicios de evolución de este tipo.



Metaclass *Component*
new NewComponent(Name: string);
destroy DestroyComponent(Name:string);
AddAspect (Name: string, type: AspectType);
RemoveAspect(name: string);
AddPort(Name: string, Interface: string);
RemovePort(name: string);
AddWeaving(InAspect: string,InService:string, OutAspect:string, OutService: string);
RemoveWeaving(name: string);
End_Metaclass *Component*;

Figura 2. Paquete *Componente* del metamodelo

4. Implementación del metamodelo

Uno de los objetivos de PRISMA es dar soporte a la evolución de sus modelos arquitectónicos. Para ello, su metamodelo y servicios de evolución han sido implementados como un middleware que permite ejecutar arquitecturas dinámicas y orientadas a aspectos sobre tecnología .NET.

4.1. El middleware PRISMA

El desarrollo del middleware PRISMA se ha realizado sobre la plataforma .NET y ofrece aquellas características requeridas por PRISMA que .NET no proporciona de forma directa. En la Figura 3 se muestra la estructura del middleware PRISMA, dividido en cuatro áreas funcionales:

- Modelo de ejecución de los tipos PRISMA: contiene las construcciones .NET que definen el comportamiento de los *aspectos*, *componentes* y *attachments*. Estas construcciones proporcionan el comportamiento genérico de cada uno de los elementos del modelo. Las instancias de un modelo PRISMA específico extenderán dicho comportamiento.
- Gestión de los elementos arquitectónicos: se encarga de la creación, destrucción y mantenimiento de las instancias de los elementos arquitectónicos PRISMA
- Servicios de comunicación distribuida y movilidad.
- Servicios de evolución: permiten la evolución dinámica de los tipos y la reconfiguración de la arquitectura. Desencadena los cambios necesarios para mantener la consistencia del modelo.

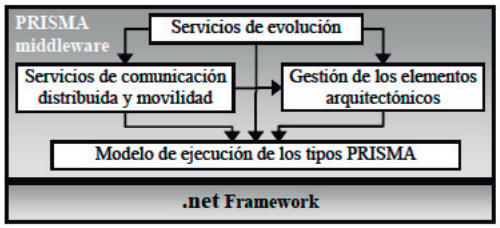


Figura 3. Estructura del middleware PRISMA

Las flechas que aparecen en la Figura 3 entre las distintas áreas funcionales del middleware indican la interacción que requieren las áreas entre sí. Como se puede observar, el área de servicios de evolución requiere interactuar con el resto de áreas funcionales. Requiere interactuar con los servicios de movilidad para la reconfiguración dinámica, para el caso en que un elemento arquitectónico deba ser movido a otra ubicación. Igualmente requiere interactuar con los servicios de gestión de elementos arquitectónicos para posibilitar la creación o destrucción de las instancias existentes. Y además, debe interactuar con el modelo de ejecución genérico para poder incorporar los cambios a las instancias PRISMA que se encuentran en ejecución.

Aquellas arquitecturas que son de carácter distribuido, requieren la ejecución de una instancia del middleware en cada una de las máquinas susceptibles de contener instancias en ejecución de la arquitectura PRISMA. Cada middleware será el encargado de gestionar las instancias en ejecución de cada nodo específico, proporcionándoles los servicios de mantenimiento y de evolución. Por otro lado, con el objetivo de mantener la consistencia de la arquitectura software distribuida y permitir a las instancias funcionar como si trabajaran a nivel local, cada middleware se comunica con los middlewares de los otros nodos implicados.

4.2. Servicios de evolución del middleware

La evolución de las arquitecturas PRISMA se realiza en dos etapas complementarias:

- Evolución dinámica: permite la evolución de las instancias en ejecución, las cuales se

encuentran almacenadas en memoria. Se mantiene el estado y los servicios en ejecución no afectados por los cambios.

- Persistencia de los cambios: mediante técnicas para la generación dinámica de código se actualizan los tipos definidos por la aplicación PRISMA específica para que reflejen los cambios realizados.

Todos los cambios provocados son propagados por el middleware en el que se han desencadenado hacia los demás middlewares. De esta forma se mantiene la consistencia del modelo en cada uno de los nodos en los que se está ejecutando la aplicación.

La separación de la evolución en dos etapas es importante para permitir la evolución en tiempo de ejecución. Aplicar los cambios en memoria permite que las instancias en ejecución puedan reflejar los cambios realizados sin necesidad de detener la ejecución de la aplicación. Sin embargo, también es necesario actualizar dichos cambios en disco, para que futuras instancias de los tipos de la aplicación PRISMA modificada sean consistentes. La persistencia de dichos cambios es realizada por un conjunto de clases especializadas que generan dinámicamente el código, ya que conocen los patrones de código para cada uno de los elementos arquitectónicos.

En cambio, la evolución dinámica es proporcionada por las clases genéricas que definen el comportamiento del modelo PRISMA. Esto es posible gracias a que la estructura interna de los elementos arquitectónicos es altamente independiente: pueden sustituirse las partes que lo forman sin afectar a las otras áreas funcionales de cada elemento arquitectónico. Por ejemplo, puede sustituirse un aspecto de un componente sin que por ello afecte a los otros aspectos en ejecución del componente.

4.3. Implementación de los tipos PRISMA y su soporte a la evolución dinámica

La implementación de los tipos PRISMA se ha realizado teniendo en cuenta que la evolución en tiempo de ejecución de las aplicaciones PRISMA ha de ser posible. Por tanto, también ha de considerarse la generación dinámica de código[1] para añadir/eliminar aspectos, componentes, conectores o attachments. El conjunto de clases que implementan los tipos PRISMA se han agrupado en espacios de nombres de .NET con el objetivo de realizar una distinción clara entre las clases implicadas en la implementación de cada uno de los tipos. Puede observarse que cada uno de los tipos está formado por varias clases.

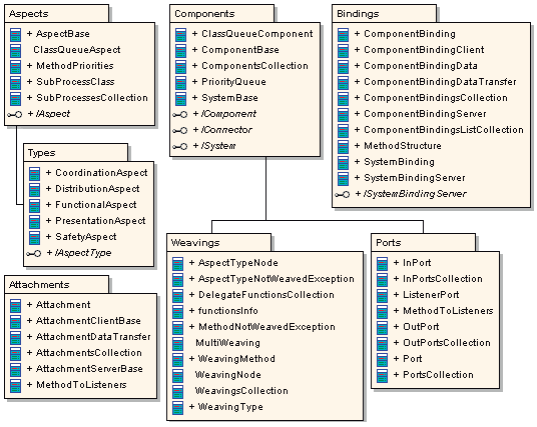


Figura 4. Conjunto de clases de los tipos PRISMA[2]

El comportamiento genérico de los aspectos. se encuentra definido en la clase *AspectBase*, cuyo comportamiento es extendido por cada uno de los diferentes tipos de aspectos existentes: *DistributionAspect*, *FunctionalAspect*, etc. A su vez, cada uno de estos tipos de aspectos es extendido para implementar el aspecto concreto definido por el analista PRISMA.

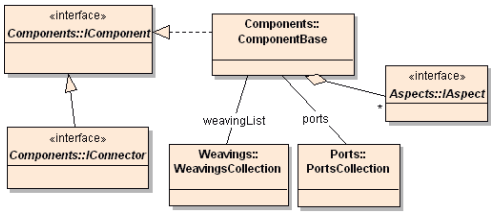


Figura 5. Implementación de los componentes[2]

En la Figura 5 se muestra la estructura de las clases genéricas que implementan el comportamiento de los componentes y conectores. La clase *ComponentBase* define el comportamiento común a todo componente y conector. Como puede observarse, un componente

[1] La inyección de código en tiempo de ejecución se realiza mediante un conjunto de funciones proporcionadas por NET: *System.Reflection.Emit*

[2] El conjunto de clases que aparece en la figura ha sido generado automáticamente a partir del código fuente del middleware, utilizando la herramienta Sparx: http://www.sparxsystems.com/

se ha implementado como una agregación dinámica de aspectos, lo que permite que puedan ser añadidos o eliminados en tiempo de ejecución. Dichos aspectos se ejecutan de forma concurrente, con lo que un componente puede servir simultáneamente peticiones en varios aspectos.

De forma similar, los puertos/roles y los *weavings* también han sido implementados mediante una agregación dinámica, lo que permite que puedan añadirse en tiempo de ejecución al componente. Sin embargo, su implementación es más compleja ya que son los encargados de las comunicaciones entre los componentes, para el caso de los puertos, y las sincronizaciones entre aspectos, para el caso de los weavings.

Los puertos permiten dar visibilidad a los servicios implementados por los aspectos, de modo que, cuando un aspecto es actualizado, debe desencadenarse también el servicio de evolución correspondiente para el puerto. De igual modo, la eliminación de un aspecto también requiere comprobar si había algún weaving relacionado y, en ese caso, eliminarlo.

La creación de un componente es realizada mediante la invocación de los servicios de evolución de la metaclase *Component* para realizar la agregación dinámica de aspectos (*AddAspect)*, weavings (*AddWeaving*) y puertos (*Inports.Add* y *OutPorts.Add*). De igual modo, estos servicios pueden ser invocados en tiempo de ejecución para modificar la estructura de un componente.

La reconfiguración dinámica puede realizarse añadiendo/eliminando instancias de componentes y conectores, y/o mediante la modificación de las relaciones entre ellos. La creación y destrucción de nuevas instancias de elementos arquitectónicos en tiempo de ejecución es implementada por los respectivos servicios de evolución, como por ejemplo *NewComponent* y *DestroyComponent*.

Los servicios de evolución *AddAttachment* y *RemoveAttachment* llevan a cabo la creación o modificación de las relaciones entre los elementos arquitectónicos. Dichas relaciones son establecidas mediante la creación/destrucción dinámica de los attachments y su conexión a los puertos implicados en la comunicación. Esta conexión se realiza mediante un mecanismo de suscripción a los puertos, tras lo cual el attachment puede enviar y/o recoger peticiones de servicio y reenviarlas a otro elemento arquitectónico, que puede ser remoto. De esta forma, la creación de una conexión entre componentes se traduce a la creación de un attachment específico para los puertos que conecta, y la suscripción a dichos puertos para poder enviar/recibir peticiones de servicio.

5. Evolución en el TeachMover

Con el objetivo de validar el modelo y sus capacidades de evolución, se ha realizado la especificación completa de la arquitectura PRISMA del robot TeachMover [13], el cual posee una gran cantidad de componentes, conectores y attachments a diferentes niveles de abstracción [8]. Su funcionalidad consiste en mover objetos, siendo controlado por un operario desde un ordenador. Está compuesto por una serie de articulaciones que permiten su movimiento: base, hombro, codo, muñeca y herramienta. La especificación PRISMA del robot ha sido validada mediante su ejecución a través del middleware PRISMA. En la Figura 6 puede observarse cómo el componente Operator, a través de un conector, se encuentra conectado con el componente complejo *RUC*, el cual define el comportamiento del robot. Toda la arquitectura del robot se encuentra ubicada en un mismo nodo, lo que implica que el control del mismo ha de ser local.

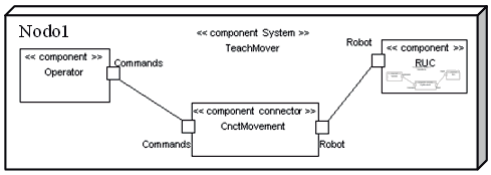


Figura 6. Arquitectura del robot TeachMover

Al poco tiempo de haber especificado y ejecutado el prototipo, emergieron nuevos requisitos software no contemplados. Entre otros, se detectó la necesidad de que el componente encargado de la interfaz con el operario pudiese ser distribuido, es decir, que el operario pudiese controlar el robot desde diferentes máquinas. Mediante los servicios de evolución del middleware, este requisito se implementó en dos etapas.

En primer lugar, para dotar al componente *Operator* de soporte a la comunicación distribuida y la movilidad, el analista le añadió el aspecto de distribución de forma dinámica mediante la invocación del servicio de evolución *AddAspect()*, proporcionado por el middleware. La ejecución de

este servicio se traduce en la agregación dinámica del nuevo aspecto a todas las instancias en ejecución del componente *Operator*, y en la generación automática de código[1] para reflejar dicho cambio en la clase que representa dicho componente. De esta forma, el código generado del componente es el siguiente:

```
public class Operator: ComponentBase {
    public Operator(string name,
      MiddlewareSystem middlewareSystem):
      base (name,middlewareSystem) {
      IAspect functionalAspect = new FAskMovement();
      AddAspect(functionalAspect);
      IAspect presentationAspect = new PRobot();
      AddAspect(presentationAspect);
      IAspect distributionAspect = new DOperator();
      AddAspect(distributionAspect);
      AddWeaving(functionalAspect, "MoveOK",
          WeavingType BEFOREIF("success",1),
          presentationAspect,"ShowMessage",functions);
      InPorts.Add("Commands", "IOperator",
        functionalAspect, "ASK");
      OutPorts.Add("Commands", "IOperator", "ASK");
  } }
```

A partir de este momento, el componente ofrece un nuevo servicio, *Move*, proporcionado por el aspecto de distribución. El operario podrá invocar este servicio para solicitar la movilidad del componente *Operator* (y con ello el control del robot) a otra máquina.

La ejecución del servicio *Move* desencadena la invocación de otros servicios de evolución para lograr la movilidad del componente manteniendo su estado. Para ello, en primer lugar el middleware obtiene el conjunto de attachments que están conectados a los puertos del componente que va a ser movido y detiene tanto los attachments como el componente para evitar que otros componentes envíen nuevas peticiones de servicio. Después, el middleware serializa el estado del componente y la información de los attachments que tenía conectados y lo envía a la máquina destino. El middleware de la máquina destino, una vez recibe el estado, crea un nuevo componente del mismo tipo invocando el servicio *NewComponent(Operator)* y transfiere el estado recibido al nuevo componente. Del mismo modo, también crea tantos attachments como tenía el componente mediante sucesivas invocaciones al servicio *AddAttachment*(). Si todo el proceso ha finalizado con éxito, el middleware de la máquina origen es notificado por el middleware remoto y procede a la destrucción de las copias originales, ejecutando respectivamente los servicios *RemoveComponent* y *RemoveAttachment*. Finalmente, el middleware remoto reanuda la ejecución del nuevo componente *Operator* y de los attachments. De esta forma, mediante la combinación de los servicios de evolución con los de distribución y movilidad, la arquitectura es modificada dinámicamente, manteniendo el estado de los elementos arquitectónicos implicados y las comunicaciones con el resto.

6. Trabajos relacionados

Existen una gran variedad de trabajos relacionados con la evolución de arquitecturas y la orientación a aspectos. En cuanto a evolución, uno de los LDAS con mayor semejanza al LDA de PRISMA es el propuesto por Loques y Leite [4] en su modelo R-RIO. El LDA de R-RIO tiene capacidades de evolución y reconfiguración al igual que PRISMA, sin embargo no incorpora la noción de aspecto. Existen otros enfoques que utilizan técnicas con un nivel de abstracción inferior, orientadas a implementación [5]. En GUARANÁ [7] se presenta un meta-nivel complejo que hay que definir durante la compilación, siendo su estructura ajena al modelo.

Por otro lado, existen muchas aproximaciones de implementación que soportan el DSOA, como AspectJ [2] para entornos Java, y AspectC# [3] y SourceWeave.NET [1] para entornos .NET. Sin embargo, ninguna de estas aproximaciones aporta mecanismos para añadir o eliminar aspectos dinámicamente. En cambio, Rapier-Loom.NET [11] sí que lo permite, pero a costa de definir los weavings en los aspectos, penalizando de este modo su reutilización. SetPoint! [12] también permite añadir y eliminar aspectos dinámicamente, pero requiere etiquetar adecuadamente el código de la aplicación para realizar el weaving en tiempo de ejecución.

Ninguna de las aproximaciones anteriores tiene en cuenta de forma explícita las relaciones existentes que puedan surgir a consecuencia de agregar varios aspectos a un mismo punto del código. En cambio, JAsCo.Net [14] proporciona un lenguaje más expresivo que permite definir las relaciones entre los aspectos. En un intento por unir el DSOA con el DSBC, introduce el concepto de conectores para realizar el weaving entre los aspectos y el código y mantener con ello una alta reutilización. Sin embargo, el weaving dinámico requiere de una plataforma de ejecución para interceptar la aplicación en ejecución e insertarle

los aspectos.

En cambio, PRISMA define un modelo que combina la reutilización proporcionada por el DSOA y la reconfiguración dinámica de los modelos DSBC. Los aspectos se definen de forma separada a los weavings, y son altamente reutilizables. La implementación en .NET de dicho modelo ha permitido añadir y eliminar aspectos dinámicamente, así como modificar los weavings definidos, sin que para ello se necesite detener el componente en ejecución.

7. Conclusiones y trabajos futuros

En este trabajo se han presentado las características más importantes del modelo PRISMA para explicar el soporte que ofrece a la evolución. PRISMA se presenta como un modelo innovador que permite evolucionar tanto componentes como aspectos. Éste sigue la propuesta MOF definiendo un metanivel para evolucionar sus modelos. Además, proporciona servicios de evolución para modificar la aplicación sin necesidad de ser detenida. Las características de evolución de PRISMA son soportadas por PRISMANET, un middleware que permite la ejecución y evolución de arquitecturas software distribuidas y orientadas a aspectos sobre tecnología .NET. Los mecanismos de evolución que ofrece PRISMANET han sido validados con un caso de estudio de ámbito industrial como el robot *TeachMover*. Actualmente, nuestro trabajo está centrado en la implementación de un compilador de modelos para generar código automáticamente partiendo de especificaciones PRISMA. Además, el soporte gráfico al lenguaje y su integración con PRISMANET se está realizando para proveer un marco de trabajo PRISMA que proporcione un soporte completo al desarrollo de arquitecturas software orientadas a aspectos. Como trabajo futuro se deberá considerar el problema de la evolución de datos de arquitecturas software persistentes. Para este trabajo serán considerados nuestros trabajos previos en migración y evolución de datos de esquemas conceptuales orientados a objetos [10].

Referencias

[1] Jackson A., Clarke S., *SourceWeave.NET Cross-Language Aspect-Oriented Programming*. In Proc. of Generative Programming and Component Engineering (GPCE). Vancouver, Canada, 2004.

[2] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., *An Overview of AspectJ*. In ECOOP, (Budapest, Hungary, 2001), Springer-Verlag, pp.327-355.

[3] Kim, H. *AspectC# An AOSD implementation for C#*. MSc. Thesis, Comp.Sci, Trinity College, Dublin, Ireland, 2002.

[4] Loques O., Sztajnberg A., Leite J., Lobosco, M., *On the Integration of Meta-Level Programming and Configuration Programming*. In Reflection and Software Engineering, V.1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[5] McGurren F., Conroy D., *X-ADAPT An Arquitecture for Dynamic Systems*. Workshop on Component-Oriented Programming, ECOOP, Málaga, Spain, 2002.

[6] MOF, Meta-Object Facility, http://www.omg.org/technology/documents/formal/mof.htm

[7] Oliva A., Garcia I.C., Buzato L.E., *The Reflective Architecture of Guaraná*. Technical Report IC-98-14. Instituto de computación, Universidad de Campiñas, abril, 1998.

[8] Pérez J., Ali N., Carsí J.A., Ramos I, *Arquitectura PRISMA para el Caso de Estudio Robot 4U4*. Informe Técnico DSIC II/13/04, Universidad Politécnica de Valencia, 72 pages, 2004.

[9] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E.: *PRISMA Towards Quality, Aspect Oriented and Dynamic Software Architectures.* In proceedings of 3rd IEEE International Conference on Quality Software (QSIC), Dallas, Texas, USA, November 2003

[10] Pérez, J., Carsí, J A., Ramos, I.: *On the implication of application's requirements changes in the persistence layer an automatic approach.* Workshop on the Database Maintenance and Reengineering (DBMR'2002), IEEE International Conference of Software Maintenance, Montreal, ISBN: 84-699-8920-0, Canada, October 1st (2002) 3-16.

[11] Schult, W. and Polze, A., Speed vs. Memory Usage – An Approach to Deal with Contrary Aspects. In 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) in AOSD 2003, (Boston, Massachusetts, 2003).

[12] SetPoint! Project, http://www.dc.uba.ar/people/proyinv/setpoint/

[13] *The TeachMover* robot: http://www.questechzone.com/microbot/teachmover.htm

[14] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V., *JAsCo.NET Capturing Crosscutting Concerns in .NET Web Services.* In Proc. of Second Nordic Conference on Web Services NCWS, In "Mathematical modelling in Physics, Engineering and Cognitive Sciences", Vol. 8, November 2003.

Implementación Orientada a Aspectos en C# de un Sistema Robótico Tele-operado[1]

Rafael Cabedo, Jennifer Pérez, Nour Ali, Isidro Ramos, Jose Ángel Carsí

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
46071 Valencia, España
{rcabedo | jeperez | nourali | iramos | pcarsi }@dsic.upv.es

Resumen. El desarrollo de software basado en componentes y orientado a aspectos esta despertando un gran interés, no sólo en el ámbito académico, sino también en el industrial. El modelo prisma se caracteriza por la combinación que realiza del Desarrollo de Software Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA) y por su metanivel que permite la evolución de sus arquitecturas. El modelo PRISMA ha sido aplicado a un caso de estudio de ámbito industrial, el robot TeachMover. Se eligió este caso de estudio con el objetivo de abordar un sistema real que permitiera comprobar que realmente el modelo reúne los requisitos necesarios para la construcción de arquitecturas software complejas, distribuidas, dinámicas y reutilizables. Concretamente, este trabajo presenta la implementación en lenguaje C# de los elementos arquitectónicos, aspectos y *weavings* (tejidos, sincronizaciones) entre aspectos del robot mediante el uso del *middleware* de PRISMA, llamado PRISMANET. La implementación del robot ha dado como resultado una aplicación orientada a aspectos sobre .NET que permite mover a un sistema robótico tele-operado de una forma segura.

1. Introducción

En la construcción de sistemas software actuales, cada vez cobran más relevancia características que no forman parte del proceso de negocio. Este cambio, junto con el hecho de que es necesario dar un soporte a la evolución de los sistemas software de la forma más dinámica y menos costosa posible, ha hecho plantearse nuevos enfoques de desarrollo software como el propuesto por PRISMA.

PRISMA [3] es un enfoque de desarrollo de software cuyo modelo combina el Desarrollo de Software Basado en Componentes (DSBC) [8] y el Desarrollo de Software Orientado a Aspectos (DSOA) [1]. El modelo da la misma relevancia tanto a

[1]Este trabajo ha sido financiado por el CICYT (Centro de Investigación Científica Y Tecnológica), Proyecto DYNAMICA (DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures), TIC 2003-07804-C05-01. Además, su desarrollo ha sido financiado por Microsoft Research Cambridge, proyecto "PRISMA: Model Compiler of Aspect-oriented component-based software architectures".

Cabedo, R., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Implementación Orientada a Aspectos en C# de un sistema robótico tele-operado. III Taller de Desarrollo de Software Orientado a Aspectos de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2005.

requisitos funcionales como a requisitos no funcionales. Esto se consigue especificándolos de la misma manera mediante el concepto de aspecto. Por otro lado, la orientación a aspectos aporta simplicidad en el proceso de evolución, ya que cuando se desean cambiar propiedades que afectan a un solo interés del producto software, solo es necesario considerar las propiedades del aspecto que especifica el interés (*concern*) que se desea evolucionar.

Además, la combinación del DSBC y el DSOA permite tener diferentes niveles de reutilización y adaptación que permiten mejorar tanto los tiempos de desarrollo como los de mantenimiento.

Dicho enfoque está siendo desarrollado sobre la plataforma .NET [6]. Si bien es cierto, la tecnología .NET no proporciona directamente la programación orientada a aspectos. Por este motivo, ha sido necesario desarrollar un *middleware*, llamado PRISMANET, que proporciona todas estas funcionalidades necesarias para la ejecución de arquitecturas software PRISMA sobre la plataforma .NET.

Se ha especificado [5] e implementado un caso de estudio real haciendo uso del enfoque PRISMA, el robot TeachMover. El objetivo de este trabajo es mostrar la implementación de los aspectos de las componentes del robot y cómo se han implementado los *weavings* (tejidos, sincronizaciones) entre aspectos, para finalmente dar como resultado una aplicación C# que permite el movimiento del robot con una implementación orientada aspectos siguiendo el modelo PRISMA y sobre tecnología .NET.

La estructura del trabajo es la siguiente: en primer lugar se presenta brevemente el modelo PRISMA y su *middleware* PRISMANET. En el apartado 3, se presenta el caso de estudio real del TeachMover. En el apartado 4, se presenta la implementación de aspectos y *weavings* haciendo uso de PRISMANET. Finalmente, en el apartado 5 se presentan las conclusiones y trabajos futuros.

2. PRISMA sobre PRISMANET

El modelo PRISMA [3] permite definir arquitecturas de sistemas software complejos. Principalmente, se caracteriza por la integración que realiza del DSOA y del DSBC. Esta integración se realiza definiendo las características relevantes (distribución, coordinación, seguridad, etc.) (*concerns*) de un elemento arquitectónico como aspectos.

Un aspecto PRISMA es un *concern* que es común y compartido por el conjunto de tipos del sistema (*crosscutting concerns*). Los tipos de aspectos que forman un elemento arquitectónico varían dependiendo del sistema software.

Un elemento arquitectónico PRISMA (componente o conector) está formado simétricamente [2] por un conjunto de aspectos de distinto tipo, las relaciones de sincronización entre estos aspectos (*weavings*) y uno o más puertos o roles. Los puertos se utilizan para definir componentes y los roles para definir conectores. Estos representan los puntos de interacción entre los elementos arquitectónicos. El *weaving* establece las sincronizaciones necesarias entre los distintos aspectos que conforman un elemento arquitectónico. El *weaving* define que la ejecución de un servicio de un aspecto puede generar la invocación de un servicio de otro aspecto.

Uno de los objetivos de PRISMA es el de generar automáticamente el código de sus arquitecturas software orientadas a aspectos sobre la plataforma .NET. Para ofrecer aquellas características PRISMA que .NET no ofrece de forma directa, ha sido desarrollado un *middleware* llamado PRISMANET [4]. Dicho *middleware* permite la ejecución de arquitecturas dinámicas y orientadas a aspectos PRISMA. El *middleware* implementa el modelo PRISMA mediante un conjunto de clases base que se corresponden con los conceptos PRISMA. De este modo, el desarrollo de una aplicación PRISMA específica se realiza mediante la extensión de dichas clases.

3. CASO DE ESTUDIO: Robot TeachMover

Con el objetivo de validar el lenguaje de descripción de arquitecturas de PRISMA, se ha realizado la especificación e implementación de un caso de estudio real. Este caso de estudio se basa en el control de un brazo robot conocido como TeachMover [7]. Este robot está formado por un conjunto de articulaciones (*Base, Shoulder, Elbow, Wrist*) y una pinza que posibilitan la recogida, el desplazamiento y la deposición de objetos.

Cada una de las articulaciones está compuesta por un actuador y un sensor. Los actuadores son los encargados de mandar acciones a las articulaciones o a la herramienta del robot, para que procesen la acción que éstos especifican. Mientras que los sensores realizan la lectura del resultado de dichas acciones, para saber si éstas se han ejecutado correctamente. La función del robot es la de moverse, bien sea por pasos simples o especificando la posición en grados que debe alcanzar una articulación. Además, se debe de realizar un control antes de cada movimiento puesto que las articulaciones sólo pueden desplazarse entre un rango predeterminado. Así, se comprueba que los movimientos son seguros, que pueden ejecutarse y que ni el robot, ni cualquier otro elemento que forme parte del entorno corre peligro.

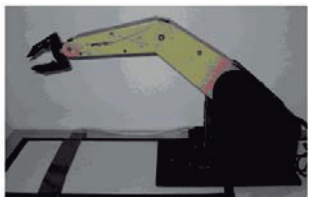


Fig. 1. Robot TeachMover

Concretamente, en este artículo se va a utilizar como ejemplo la implementación que se ha realizado del aspecto de seguridad de cada una de las articulaciones. A continuación se muestra parcialmente su especificación:

```
Safety Aspect SMotion
  Attributes
    Constant
     minimum: integer, NOT NULL;
```

```
    maximum: integer, NOT NULL;
  Services     ... ...
    in check(input Degrees: integer, output Secure: boolean);
    Valuations
    {(Degrees >= minimum) and (Degrees <= maximum)}
    [check(Degrees, Secure)] Secure := true};
    {(Degrees < minimum) or (Degrees > maximum)}
    [check(Degrees, Secure)] Secure := false;
  Protocol ... ...
End_Safety Aspect SMotion;
```

4. IMPLEMENTACIÓN

En PRISMA, un aspecto se implementa como una clase de C# definida dentro de un ensamblado para que sea totalmente reutilizable. Para dar soporte a la implementación de los aspectos se emplean las clases definidas en PRISMANET [4]. Estas proporcionan la funcionalidad genérica y común para todos los aspectos. En PRISMA existen un número indefinido de tipos de aspectos (coordinación, seguridad...), cada tipo tiene ciertas propiedades que lo caracterizan pero comparte una funcionalidad común con el resto de aspectos. En PRISMANET existe una clase para cada tipo de aspecto, estas clases extienden la clase raíz *AspectBase* en la que está definida la funcionalidad común. Para definir un aspecto concreto se debe extender por herencia una clase que representa un tipo de aspecto. En la implementación del caso de estudio, el aspecto de seguridad *SMotion* extiende la clase *SafetyAspect* de PRISMANET. La figura 2 muestra los distintos tipos de aspecto que se han utilizado en el caso de estudio. Concretamente, la figura muestra cómo se define el aspecto de seguridad *SMotion* extendiendo la clase *SafetyAspect* de PRISMANET. El propósito del aspecto *SMotion* es el de asegurar que los movimientos que se solicitan a las articulaciones del robot son seguros.

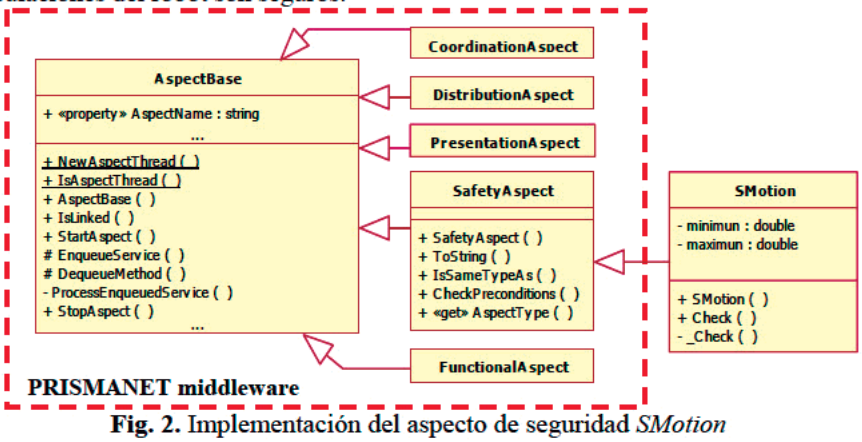


Fig. 2. Implementación del aspecto de seguridad *SMotion*

Los mecanismos heredados de *AspectBase* permiten a los aspectos ejecutarse concurrentemente y de forma asíncrona a través de un hilo de ejecución propio y de una cola, donde se almacenan las invocaciones dirigidas a los servicios del aspecto.

En implementación, los atributos de los aspectos se implementan como variables privadas de clase, por ejemplo los atributos *minimun* y *maximun* son los valores máximo y mínimo en grados que el movimiento de una articulación puede alcanzar.

Los servicios de los aspectos se implementan a través de dos métodos, uno público y otro privado. El método privado implementa el comportamiento definido en el servicio y el método público da soporte a la ejecución de los servicios de forma asíncrona. Cuando se solicita un servicio de un aspecto, se invoca el método público. Éste encola, en forma de delegado, la petición del método privado en la cola de petición de servicios (e*nqueueService*). De esta forma, cuando el hilo de ejecución del aspecto no esté procesando ninguna otra petición, este desencolará la petición y ejecutará el método privado asociado. En el aspecto *Smotion*, el servicio *Check* toma como parámetro los grados en los que se va a situar una articulación y devuelve si el ángulo es válido o no. Este comportamiento se ha implementado del siguiente modo:

```
public AsyncResult Check(double newAngle) {
   return EnqueueService(cinematicsMoveJointDelegate,newAngle);}
private AsyncResult _Check(double newAngle, out bool secure){
   if ((newAngle >= minimun) && (newAngle <= maximun))
      secure = true;
   else secure = false;}
```

Los aspectos son instanciados y agregados dinámicamente dentro de los elementos arquitectónicos. En implementación, un componente o un conector se han traducido por una clase que extiende la clase base *ComponentBase* de PRISMANET. Esta clase, además de integrar en su interior los aspectos y los puertos de comunicación, también realiza dinámicamente los *weavings* entre los aspectos. Se ha de tener en cuenta, que no sólo se da soporte a la agregación de aspectos y weavings a un componente en tiempo de carga, también es posible realizar su composición dinámica de aspectos o weavings emergentes en tiempo de ejecución. Si bien es cierto, dicha agregación dinámica no es persistente porque el middleware todavía no da soporte a la emisión de código en tiempo de ejecución.

En el caso de estudio existe un conector que se encarga de llevar a cabo la coordinación de los componentes Actuador y Sensor a través de su aspecto de coordinación. El componente Actuador es el que envía los comandos de movimiento al robot y el componente Sensor es el que recupera, después del movimiento, si se ha efectuado correctamente. Mediante el servicio *cinematicsMoveJoint* del aspecto de coordinación, el conector indicará al Actuador que efectúe un movimiento. Pero, antes de que se efectúe el movimiento es necesario comprobar que es seguro. Por este motivo, el aspecto de seguridad *SMotion* se añade al conector cuando este es construido.

En PRISMANET existe un conjunto de clases para dar soporte a los *weavings*. Un *weaving* se asocia a un servicio origen de algún aspecto del elemento arquitectónico. Este servicio desencadenará la invocación de servicios destino de otros aspectos del elemento. La asociación entre un servicio origen y un servicio destino se establece a

través de una lista dinámica de delegados. El servicio origen puede tener asociados hasta tres tipos de *weavings* distintos: *after*(o *afterIf*), *before*(o *beforeIf*) e *instead*.

Cuando se invoca el servicio de un aspecto, si tiene *weavings* asociados, la dependencia de los servicios requiere que el *weaving* se ejecute de forma síncrona, por lo que su ejecución se realiza en un hilo de ejecución independiente.

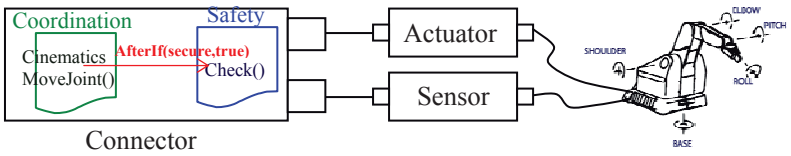


Fig. 3. *Weaving* entre el aspecto de seguridad y el aspecto de coordinación

Mediante el *weaving* se define que el servicio *cinematicsMoveJoint* se ejecutará después del servicio *Check* si el movimiento es seguro. Es decir, la invocación de *cinematicsMoveJoint* del aspecto de coordinación desencadena la ejecución (*weaving*) del servicio *Check* definido en el aspecto de seguridad (ver figura 3). De esto modo, los *weavings* hacen de sincronizadores entre los distintos aspectos del componente que se están ejecutando concurrentemente.

La definición de un *weaving* se realiza en el cuerpo de constructor del elemento arquitectónico. Su definición se realiza mediante el método *AddWeaving,* codificado en la superclase que da soporte a los elementos arquitectónicos de PRISMANET, y requiere como argumentos el nombre del servicio origen y la instancia del aspecto donde está definido el servicio, el nombre del servicio destino y la instancia del aspecto donde está definido, así como el tipo de *weaving*. A continuación se muestra la instrucción que asocia el *weaving* entre el aspecto de seguridad y el aspecto de coordinación del conector.

```
AddWeaving(coordinationAspect,"CinematicsMoveJoint",
WeavingType.AFTERIF("secure",true),safetyAspect,"Check");
```

5. CONCLUSIONES Y TRABAJOS FUTUROS

El trabajo presenta una introducción de como han sido implementados los aspectos y *weavings* del modelo PRISMA mediante C#. NET. Para ello, se ha utilizado como ejemplo el caso de estudio de un sistema robótico tele-operado, el robot TeachMover. El resultado de este trabajo ha permitido ejecutar una arquitectura orientada a aspectos sobre la plataforma .NET que ha posibilitado el movimiento del robot TeachMover.

A corto plazo, este trabajo va a servir para la identificación de los patrones de generación automática de código. Dichos patrones se utilizarán para desarrollar el compilador de modelos PRISMA a lenguaje C#.

Bibliografía

[1] Aspect-Oriented Software Development, http://aosd net
[2] Cuesta, C. E., Romay, P., De La Fuente, P., Barrio-Solórzano, M., Architectural Aspect of Architectural Aspects. Second European Workshop on Software Architecture (EWSA'05), Lecture Notes in Computer Science 3527, Pisa, June 2005.
[3] Pérez J., Ali, N., Carsí J.A., Ramos I., Dynamic Evolution in Aspect-Oriented Architectural Models. Second European Workshop on Software Architecture (EWSA'05), Lecture Notes in Computer Science 3527, Pisa, June 2005.
[4] Pérez J., Ali N. , Costa C., Carsí J.A., Ramos I., Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology, Journal of .NET Technologies, Vol.3, No.1-3, 2005 ISSN 1801-2108.
[5] Pérez J., Ali N., Carsí J.A., Ramos I.,"Arquitectura PRISMA para el Caso de Estudio Robot 4U4", DSIC-II/13/04, Universidad Politécnica de Valencia, 2004.
[6] Robinson S. et al. , Professional C# 2nd Edition, Wrox Programmer to Programmer.
[7] TeachMover Robot, http://www.questechzone.com/microbot/teachmover htm
[8] Szyperski, C., *Component software: beyond object-oriented programming*, (New York, USA: ACM Press and Addison Wesley, 2002).

Sistematizando la Especificación de Requisitos Safety: un Caso de Estudio sobre Aplicaciones Teleoperadas

†Elena Navarro, ⌉Pedro Sánchez, ‡Patricio Letelier, ⌉Juan A. Pastor, ‡Isidro Ramos

†Departamento de Informática, UCLM, Albacete
enavarro@info-ab.uclm.es

‡Departamento de Sistemas de Información y Computación UPV, Valencia
{letelier|iramos}@dsic.upv.es

⌉Departamento Tecnologías de la Información y Comunicaciones, UPCT, Cartagena
{pedro.sanchez | juanangel.pastor | }@upct.es

Resumen

Los sistemas robóticos suelen desempeñar tareas que involucran importantes riesgos. Los posibles daños que pueden ocasionarse, tanto en el entorno de trabajo como en el propio sistema, llevan a clasificar estos sistemas dentro de la categoría de *Safety critical systems*, es decir, sistemas en los cuales la gestión rigurosa de los aspectos Safety es fundamental. En este trabajo presentamos nuestra propuesta para el tratamiento de los requisitos Safety y su trazabilidad hacia especificaciones de la arquitectura del sistema. Se presenta un procedimiento para la identificación y especificación de requisitos Safety basado en un marco de orientación a objetivos. Además, en nuestro trabajo se han considerado e integrado otros enfoques para requisitos Safety y recomendaciones de estándares relevantes en este ámbito. Mediante un caso de estudio de carácter industrial se ilustra cómo utilizando nuestra propuesta se han establecido patrones de operacionalización para requisitos Safety de un sistema teleoperado.

1. Motivación

Los sistemas robóticos difieren significativamente de otras aplicaciones software por factores como la necesidad de interacción con el entorno, la presencia de perturbaciones, etc. Además, si los errores derivados del uso del sistema pueden tener consecuencias importantes, en cuanto a accidentes o daños, entran en juego aspectos cruciales de seguridad (*Safety*). Aparte del riesgo inherente al uso de los propios mecanismos, estos sistemas trabajan en zonas de riesgo donde la probabilidad de accidentes es de por sí más alta de lo habitual. Entre los factores posibles se pueden destacar [2]: mal funcionamiento del sistema de control (hw/sw), acceso indebido del personal a la zona de trabajo, errores humanos, roturas de partes mecánicas o liberación de energía almacenada.

En el contexto de la especificación de requisitos Safety para un sistema teleoperado, una de las principales deficiencias es la falta de integración de dicha especificación en el ámbito global de la especificación de requisitos del sistema, además de las pautas metodológicas necesarias. En este trabajo, nos hemos basado en el marco orientado a objetivos para la Ingeniería de Requisitos definido en ATRIUM [16], una metodología para la definición concurrente de requisitos y arquitectura software. Dicho marco se ha extendido para dar soporte al tratamiento de requisitos Safety. Nuestro enfoque recoge las recomendaciones del estándar ANSI/RIA R15.06-1999 [1], e integra los aspectos más relevantes de las propuestas de Lemos [11], para la división por modos de operación del sistema, y Douglas [2] respecto de patrones y pautas en la especificación de requisitos Safety. Nuestra propuesta se ha validado en el marco del proyecto europeo EFTCoR [4].

El trabajo se ha estructurado como sigue. En la sección 2, se presenta el marco conceptual empleado. La sección 3 describe el proceso para especificación de requisitos Safety en nuestro marco de Ingeniería de Requisitos Orientada a Objetivos. El caso de estudio presentado en la sección 4 describe cómo se ha aplicado dicho proceso en un caso real. La sección 5 presenta un ejemplo de los patrones definidos para la operacionalización de requisitos Safety. Las

Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos, JISBD'2005, pp.35-42
ISBN: 84-9732-434-X **© 2005 Los autores, Thomson**

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005.

conclusiones y trabajos relacionados finalizan el presente trabajo.

2. Marco conceptual

Con objeto de facilitar la comprensión del presente trabajo se hace necesario describir los conceptos utilizados en el contexto de los sistemas robóticos, en particular los tele-operados. En este tipo de sistemas, por su naturaleza, la probabilidad de sufrir accidentes es más elevada que en otros dominios. Es por ello que una precisa identificación, especificación y satisfacción de requisitos *Safety* es imprescindible.

Así, encontramos que primero hemos de definir qué se entiende por requisitos *Safety*, dadas las diferentes acepciones existentes en la bibliografía. El estándar ANSI/RIA R15.06-1999 utilizado como base en el presente trabajo, define requisitos Safety *como aquellos que ha de satisfacer un sistema robótico industrial para garantizar la seguridad del personal involucrado en su utilización*. En este trabajo se amplía este concepto incorporando las ideas de Leveson [14], a *cualquier daño o destrucción de propiedad o herida o daño de un ser viviente, espacialmente, de un ser humano*.

Para la descripción del funcionamiento de cualquier sistema robótico es necesario identificar y especificar las *tareas* que pueden ser proporcionadas por cualquiera de los elementos del sistema. Estas tareas son identificadas mediante un refinamiento intencional de los requisitos funcionales, hasta conseguir una granularidad acorde al uso que se quiere hacer de dicha especificación. Es la ejecución de dichas tareas la fuente potencial de los posibles daños o roturas que pueda ocasionar o tener el sistema porque, tal y como refleja el estándar ISO/IEC 9126 [9], los mayores riegos a la seguridad provienen de deficiencias en la funcionalidad, fiabilidad o usabilidad.

En la especificación de requisitos Safety se persigue la detección temprana de los *hazards* (amenazas) que puedan aparecer asociados a las diferentes tareas. Un *hazard* es cualquier fuente potencial de daño o rotura de una entidad del sistema, ya sea un usuario, elemento del entorno o del propio robot. De esta manera, durante el proceso de captura y especificación de requisitos Safety, el analista ha de identificar los posibles *hazards* del sistema y llevar a cabo su análisis a fin de determinar qué estrategia es la más adecuada para su gestión, en función del *riesgo* que lleven aparejados, es decir, *en función del daño o rotura que pueda sufrir cualquier entidad del entorno o del propio sistema*.

Todo ello pone de manifiesto que aunque el proceso a seguir para la identificación y especificación de este tipo de requisitos es similar al aplicado en otros contextos, sí exhiben algunas diferencias significativas. Además, el hecho de que los sistemas tele-operados sean *Safety critical* pone de manifiesto la relevancia de ofrecer al analista un proceso y una notación más específica que contemple los elementos antes comentados.

3. Identificando y especificando Requisitos Safety

Para la determinación del proceso de identificación y especificación de requisitos Safety se ha realizado la integración de las recomendaciones del estándar ANSI/RIA R15.06-1999 junto con las aproximaciones de Douglass y de Lemos. Se trata así tanto de sistematizar el proceso de identificación y especificación como de proporcionar una notación que ayude en el proceso de análisis.

Para la definición de los requisitos Safety de entornos tele-operados se han definido una serie de pasos que se describen seguidamente. Los primeros pasos (I-II) están relacionados con la especificación del comportamiento del sistema, empleando el Modelo de Objetivos de ATRIUM. Éste emplea el estándar ISO/IEC 9126 como punto de partida para proporcionar una adecuada organización de la especificación.

I. *Identificar los modos de operación del sistema*, tal y como recomienda Lemos para sistemas de control, como es nuestro caso. Ello nos permitirá proporcionar una adecuada organización de la para cada modo de operación se realizará un refinamiento intencional desde objetivos a requisitos hasta obtener una granularidad adecuada para la identificación de tareas.

II. *Identificar Tareas (T_i) asociadas a cada uno de los modos de operación.* De la ejecución de las tareas es de donde proceden los posibles *hazards* del sistema. Dichas tareas se

especificarán como requisitos dentro del modelo de objetivos, mediante un refinamiento AND/OR/XOR aplicado a partir de los objetivos identificados en el punto anterior. La relación a aplicar dependerá de si tienen que incorporarse todas o algunas o sólo una, respectivamente, de las tareas identificadas para conseguir que dicho modo de operación se realice en el sistema. A su vez, también proporciona una mayor visibilidad sobre qué componentes del sistema se ven involucrados en los modos de operación.

Tanto la identificación de las tareas como los modos de operación nos proporcionan la especificación del comportamiento del sistema. Esto es necesario como paso previo a la identificación de los requisitos Safety del sistema, ya que su ejecución es la que puede ocasionar la aparición de daños o roturas. Una vez se ha establecido esa vista del sistema se determinarán los requisitos Safety, utilizando la correspondiente categoría del ISO/IEC 9126. Así, al proceso anterior se le añaden los siguientes pasos:

III. *Determinar objetivos Safety del sistema*. Para cada una de las tareas identificadas T_i se evaluará su capacidad para provocar algún daño. Si éste es el caso, se especificará como objetivo Safety, *Safe* (T_i), conseguir salvaguardar la tarea T_i. Esto representa implícitamente una relación de composición entre un requisito Safety con un requisito funcional a nivel de tarea.

IV. *Determinar Hazards del sistema*. Una vez que un objetivo Safety, *Safe* (T_i), ha sido identificado se determinan los hazards asociados que pueden provocar un accidente, y se especifican como sub-objetivo *Manage* (Hz_j). Se establecerá a su vez una relación, AND/OR/XOR con aquel objetivo Safety, *Safe* (T_i), que ha determinado su identificación. La relación a aplicar dependerá de si tienen que administrarse todos o algunos o sólo uno, respectivamente, de los hazards identificados para conseguir asegurar adecuadamente la tarea. Hemos de tener en cuenta que un mismo hazard puede aparecer como refinamiento de más de un objetivo Safety.

V. *Identificar Riesgos (R_k) asociados a cada par Safe* (T_i)—*Manage* (Hz_j). A fin de determinar la estrategia más adecuada para gestionar el Hazard, se han de identificar los riesgos asociados. Un mismo Hazard puede conllevar distintos riesgos dependiendo de la tarea sobre la que sea evaluado. Así, de acuerdo con el marco de trazabilidad presentado en [12], la especificación del riesgo R_k se realizará mediante la introducción de un artefacto de tipo *rationale* aplicada sobre la relación de refinamiento en el par *Safe* (T_i)—*Manage* (Hz_j). Asimismo, tal y como recomienda Douglass para la evaluación de riesgos, se incorpora en la especificación de dicha *rationale* otra información necesaria: identificación de las causas (software, hardware, humanas,…) que pueden originar la aparición del hazard, la reacción necesaria para su tratamiento y los tiempos máximos de exposición, de detección y de tolerancia que se admiten.

VI. *Determinar Categoría de Reducción del Riesgo (R_i)* sobre cada relación *Safe* (T_i)—*Manage* (Hz_j), teniendo en cuenta los riesgos R_1 ... R_n asociados a la misma. Para ello es precisa la evaluación de tres atributos:

a. *Severidad (Severity)*. Nivel de daño que puede sufrir un elemento de entorno o del propio sistema. El ANSI/RIA R15.06-1999 proporciona una tabla para realizar su evaluación. Se ha planteado la necesidad de realizar su modificación a fin de considerar el concepto de seguridad en su sentido más amplio, es decir, incluyendo el daño a elementos del sistema o del entorno y no sólo a personal. Como resultado, se han extendido las interpretaciones asociadas a los diferentes valores proporcionadas por el estándar, tal y como se muestra seguidamente:

S2	Daño a personal más allá de primeros auxilios *Rotura irreemplazable en tiempo y coste admisibles de componente del sistema.*
S1	Daño a personal salvable con primeros auxilios *Rotura reemplazable en tiempo y coste admisibles de componente del sistema*

b. especificación para el subsiguiente proceso de análisis. Estos modos son especificados como objetivos del sistema, mediante un refinamiento de la característica Idoneidad (*suitability* en la ISO/IEC 9126). Así, *Exposición (Exposure)*. Frecuencia de exposición al hazard. Para ello ANSI/RIA

R15.06-1999 define las categorías *E2* como *frecuente* y *E1* como *poco frecuente*.

c. *Evitación* (Avoidance). Probabilidad de evitar la exposición al hazard. Para ello ANSI/RIA R15.06-1999 define las categorías *A2* como *no probable* y *A1* como *probable*.

Este estándar reconoce ocho posibles combinaciones, representadas en la Tabla 1 junto con la acción recomendada sobre el hazard, según los valores de los tres atributos. En función de qué combinación de valores sea la que esté presente en un determinado par *Safe* (T_i)—*Manage* (Hz_j) para un determinado riesgo R_k, se seleccionará la estrategia apropiada para el tratamiento del hazard.

Una vez aplicada la estrategia para los diferentes pares, es necesario volver a evaluar las categorías de riesgo debido a las posibles modificaciones en el conjunto de los hazards. Este proceso se repetiría hasta que todos los hazards del sistema estén en un nivel "tolerable", es decir, un nivel razonable de riesgo para este tipo de sistemas. Estos serían considerados riesgos residuales para el sistema.

Tabla 1. Categorías de Reducción de Riesgo

Expo.	Evitación	Severidad	Categoría Reducción	Acción sobre Hz
E2	A2	S2	R1	Eliminar/ Sustituir
		S1	R2C	Prevenir/ Cesar
	A1	S2	R2A	
		S1	R3A	Aislar
E1	A2	S2	R2B	Prevenir/ Cesar
		S1	R3A	Aislar
	A1	S2	R3B	
		S1	R4	Avisar/ Entrenar/ Proteger

4. Caso de estudio

Esta sección ilustra cómo se ha aplicado la propuesta anterior en el desarrollo de los requisitos *Safety* del proyecto industrial EFTCoR [5]. Su objetivo fundamental está relacionado con el desarrollo de una familia de robots para operaciones de mantenimiento de cascos de buques.

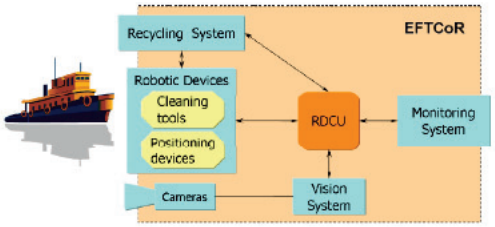


Figura 1. Sistema EFTCoR

La plataforma de teleoperación está integrada por diferentes subsistemas. Entre ellos, la Unidad de Control de los Dispositivos Robóticos (RDCU) que centraliza toda la funcionalidad relativa al gobierno del robot (Figura 1). Los componentes mecánicos del robot son las herramientas de limpieza y los dispositivos de posicionamiento primario y secundario. El primario (Figura 3) permite el acercamiento del robot al barco y el secundario el posicionamiento de las herramientas de limpieza. La definición arquitectónica del EFTCoR es especialmente compleja debido a las restricciones que han de satisfacerse para permitir un comportamiento seguro.

Durante la elaboración de la especificación se detectaron 6 modos diferentes de operación (paso I): *funcionamiento*, *mantenimiento (calibración, aprendizaje, diagnóstico* y configu*ración)* y *parada segura*. Para cada uno de los modos se definieron qué subsistemas estaban involucrados y las tareas asociadas (paso II). Por limitaciones de espacio, este artículo se centra únicamente en el modo *funcionamiento* y en el sistema de posicionamiento primario del robot.

Tabla 2. Simbología de la Figura 2

Símbolo	Objetivo
T_i	*Safe* (T_i)
Hz_j	*Manage* (Hz_j)

Tabla 3. Extracto de Tareas del Sistema Primario

Tarea	Descripción
T1	Mover articulación brazo
T2	Mover articulación raíles
T3	Parar el movimiento del primario

Una vez definida la operación del sistema se establecieron sus requisitos *Safety*. Para ello se determinaron qué tareas había que asegurar, *Safe* (T_i), especificándolas como objetivos Safety del sistema (paso III). En la Figura 2 (la simbología se

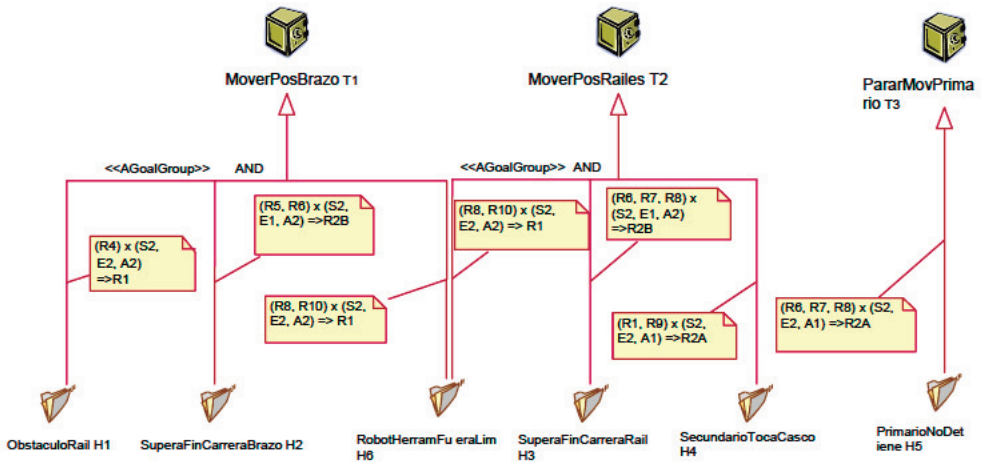


Figura 2. Extracto del Modelo de Objetivos

describe en la Tabla 2), podemos apreciar un extracto del modelo de objetivos en el que aparecen las tareas a asegurar (Tabla 3).

Cada objetivo Safety, *Safe* (T_i) fue refinado en subobjetivos, *Manage* (Hz_j) (paso IV). Un extracto de los Hazards del Sistema Primario identificados se muestra en la Tabla 4.

La aplicación del paso V determina la identificación de los riesgos que aparecen en cada par *Safe* (T_i) — *Manage* (Hz_j). Cada uno de los riesgos identificados aparece descrito en la Tabla 5. En la Figura 2 se han descrito mediante comentarios sobre las relaciones para facilitar su examen. Junto con la identificación de los riesgos, también se hace necesaria la descripción de otra información para el posterior proceso de evaluación. Por ejemplo, para el par *Safe* (H6)-*Manage*(T2), se establecen R8 y R10 como posibles riesgos y se describe *Error Sw. de control robot* como posible *causa* de la aparición del hazard; la realización de una *Parada de emergencia* como posible *reacción* ante su aparición; la determinación *media* como *probabilidad de aparición*; etc.

Como resultado de la aplicación del paso VI, se determinaron las categorías de reducción de riesgo a aplicar en cada caso concreto. Podemos apreciar, por ejemplo, que para el par *Safe* (H6)-*Manage* (T2) se determinó, teniendo en cuenta los riesgos (R8, R10), que la evaluación de severidad, exposición y evitación era (S2, E2, A2). Empleando las categorías de reducción mostradas en la Tabla 1 se determinó que era R1 la que debía aplicarse.

Tabla 4. Extracto de Hazards del Sistema Primario

Hz	Descripción
H1	El primario se está moviendo y se encuentra con un obstáculo en rail.
H2	Se supera el fin de carrera de articulación primario (brazo).
H3	Se supera el fin de carrera de articulación primario (sobre raíles).
H4	El secundario toca la superficie del casco
H5	El primario (brazo y raíles) no se detiene.
H6	El robot coloca el punto herramienta fuera del estereoradio de limpieza estando en modo funcionamiento.

Tabla 5. Extracto de Riegos del Sistema Primario

Risk	Descripción
R1	Rotura herramienta o componentes mecánicos secundario.
R2	Rotura mecánica articulación brazo o raíles.
R3	Daños en la superficie del casco.
R4	Daños a operario, a primario o a objeto.
R5	Rotura mecánica primario raíles y volcado del robot.
R6	Rotura mecánica primario articulación sobre raíles o brazo.
R7	Rotura del secundario
R8	Daños al operador
R9	Rotura mecánica secundario
R10	Daños a objetos del entorno.

La aplicación de las correspondientes categorías de reducción de riesgo conlleva una re-evaluación del grafo. Por ejemplo, dado que la

categoría de reducción fue R1 para el par *Safe* (H6)-*Manage* (T2), la introducción de las acciones correctoras aumentarían la probabilidad de evitar la exposición al hazard y por tanto, prevenir que éste suceda. Por ello, en la siguiente evaluación la categoría de reducción que podría aparecer sería R2C, que se considera "tolerable" en el caso del EFTCoR.

5. Patrones Safety de Operacionalización

El estándar ANSI define una serie de estrategias para tratar los hazards, aplicables en función de la categoría de reducción de riesgo. El nivel de detalle que aporta dicho estándar se ilustra en la Tabla 6, donde se describe la categoría de reducción de riesgo R2A. Sin embargo, en un dominio específico como es el caso del EFTCoR, es importante llegar a un nivel más detallado de descripción que facilite el desarrollo de estos sistemas. Por ello aparece de forma natural el establecimiento de un catálogo de patrones de reducción de riesgo clasificados conforme a la categoría de reducción. Dichos patrones, de forma similar a los propuestos por Gamma et al. [6], describen soluciones a problemas que aparecen reiteradamente. Para facilitar su comprensión, se muestra un ejemplo concreto de uno de dichos patrones desarrollado en el contexto del EFTCoR.

Tabla 6. Categoría de reducción R2A

R2A Circuitos de seguridad Control Fiable (basados en hardware o controladores software o firmware)
• *Monitorización genera señal de parada y warning.*
• *Estado seguro hasta eliminación del fallo.*
• *Controlar Modos Comunes.*
• *Detección en el momento del fallo.*

El sistema de posicionamiento primario mostrado en la Figura 3 tiene una altura de 12 m y un peso de más de 20 toneladas que hace imprescindible considerar requisitos *Safety* relacionados con el movimiento. La grúa posee en su zona central un brazo articulado de 2 toneladas que lleva asociado en su extremo el sistema de posicionamiento secundario: una mesa XYZ que incluye la herramienta de limpieza (no aparece en la fotografía). Es imprescindible que el sistema garantice un movimiento seguro del brazo acorde con los comandos recibidos. Analizando con más



Figura 3. Posicionamiento primario raíles (verde) más brazo (amarillo) del EFTCoR

detalle el caso particular de la amenaza H5 (el brazo del primario no se detiene) se le pueden asociar las siguientes causas:

- La sensorización ligada a los motores que mueven el brazo falla.
- Se produce un corte en el suministro eléctrico.
- La unidad de control no funciona correctamente (fallo del Hw o un error Sw).

Esta amenaza H5 tiene asociado el riesgo de rotura de cualquiera de los componentes e incluso la precipitación de partes mecánicas a la zona de trabajo con daño al operador (R6, R7, R8). La categoría de reducción de riesgo en la que encaja dada su severidad y posibilidades de ocurrencia es R2A. Las restricciones que impone dicho estándar para esta categoría de riesgo se detallan en la Tabla 6. La Figura 4 muestra el esquema de despliegue que cumple la categoría R2A de reducción del riesgo asociado a la amenaza

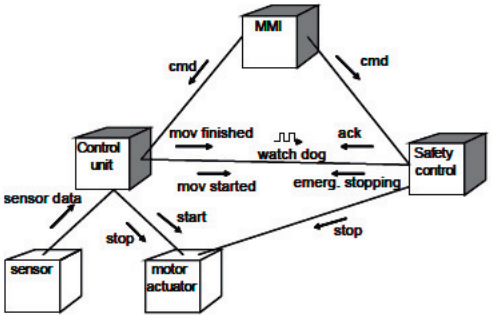


Figura 4. Despliegue considerando safety para la amenaza H5

descrita. Sin entrar en excesivos detalles del modelo formal subyacente mediante máquinas de estados, la descripción se resume a continuación:

- Cuando se recibe un comando de movimiento, el nodo de interfaz (MMI) lo dirige simultáneamente tanto al encargado de su procesamiento (nodo *control unit*) como al nodo redundante dedicado a monitorizar las posibles amenzadas (nodo *Safety control*).
- El nodo de control recoge desde un sensor la posición actual de la articulación y controla directamente el funcionamiento del motor. El nodo de monitorización parará el motor si detecta un funcionamiento incorrecto.
- Justo antes de iniciar la ejecución del comando, el nodo de control envía un mensaje al nodo de monitorización indicándole el inicio del movimiento. A partir de este instante, la unidad de control envía al nodo de monitorización, mediante pulsos periódicos (*watch dog*), el valor actual leído por el sensor. El nodo de monitorización responde a este tick con una señal de reconocimiento (*ack*) que incluye como parámetro el valor que ha estimado y se corresponde con la posición del motor. Debe destacarse que ambos nodos realizan un cómputo detallado de la curva de posiciones discretas por las que va pasando el brazo del robot. Este cómputo discretizado en el tiempo se hace partiendo de la lectura inicial del sensor y del comando que se ejecuta. Una discrepancia en los valores calculados supone una anomalía en el funcionamiento. En cuanto uno de los dos nodos detecte una discrepancia respecto del otro en la estimación de valores de posición se genera una parada de emergencia.

Las distintas situaciones de error que se pueden producir y que llevarían a la parada segura del robot son las siguientes: (1) el diseño y construcción del robot está hecho de forma que ante un fallo total el sistema queda enclavado de manera segura; (2) si cualquiera de los dos nodos de cómputo no funciona como debe (por cuestiones Hw o Sw) o falla la red de comunicaciones, entonces el otro detectará la discrepancia con lo que se iniciará la parada de emergencia, y (3) si el brazo robótico se mueve por razones ajenas al sistema entonces el nodo unidad de control detectará la discrepancia. Para este último caso, se hace necesario que la unidad de control lea periódicamente los datos del sensor incluso si no se está ejecutando ningún comando de movimiento.

6. Trabajos relacionados

Uno de los enfoques más populares para identificar, evaluar y mejorar los requisitos Safety son los *fault trees* [8]. Estos proporcionan una notación gráfica y un soporte formal que facilitan el análisis desde la perspectiva de los fallos de un sistema y sus causas. Sin embargo, no ofrecen un marco global para la especificación de requisitos. El refinamiento de objetivos en nuestra propuesta es similar al realizado por los *fault trees*, pero en nuestro caso el análisis de requisitos Safety viene integrado y derivado a partir de los requisitos funcionales del sistema.

Letier et al [13], han propuesto también la utilización de KAOS para especificación de requisitos Safety. Para la definición de una especificación que satisfaga las necesidades Safety, han introducido el concepto de *obstáculo* como *un conjunto de comportamientos no deseables*; de manera que la aparición de dichos obstáculos implica obstrucción en la satisfacción del objetivo y su negación, por el contrario, genera las precondiciones necesarias para su satisfacción. Los objetivos Safety del sistema en desarrollo se especifican formalmente mediante una lógica temporal, y los obstáculos (que podríamos equiparar a los hazards) son obtenidos de forma automática mediante la negación de los objetivos Safety, siguiendo los patrones definidos en su propuesta [10]. El problema se plantea en la aplicación de dicha propuesta, ya que no ofrece una guía para la identificación de objetivos Safety, ni recoge información de severidad, exposición, etc., definida en el estándar ANSI como necesaria para el correcto análisis de estos sistemas y que sí se plantea en la presente propuesta.

7. Conclusiones

En los sistemas teleoperados la especificación de requisitos Safety presenta importante desafíos tanto por la compleja combinación de componentes hardware/software, como por el grado de exposición a daños en el sistema o su

entorno. En este trabajo hemos presentado una propuesta unificada para la especificación de requisitos en este dominio que provee una integración y derivación de los requisitos Safety con el resto de requisitos del sistema. Se ha empleado para ello la aproximación orientada a objetivos por sus ventajas inherentes para el análisis y evaluación de alternativas, así como de verificación gracias a la formalización del modelo (no descrito por limitaciones de espacio).

Por otra parte, siguiendo las recomendaciones del estándar ANSI/RIA R15.06-1999 y de otras propuestas relevantes en el ámbito Safety hemos establecido pautas metodológicas para la especificación de requisitos de un sistema tele-operado. Para el dominio específico de sistemas teleoperados resulta interesante contar con un catálogo de requisitos Safety y sus respectivas operacionalizaciones con el propósito de reutilización, especialmente en el contexto de líneas de producto [7]. En este sentido, nuestra propuesta establece una descripción inicial de patrones en el ámbito del proyecto EFTCoR.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el proyecto CICYT DYNAMICA TIC2003-07804-C05-01 y TIC2003-07804-C05-02.

Referencias

[1] ANSI/RIA R15.06-1999. American National Standard for Industrial Robots and Robot Systems, Safety Requirements. Robotic Industries Association, 1999.

[2] B. P. Douglass, Real-Time Design Patterns. Robust Scalable Architecture for Real-Time Systems. Reading, Addison-Wesley. 2003.

[3] EFTCoR. Requisitos del Sistema. Anexo I: Resumen de Requisitos Funcionales de los sistemas robóticas. Subproyecto ANCLA, Abril 2004. http://dynamica.dsic.upv.es:8090/

[4] EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. (GROWTH G3RD-CT-00794) EU Project within the 5th Framework Program, 2003.

[5] C. Fernández, J.A. Pastor, P. Sánchez, B. Álvarez, A. Iborra: "Co-operative Robots for Hull Blasting in European Ship repair Industry. Robotics and Automation Magazine (RAM)", special issue on Industrial Robotics Applications, September 2005.

[6] E. Gamma, R. Helm, R. Johnson, J.Vlissides, Patrones de Diseño, Addison Wesley. 2003

[7] J. van Gurp, J. Bosch and M. Svahnberg "On the notion of variability in software product lines", Proc. Working IEEE/IFIP Conference on Software Architecture, pp 45—54.

[8] K.M. Hansen, A.P. Ravn, V. Stavridou, From Safety analysis to software requirements, IEEE Transactions on Software Engineering, 24 (7), Jul 1998, 573-584.

[9] ISO/IEC Standard 9126-1 SE- Product Quality-Part1: Quality Model, ISO Co. Office, Geneva, June 2001.

[10] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", IEEE Transactions on SE, Special Issue on Exception Handling, 26 (10), October 2000, 978-1005.

[11] R. Lemos, A. T. Saeed, "Analyzing Safety Requirements for Process-Control Systems", IEEE Software, 12(3), 42-53, May 1995.

[12] P. Letelier, E. Navarro, V. Anaya, "Customizing Traceability in a Software Development Process", 13th Inter. Conf. on Information System Development, Vilnius, Lithuania, September, 2004.

[13] E. Letier and A. van Lamsweerde, High Assurance Requires Goal Orientation, Proc. International Workshop on Requirements for High Assurance Systems, Essen, Sept. 2002.

[14] N. Leveson. Safeware: System Safety & Computers, Addison-Wesley, 1995.

[15] F. Modugno, N.G. Leveson, J.D. Reese, K. Partridge and S.D. Sandys, "Integrated Safety Analysis of Requirements Specifications," Requirements Engineering 2 (2), 1997.

[16] E. Navarro, P. Letelier, I. Ramos, "Goals and Quality Characteristics: Separating Concerns", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Oct. 25, 2004, Vancouver.

[17] T.P. Nelly and J.A. McDermid, A systematic approach to Safety case maintenance, Reliability Engineering and System Safety 71 (2001) 271–284.

[18] C. Potts, "Using Schematic Scenarios to Understand User Needs", Proc. ACM Symposium on Designing Interactive Systems: Processes, Practices and Techniques, University of Michigan, August 1995.

A METAMODELING APPROACH FOR REQUIREMENTS SPECIFICATION

ELENA NAVARRO
UCLM
Albacete, Spain

JOSE ANTONIO MOCHOLI
UPV
Valencia, Spain

PATRICIO LETELIER
UPV
Valencia, Spain

ISIDRO RAMOS
UPV
Valencia Spain

ABSTRACT

There are many Requirements Engineering approaches and techniques that help to specify, analyze and validate requirements. However, they are neither widely accepted nor widely used by the industrial software community. One of the main problems faced when applying them is to what extent they can be easily adapted to the specific needs of the project. This has often led to unsatisfactory requirements management in industrial software development. In this work, a proposal for requirements modeling is presented that allows the integration of the expressiveness of some of the more relevant Requirements Engineering techniques by taking advantage of metamodeling. This proposal focuses on scalability with respect to expressiveness and adaptability to the application domain to establish some basic guidelines and extension mechanisms that lend coherence and semantic precision. A case study and 4 tool support are presented to describe the application of this proposal in a real-life project.

Keywords: Software Requirements Engineering, Goal-Oriented, Aspect-Oriented, Variability, Use Case, UML, Metamodeling.

INTRODUCTION

The drawbacks in Requirements Engineering (RE) and their negative impact on the success of software development projects have led to a great deal of research in this field. However, these supposed advances are not widely applied in industrial software developments. Some critical obstacles for applying RE techniques are: (a) the increasing adoption and integration of different requirements specification approaches to development projects; (b) the prevalence of adaptation over adoption of method; and, (c) the increasing importance of the definition of domain specific languages (and approaches) in a development environment which tends toward being model driven.

A point has been reached where it is absolutely necessary to integrate knowledge and techniques that have already been developed and facilitate their use in real-life projects. This integration must consider scalability mechanisms for use at different levels of sophistication as well as providing adaptability in order to make their application in specific domains smoother.

This work aims at elaborating a proposal for integrating different RE techniques in a common framework and providing some guidelines for adaptation to the specific needs of the project. Our approach is based on metamodeling as a way of providing a smooth integration and scalability of RE concepts. In this sense, we provide some guidelines to extend a core set of concepts, in such a way that a proper semantic coherence can be maintained. To establish the expressiveness needed for requirements specification, we have studied five approaches that we consider to be highly representative in the current state-of-the-art in Requirements Engineering: traditional (based on the IEEE 830-1998 (18)), Use Cases (7), Goal-Oriented (27), Aspect-Oriented (39) and variability management (15). We have not explicitly included all the details found in each requirement technique. We have started from a limited set of concepts, so that it will be simple and easy to reach a consensus.

Our approach took shape in the context of a medium-size real-life project, where we faced the problem of integrating several RE techniques to provide support for a complex requirements specification, including critical non-functional requirements (safety, performance, interoperability, etc.) and requirements for a product line with sophisticated variability concerns. These characteristics were propitious enough to apply an Action-Research method, allowing us to define and improve our approach iteratively through continuous discussion with the analysts.

This work is organized as follows. The following section establishes a global view of the required expressiveness associated with requirements specifications. This is achieved by describing the essential concepts included in five relevant RE techniques. Section 3 presents our approach for requirements specification based on metamodeling, which provides integration facilities to include requirements concepts of different RE techniques. Section 4 describes the application of our proposal in a case study along with the support tool. Finally, related work in section 5 and conclusions in section 6 conclude this paper.

APPROACHES TO REQUIREMENTS ENGINEERING

RE has been recognized as one of the cornerstones for the success of a project (34). Zave (49) provides one of the best known definitions of RE: "Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families." This definition emphasizes

Navarro, E., Letelier, P., Mocholí, J. A.,Ramos, I. A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems, XLVI(5), Special Issue on Systems Analysis and Design. 2006.

how important the relation between the expectations of the software and the real system-to-be developed is.

Several challenges have been identified related to research in RE that need to be dealt with in the years ahead. Nuseibeh and Easterbrook (34) have pointed out two which are closely related to the intentions of this work. First of all, they describe the need for richer models for capturing and analysing non-functional requirements. This means that a Requirements Model will have to cope with non-functional requirements as an assurance of usefulness. Secondly, the reuse of requirements models for specific application domains will greatly lighten the load involved in developing them from scratch. It will therefore be a real advantage to provide specific tools and techniques to customize these reused requirements models according to specific needs.

Consequently, we have selected five approaches to RE focusing mainly on the integration of their expressiveness. They have been selected as being highly representative of the state-of-the-art in this field and, moreover, are the subject of a great deal of research too.

Standard IEEE 830-1998 Approach

Standard recommendations are a reference point with regard to the content expected in a Software Requirements Specification (SRS). The IEEE 830-1998 standard (18) proposes a template which identifies several requirements artifacts, the main ones being described in a section called "Specific Requirements." These artifacts include: "External interfaces" for user, hardware, software and/or communications, "Functions" of the software, "Constraints" to software; "Design Constraints," etc.

This standard does not propose details with regard to the attributes that each of these artifacts should possess. However, it does include some recommendations to deal with potential organizations of specific software requirements that are based on "operation modes of the system," "user classes," "objects," "characteristics," "stimuli," "responses" or "functional hierarchies." Nevertheless, in practice, when the number of requirements and/or the level of complexity (due to the high number of relationships between them) are significant, those recommendations are not enough. Finally, IEEE 830-1998 does not provide any assistance to the process of elaboration or analysis of the requirements.

Use Case Approach

Use Case modeling (7) has been widely embraced by the industrial community because its straightforward notation and application allows stakeholders to easily understand the requirements specification. This facilitates the elicitation and validation of the requirements. In a Use Case diagram, we mainly distinguish the following artifacts:

Use Cases (UC) represent an atomic functionality which the system offers to the environment for achieving some specific goal. Basically, templates for specifying UC usually include other artifacts such as: preconditions, post-conditions, communication steps between the environment and the system to obtain a desired service, etc.

Actors represent the environment of the system and can be users, devices or any other system that interacts with the system under development. The name of the *Actor* describes the role that is played in that interaction.

Relationships which include **Communication** (interaction between the Actor and the Use Case); **Generalization** (applicable both to Use Cases and Actors to establish specialization hierarchies); **Include** and **Extend** (to factorize an original UC).

By means of the Include and Extend unidirectional dependencies, the relationships between requirements are specified. Both of them are factorizations of UC specifications but with a different purpose. The Includes relationship permits a UC to be reused in other UC specifications; and the Extends relationship simplifies the specification of a complex UC. Additionally, UCs do not allow hierarchical refinement, making it important to decide the proper granularity level of functionality that a UC should have and to correctly exploit Extends and Includes relationships (see (45) for more details about analysis of UCs). Consequently, the simple and easy notation of UCs is actually one of their major inconveniences, in situations where modeling has to be rigorous and/or precise.

Goal Oriented Approach

One of the main activities in the RE process is related to requirements analysis (to determine conflicts, dependencies and alternatives to satisfy the construction of software). This aim has inspired Goal-Oriented proposals such as KAOS (8) or the NFR Framework (5). Although there are a wide number of proposals (25), some concepts are common to all of them:

Goal describes why a system is being developed. In order to identify it, both functional goals (expected services of the system) and non-functional goals related to the quality of services (constraints on the design, quality attributes, etc.) should be determined.

Agent is any active component, either from the system itself or from the environment, whose cooperation is needed to define the operationalization of a goal.

Refinement Relationships: AND/OR/XOR relationships allow the construction of the goal model as an acyclic directed graph. These relationships are applied from generic goals toward subgoals until they have enough granularity to be operationalized.

This approach is especially appropriate for analyzing the specification for two reasons. First, its ability to specify and manage positive and negative interactions among goals (5) allows the analyst to weigh up different design alternatives. Secondly, its capabilities regarding traceability from low-level details to high-level goals (**concerns**) (8) makes it especially suitable to bridge the gap between architectural and requirements models. In spite of these advantages, Goal-Oriented approach has not been widely embraced by the industrial community, mainly because hitherto not much attention has been paid to the so-called "-ilities" in the requirements specification (34).

Aspect-Oriented Approach

This approach has emerged from the implementation level (26) as a mechanism to manage code complexity. With this aim in mind, a set of techniques has been provided that allows the analyst to specify each concern of the system separately. Afterwards, these concerns are weaved according to the expected behavior of the system. The concerns of the system can refer to either its functionality or any other issue such as security or distribution. In this way, the code that would arise tangled in many locations can be managed by modularizing it as an aspect.

There are several recent proposals that promote the detection and description of aspects at early stages of development (such as the design (46) and requirements phases

(39)) in order to satisfy the closure property (12). This is how the term Aspect-Oriented Requirements Engineering (AORE) emerged. AORE does not have its own notation or expressiveness and current proposals have developed their own notations. Nonetheless, we can enumerate a set of concepts common to all of them:

Concern refers to the interests of the system, which can be either functional or non-functional.

Crosscutting is a relationship among concerns that arises whenever a given concern interacts with other concerns (either by constraining, extending, etc.). A more detailed description of potential crosscutting relationships can be found in (38).

The main purpose of AORE is to improve the crosscutting management and to establish the composition relationships between specifications. This is done by encouraging the **separation of concerns** in a similar way to the role played by **weaving** in Aspect-Oriented Programming. One of the main difficulties whenever this approach is applied is related to the lack of consensus about what an early aspect is and, in particular, how to detect such aspects.

Variability Management Approach

This approach helps the analyst to delay the decision of what functionality or quality aspects will be incorporated in the final system for as long as possible. This approach has been successfully applied in two areas: Software Product Lines (SPLs) (6) and Dynamic Software Architectures (DSA) (37). Traditionally, most of these proposals have dealt with the management of variability at the architectural level by establishing a central architecture, along with a set of components that can be evolved or integrated according to the system needs. When variability identification is delayed at the architectural level a problem related to PL and DSA arises because the number of potential systems and the capability of adaptation, respectively, are more limited (15). Thus, the early identification of the variability which is performed in the requirements phase (**early variability**) is a great advantage.

As in AORE, there is no widely accepted notation to specify the variability in the requirements phase. Instead of describing the different notations available, we present below the necessary concepts as designated by Maßen and Lichter (32):

Representation of common and variable parts. The notation should allow the expression of those assets that are shared among different products of a PL, or among different instances of DSA, and those that are specific to a specific product or instance. The notation should be able to represent both variation points and variants. A **variation point** indicates a specific variability in the specification. A **variant** is the specific realization of variability in a specific variation point. Each variation point has a **time link**, i.e., when the variability is resolved during the development process: design, analysis, running, etc. It is also important to define the **multiplicity** (both maximum and minimum) in each variation, which determines how many variants must exist at the same time in a product or architecture when the variability is removed.

Distinction between types of variability. The notation must allow the different types of variability to be expressed: a) **option** - when some specific variants can be selected in the instantiation process, b) **alternative** - when only a single variant can be selected and c) **optional alternative** - when zero or one alternative can be selected from those available.

Representation of dependencies between variable parts. Variants frequently have dependencies among one another that have to be represented (4). Two examples are dependencies **require** (when a variant must be selected if another variant is present) and **exclude** (when the selection of a variant implies that another variant cannot be selected).

A META-MODELING PROPOSAL FOR REQUIREMENTS SPECIFICATION

The current proposal has been defined taking into account the approaches described in section 2 and their need for expressiveness. The first obstacle to overcome when integrating these approaches and their notations is the wide diversity of terms and concepts, with many overlaps among them. We realized that a consensus would have been very difficult if we had attempted to define a global notation so as to cope with the entire expressiveness included. Therefore, we have decided to organize our work in two parts. The first part, described in section 3.1, defines a metamodel for the essential concepts that allow us to deal with generic expressiveness. By doing so, we could achieve a consensus more easily. The second part (see section 3.2) describes a process which specifies how this core metamodel can be tailored according to the specific needs of expressiveness.

A Metamodel for Requirement Specification

The core concepts and their relationships, which are taken from the described approaches, are shown in Figure 1. The essential concept in a requirements specification is *artefact*; which represents any kind of specification at a certain level of granularity. An *artefact* can be a complete SRS document, a piece of text representing a requirement, etc.

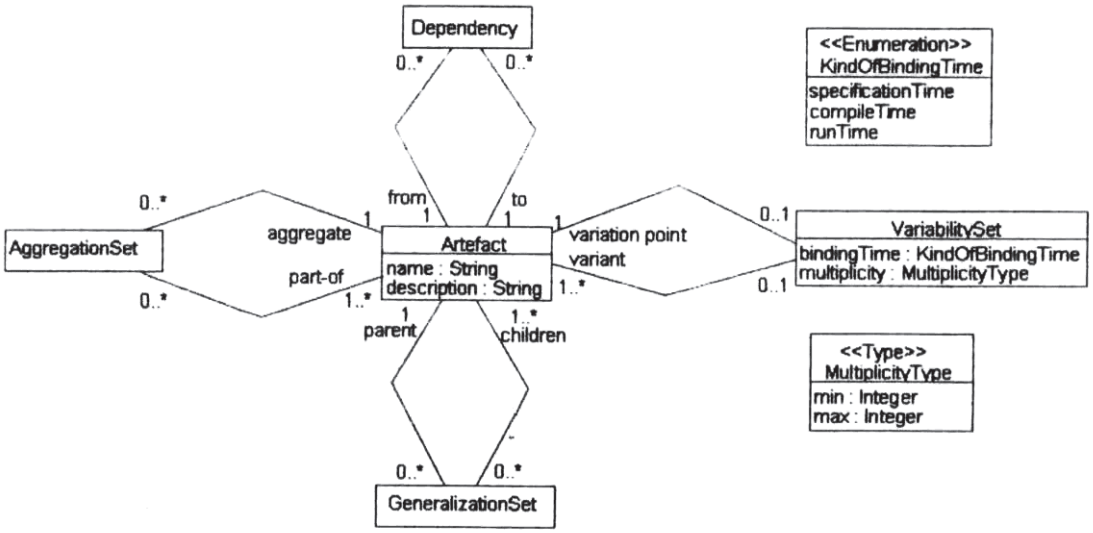
In addition to the artefacts, it is also necessary to establish relationships among artefacts of the SRS. Therefore, we have identified several kinds of relationships which are described below. An artefact can be refined forming a hierarchical structure. Three basic kinds of refinements are included: *VariabilitySet*, *GeneralizationSet* and *AggregationSet*. The *VariabilitySet* refinement specifies an artefact as a variation point and establishes its variants. This means that a starting point for a variability hierarchy can be determined from an artefact. During the specification of *VariabilitySet*, the attribute *multiplicity* and *bindingTime* must be specified (see section 2.5). Typically, *bindingTime* is assigned as: design time, compile time and runtime. For this reason, they have been included as the enumeration *KindOfBindingTime* and *bindingTime* has been typed with it.

When establishing refinements through *GeneralizationSet* or *AggregationSet*, the same artefact can belong to several hierarchies, either as parent/aggregated or as child/component, respectively. If *GeneralizationSet* is used, it allows the analyst to define hierarchies of alternative specializations from the same parent and to relate one child to more than one parent by multiple inheritance. If *AggregationSet* is employed, it means that an artefact can be part of several aggregate artefacts.

In addition to these three refinement relationships between artefacts, the dependency relationship has also been included in the metamodel. Perhaps this is the most conflictive relationship for consensus. For this reason, we have preferred to represent it in its most generic form, i.e., by means of *Dependency* which is applicable to artefacts in the core concepts. Several kinds of

dependency relationships are allowed between artefacts of different hierarchies. For example, non-functional requirements constrain functional requirements or goals, actors interact with use cases, variants require other variants, etc. Some kinds of dependency between artefacts are bidirectional, such as those used to specify conflicting goals or mutual exclusion between variants. However, others are unidirectional, such as *Extension* dependencies between Use Cases or *Require* dependencies between variants. Therefore, although in general terms we consider that dependencies are unidirectional, the metamodel also permits the specification of dependencies that in some cases may imply their inverse.

FIGURE 1
Metamodel for the Core Concepts



The following table summarizes concepts introduced in section 2 and how the mapping between them and those described in the proposed Metamodel (Figure 1) was established. We have to indicate that some concepts do not appear in the core Metamodel but are described by extending it according to the process described in section 3.2.

A Process for Customizing the Core

Once we have described the metamodel, it has to be tailored according to the specific needs of expressiveness. With this purpose in mind, we suggest the following steps to adapt and/or extend the metamodel which need not be applied sequentially, but in accordance with the analyst's preferences:

(i) To define the types of artefacts that are relevant to the model. This permits the inclusion of the needed artefacts by means of specialization. Whenever a new artefact has to be defined a specialization is specified using *Artefact* or another artefact already defined as a base class. This child artefact inherits any attribute that has been defined by its parent.

(ii) To establish the refinement relationships of interest. If necessary, any additional refinement type could be added as a specialization of any of the existing ones (*AggregationSet*, *GeneralizationSet* and *VariabilitySet*) or as a new and independent one.

(iii) To establish the types of dependency relationships between artefacts. This allows the definition of the relevant relationships between artefacts from the metaclass *Dependency* or any other already defined by means of specialization.

(iv) To include the attributes considered relevant to the types of artefacts, types of refinement and types of dependency. It is necessary to bear in mind that, whenever a new kind of artefact, refinement relationship or dependency relationship is defined, all the attributes defined in the parent are also inherited.

(v) To formalize artefacts and relationships. Those constraints to be applicable on artefacts and relationships have to be described. OCL (35) (Object Constraint Language) has been selected for this activity because it is a well-known and extended proposal for this purpose. In addition, there is a wide set of tools (36) that allow for simple consistency checks and type checking in terms of defined OCL constraints.

As can be observed in Figure 1, there is no direct connection between the relationships described and those artefacts on which they are to be applied, but rather artefacts and relationships are specified independently. This alternative provides us with an improved readability and comprehensibility of the Metamodel. However, this means that whenever a new relationship is defined, the analyst has to constrain the set of artefacts on which the relationship can be applied by means of the OCL step (v).

CASE STUDY: TAILORING THE CORE TO ACTUAL NEEDS

In this section, we will illustrate how the established metamodel can be extended and tailored according to some particular needs of expressiveness. With this purpose, we present a requirements model that we have developed and tested in a real system, which has been carried out within the European project Environmental Friendly and cost-effective Technology

for Coating Removal (EFTCoR) (11). The aim of this project is to design a family of robots capable of performing maintenance operations for ship hulls. The system includes operations such as coating removal, cleaning and re-painting of the hull.

TABLE 1
Mapping Concepts from Concepts of Approaches to RE (Rows) to our Metamodel (Columns)

Approaches to RE		Proposed Metamodel				
	Concept	*artefact*	*Dependency*	*AggregationSet*	*VariabilitySet*	*GeneralizationSet*
Use Case	**Use Case**	Can be described extending				
	Generalization					Realized by
	Communication					
	Include		Extending (*Include*)			
	Extend		Extending (*Extend*)			
Goal-Oriented	**Goal**	Extending				
	Agent	Extending (operationalization)				
	Refinement Rel.					
	AND			Realized by		
	OR/XOR				Realized by	
Variability Management	**Variant**	An artefact with a Variability or Generalization Rel.			Establishing this relation on an artefact	Establishing this relation on an artefact
	Variation point				Described by	Described by
	Time link				Using an attribute	
	Multiplicity				Using an attribute	
	Types of Variability				Using multiplicity	
	Require		Extending (*Require*)			
	Exclude		Extending (*Exclude*)			
AORE	**Concern**	Extending				
	Crosscutting		Extending (*Include, Extend*)			

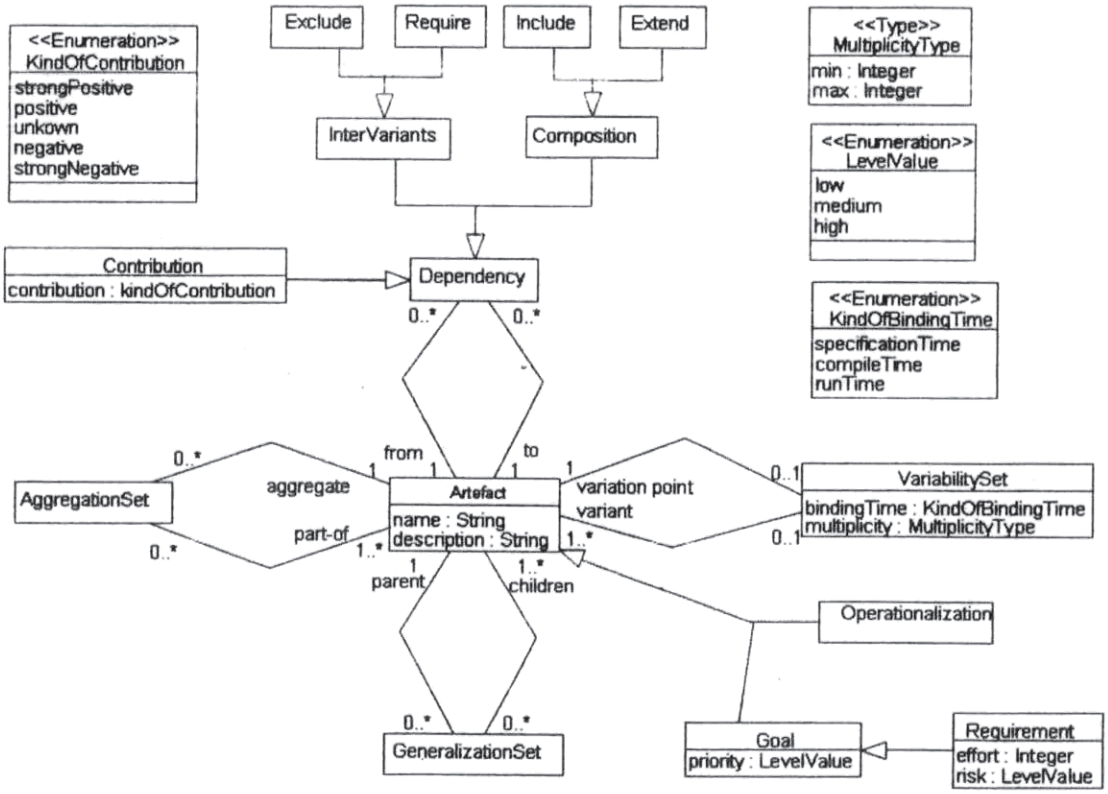
This project exhibits several specific needs in terms of requirements specification, such as, the variability inherent in the family of robots to be handled; the high incidence of non-functional requirements (reliability, performance, safety, etc) which crosscut functional ones; the need to evaluate alternative designs meeting system requirements; and, finally, a large specification where an appropriate organization is unavoidable. In this context of complexity, associated with the requirements specification and the emphasis in achieving reuse through a product family specification, we found favorable conditions for the application of Action-Research allowing us to solve a real problem by means of continuous refining of our approach throughout the 3 years of project duration. We play the researchers role, while the research aim is to define a suitable RE approach and produce the requirements specification in this project. The analysts of the project are the critical reference group and the stakeholders the consortium on behalf of the EFTCoR project.

Bearing in mind these needs, a requirements model was defined using ATRIUM (33), a methodology that guides the analyst through an iterative process, from a set of user/system needs to the instantiation of a Software Architecture. This requirement model is a Goal Model which is based on KAOS and the NFR Framework proposals. This Goal Model has been extended (32) by integrating the aspect-oriented approach, in order to achieve both the efficient management of crosscutting and the correct organization of the SRS. In its definition, a strategy for the separation of concerns has also been established based on the standard ISO/IEC 9126. We have also incorporated the variability management in order to apply the model to the definition of product lines and dynamic architectures.

Goal Model of ATRIUM

Figure 2 shows the metamodel of the ATRIUM Goal Model. The artefacts and the identified relationships shown in Figure 2 correspond to the application of steps described in section 3.2. By applying step (i), three kinds of artefacts are identified to support a Goal-Oriented approach (section 2.3): *Goal*, *Requirement* and *Operationalization*. *Goal* and *Operationalizations* are defined by inheriting from *artifact*. A *Requirement* is defined as a specialization of a *Goal*, but a *Requirement* must also be verifiable and assignable to an agent by means of an operationalization. Step (ii) is not applied because no additional refinement relationship is needed.

FIGURE 2
Metamodel of the ATRIUM Goal Model



We have defined a complete and disjoint hierarchy of types of dependencies by applying step (iii). On this hierarchy, *Intervariant*, *Composition* and *Contribution* specializations are identified. *Intervariant* expresses a dependency between variants. This dependency can be either *Require* or *Exclude*, using the semantics described in section 2.5. *Composition* allows the establishment of a composition dependency between artefacts. Following up this idea, the same dependencies as those applied to Use Cases are identified, i.e., *Include* or *Extend*; however, these relationships will model composition dependencies for crosscutting requirements (section 2.4). In addition, a *Contribution* dependency from one artefact to another is included to establish that the accomplishment of one artefact affects negatively or positively the accomplishment of another.

In the application of step (iv), the relevant attributes should be defined. In our case study, we have established the following attributes: *priority* for the metaclass *Goal*; *risk* and *effort* for the metaclass *Requirement*; and *contribution* for the metaclass *Contribution*. Their possible values have been declared at the corresponding enumeration shown in Figure 2.

Due to space limitations, we have not indicated the integrity constraints of the whole model when step (v) is applied; these constraints are applied in addition to those determined by associations and multiplicities defined in the metamodel. In the ATRIUM Goal Model, these constraints refer to the types of artefacts that can be related by means of a specific kind of dependency or refinement. Table 2 shows an example of how we have described which *Contribution* dependencies are allowed between *Operationalizations* and *Requirements* and also which *GeneralizationSets* are possible between *Requirements* or *Goals*.

MetaEdit+: Tool Support for Modeling and Metamodeling

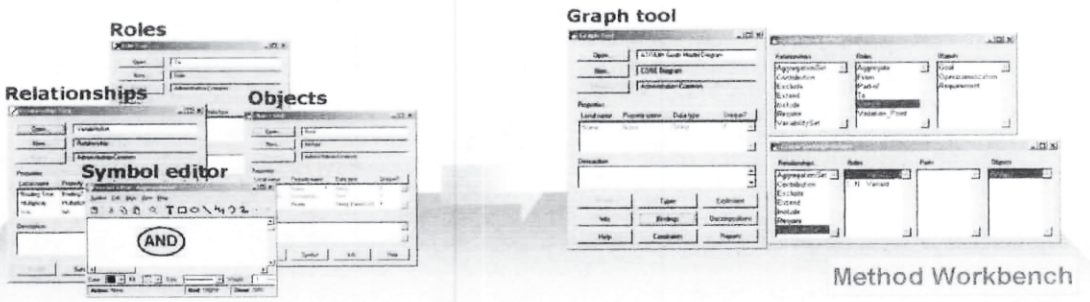
In order to provide tool support for our proposal we have studied three possibilities: (a) to implement a tool specially suited for our approach; (b) to define a UML Profile associated to our metamodel and use a CASE tool based on UML for providing visual modeling by means of UML stereotypes; and

(c) to use a Meta-CASE tool for defining directly our metamodel, attaining immediate modeling support. Option (a) is the most flexible but it requires more effort. We have tested option (b) (30) but we have found some important limitations in achieving full support for UML profiles in a CASE tool. Meta-CASE tools for Domain Specific modeling is a promising emergent technology thrust by current interest into Model Driven Development. Although we are not concerned with generating code from models, we basically need the same facilities, which include: support for metamodel definition, support for modeling according to the metamodel and access to the tool repository in order to generate reports. These reports can be formatted representations of models, new models generated by means of model transformation or following some mappings.

TABLE 2
OCL Constraints for ATRIUM Goal Model

Relationship	Constraint
Contribution	**context** Contribution **inv** allowedartefactsContribution: Self.from.oclIsTypeOf(Operationalization)**and** Self.to.oclIsTypeOf (Requirement)
Generalization	**context** GeneralizationSet **inv** allowedartefactsGeneralization: Self.parent.oclIsKindOf(Requirement)**and** Self.children->forAll(a:artefact | a.oclIsKindOf (Requirement))

FIGURE 3
Method Workbench: Tool Support for Metamodeling



Currently, we are using a Meta-CASE tool namely MetaEdit+ tool (29). It is a Domain Specific modeling tool that allows an easy metamodel specification and its exploitation by keeping models synchronized with any change performed on its metamodel. With this idea in mind, it integrates two facilities: Method Workbench (Figure 3), which provides the user with a tool suite for designing the metamodel; and MetaEdit+, which automatically gives the user full CASE tool functionality according to the metamodel description.

Method Workbench is based on the **GOPRR** metamodeling language (29) which defines the following concepts: **Graph**, **Object**, **Property**, **Relationship** and **Role**. Graphs are the graphical representations for a metamodel or part of it. Objects are kinds of entities in the metamodel. Relationships are kinds of n-ary connections among Objects. Relationships have a Role kind for each kind of Object that they connect. Finally, Property allows the specification of attributes for Graphs, Objects and Roles. Therefore, Graphs specifications contain binding information about what Objects can be connected by means of a specific Relationship and by playing a particular Role. The graphical appearance for Objects, Relationships and Roles is set by means of a Symbol editor, included in the Method Workbench, which allows metamodels to be visually customized.

Exploiting MetaEdit+ in our Case Study

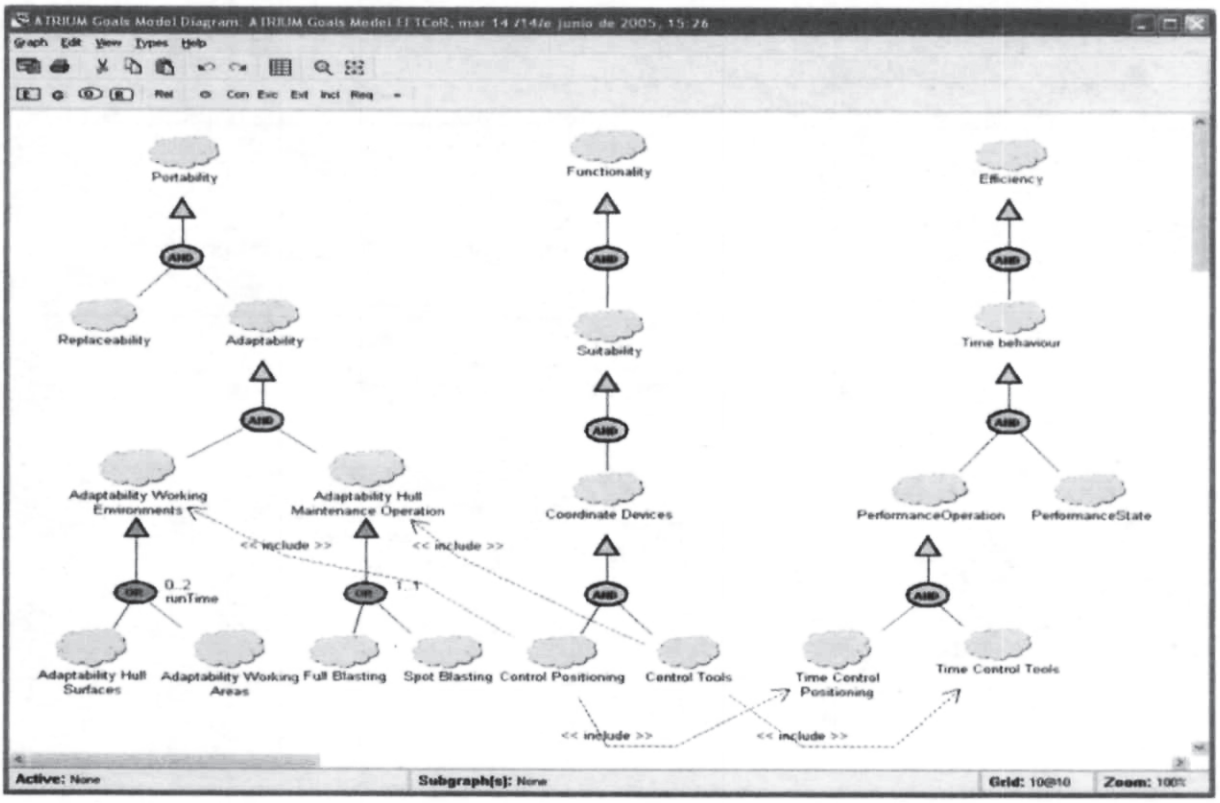
The first step has been to establish the mapping from the metamodel of the ATRIUM Goal Model to a MetaEdit+ metamodel. We have defined *artefact*, *Goal*, *Requirement* and *Operationalization* as **Objects** with their related properties. Afterwards, *AggregationSet*, *Dependency*, and *VariabilitySet* have been defined as **Relationships**. Finally, we have defined a **Graph** by establishing bindings among such Objects and Relationships.

Once we have stored the whole metamodel in the repository, MetaEdit+ Meta-CASE provides us with full CASE tool functionality for specifying ATRIUM Goal Models: MetaEdit+ (Figure 4). In addition, thanks to the report facilities of MetaEdit+ we have defined specific reports for verification and analysis of models. We should point out that this facility allows us to generate XMI files for each model. It facilitates its checking by means of any of the existing tools of OCL (36) by using the OCL constraints described following step (v).

Figure 4 shows an outline of the model for the EFTCoR system. We have started from the quality characteristics of ISO/IEC 9126 standard, using "Portability," "Adaptability," "Functionality," "Suitability," "Efficiency" and "Time Behaviour," according to our case study concerns. The *AggregationSet* refinements labeled with AND indicate that the

satisfaction of an aggregated goal is achieved if every one of its component goals is satisfied. The refinement OR under the goal "Adaptability Working Environment" indicates that it is a variation point, which is defined by means of a refinement *VariabilitySet*. In our example, either the goal "Adaptability Hull Surfaces" or the goal "Adaptability Working Areas" had to be satisfied in order to achieve a system that was self-adaptive to different working environments. Both subgoals were satisfied by means of characteristics added at runtime. The 0..2 multiplicity indicates that the goal "Adaptability Working Environments" is optional.

FIGURE 4
Modeling a Part of the Goal Model of the EFTCoR System by Means of MetaEdit+



The functionality "Coordinate Devices" was refined into two subgoals: "Control Positioning" and "Control Tools." The specification of these subgoals is composed of other non-functional goals obtained from the refinement of "Adaptability" and "Efficiency." The refinement of the goal "Efficiency" establishes time constraints on the control and tools positioning according to those established by "Time Control Positioning" and "Time Control Tools." This composition of specifications was modeled by means of dependencies labeled as *Include*.

Discussion of Results

Several conclusions were achieved after the use of this customized metamodel along with its tool support. First of all, it is important to highlight its ability to cope with all the specific needs of this project. Previously, Use Cases and IEEE 830-1998 were used for specification purposes. After an initial elicitation step, more than one hundred requirements were identified. Most of them were non-functional ones with the difficulties inherent to the crosscutting and the large specification. This crosscutting appears in several contexts, mainly whenever non-functional requirements were constraining functional ones, for instance, the performance goal "Time Control Positioning," which restricts the functional goal "Control Positioning."

In this sense, the proposed model was helpful for the analysts in the project because the metamodeling approach provides them with the ability to suitably define and use these kinds of crosscutting as specific dependencies among artifacts throughout the process development.

In addition, the availability of a tool with graphical capabilities both for metamodeling and modeling also offered a great advantage over other less flexible alternatives. One of the key points for this project was to provide a clear notation where every stakeholder could easily understand the specification, due to the fact that a multidisciplinary team is developing this project integrated by computer science masters, industrial masters, telecommunication masters, etc. For this reason, a graphical notation made the specification more readable and comprehensible to the practitioners of the project.

RELATED WORKS

This section focuses on describing works that include some of the RE approaches described in section 2. It is worthy of note that their purpose is not the integration of approaches but the extension of an existing notation to achieve some specific

expressiveness. In spite of this, none of them provides guidelines about how its notation can be tailored to specific needs nor any detail about how they extended the initial notation.

Within the scope of the Aspect-Oriented approach, the proposals included in (32) and (47) exploit a Goal Model to identify and specify aspects in early stages. Navarro et al (32) have extended the expressiveness of the Goal Model by incorporating a new type of relationship which allows the specification of the crosscutting between goals. Yu et al (47) identify the crosscutting whenever an operationalization contributes to the satisfaction of several goals.

Brito et al (3) present an extension of the Use Case diagram for its application to AORE. They have specified «wrappedBy» and «overlappedBy» relationships by stereotyping a dependency. In this way, if a non-functional requirement implies a constraint on a functional one, this would appear as a Use Case that is stereotyped as a «crosscuttingConcern» and would present a weaving relationship with the other Use Cases. Other proposals (2) find it more appropriate to employ two different models to specify functional and non-functional requirements, using a Use Case model and a Goal Model, respectively. The main problem with these approaches is that specification can hardly be analyzed, due to the fact that they use two separate models.

In (16) and (31) extensions to Use Case diagrams are used to manage the variability. All of them take advantage of the easy communication with the user. However, there are drawbacks whenever a specific expressiveness is required to specify the variability. Basically, these proposals have increased the expressiveness of the UC diagrams by means of the introduction of stereotypes that allow variability concepts to be modeled. Thus, for example, when a Use Case specifies a functionality that some products of the family have, it is stereotyped as «variant». The proposal presented by Halmans and Pohl (16) is perhaps the more extensive because it is the only one to incorporate its own graphical notation to specify the variation points and the multiplicities. However in this case, the formulated extension occasions some drawbacks that are inherent to the Use Case approach, because it does not provide by default any support to analyze or validate the model.

Similarly, the Goal-Oriented Approach has been recently applied to modeling and analyzing variability. Meaningful works in this area are those presented in (14) and (17). When the Goal-Oriented approach is applied, AND/OR/XOR relationships appear as a consequence of the construction process of the model and are recorded in the refinement graph. Thus, these goals that have been refined by means of OR and XOR relationships are capable of representing variation points, in the specification of a product line or a dynamic architecture, in a natural way. However, these proposals have some deficiencies from the point of view of the required expressiveness for variability; for example, they lack the concept of multiplicity or specific dependencies between variants.

We have analyzed some current tools for the management of requirements such as DOORS (9), IRqA (19) and RequisitePro (40). Even though they exhibit a great potential for defining artefacts, they only offer one type of dependency (the traceability specified between artefacts characterized as *from* or *to*) and only one refinement (using the *AggregationSet* semantic).

CONCLUSIONS AND FURTHER WORKS

One of the main challenges for RE is to prove in practice the advantages of the proposed techniques, providing facilities for the integration and adaptation of RE technology to real-life projects. Each project has its specific needs and requires the selection, integration and customization of suitable techniques to define its RE method. In this work, we have presented an approach based on metamodeling to offer such an integration and adaptability.

One important problem when defining highly expressive models (which can handle a wide range of types of artefacts and/or dependencies) is achieving a consensus with regard to their semantics. We have dealt with this problem by using metamodeling, which allows us to adapt and extend a core set of concepts keeping a suitable level of semantic consistence. In order to establish a global set of RE concepts and the required expressiveness, five representative techniques were studied. In this proposal, according to the project's specific needs, a proper integration is provided as well as scalability from simple up to other more sophisticated RE techniques. The main features of this proposal are:

Definition of a metamodel which includes a core of concepts corresponding to the essential expressiveness of some of the more popular and/or advanced approaches in RE.

Establishment of guidelines for adapting the metamodel to specific needs, according to the expressiveness. We have validated both the core and the extension presented of the metamodel by means of their application to the EFTCoR system. Currently, we are concerned with evaluating the stated method by using criteria like those proposed by (44).

This metamodel is the first step toward a tool that supports modeling requirements by integrating expressiveness from different approaches. The demands for scalability and expressiveness, adaptability to specific application domains and the flexible organization of the SRS can only be efficiently satisfied by means of a software tool. The proposed metamodel was implemented as a UML profile in Rational Rose, obtaining a simple tool support for the metamodel and the associated requirements models. However, due to limitations of UML profiles and incomplete CASE tool support for them, this option does not provide the necessary dynamic configuration of the expressiveness and scalability of the metamodel. Therefore, we have overcome these drawbacks using MetaEdit+ Meta-CASE tool which has allowed us to validate our proposal in the EFTCoR project.

We consider that our proposal constitutes a step forward in achieving a successful application of RE techniques in real-life projects. We have obtained a preliminary validation of our proposal through its application in a medium-size project with satisfactory results.

Two topics constitute our future work. The first of them is related to studying other interesting RE approaches in the context of our proposal: *View Points* (13), *Problem Frames* (21) and *Cognitive Mappings* (42, 43). Based on the results of our case study, we think it will be easy to include the additional required expressiveness. Our proposal facilitates a deeper analysis of the specification because proper traceabilities could be established, for instance, between conceptual maps described by means of Cognitive Mappings and services of the system-to-be described using goal models.

The second work is focused on providing a formal framework for analyzing and checking models. It is necessary to specify artefacts and to provide a precise semantic for the expressiveness provided by the model. The works by Letier and Lamsweerde (28) or Katz and Rashid (22) can be useful as a reference to achieve this aim.

REFERENCES

1. Baskerville, R.L. and A.T. Wood-Harper. "A Critical Perspective on Action Research as a Method for Information Systems Research," **Journal of Information Technology**, 11:4, 1996, pp. 235-246.
2. Brito, I. and A. Moreira. "Integrating the NFR Framework in a RE model," Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Lancaster, UK, March 22, 2004.
3. Brito, I. and A. Moreira. "Towards a Composition Process for Aspect-oriented Requirements," Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Boston, MA, March 17, 2003.
4. Bühne, S., G. Halmans, and K. Pohl. "Modeling Dependencies between Variation Points in Use Case Diagrams," 9th International Workshop on Requirements Engineering: Foundation for Software Quality, Klagenfurt/Velden, Austria, June 16-17, 2003.
5. Chung, L., B.A. Nixon, E. Yu, and J. Mylopoulos. **Non-Functional Requirements in Software Engineering.** Boston: Kluwer, 2000, p. 472.
6. Clements, P. and L. Northrop. **Software Product Lines - Practices and Patterns.** Boston: Addison-Wesley, 2001, p. 530.
7. Cockburn, A. **Writing Effective Use Cases.** Boston: Addison Wesley, 2000, p. 304.
8. Dardenne, A., A. van Lamsweerde, and S. Fickas. "Goal-directed Requirements Acquisition," **Science of Computer Programming,** 20:1-2, 1993, pp. 3-50.
9. DOORS. http://www.telelogic.com/products/doorsers/doors/, 2005.
10. EFTCOR: Environmental Friendly and Cost-effective Technology for Coating Removal. European Project, 5th Framework Program (GROWTH G3RD-CT-00794), 2003.
11. Elrad, T., M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. "Discussing Aspects of AOP," **Communications of the ACM,** 44:10: 2001, pp. 33–38.
12. Finkelstein A. and I. Sommerville. "The Viewpoints FAQ," **Software Engineering Journal**, 11:1, 1996, pp. 2-4.
13. Gonzales-Baixauli, B., J.C.S. Prado Leite, and J. Mylopoulos. "Visual Variability Analysis with Goals Models," 12th Int. Conf. on Requirements Engineering, Kyoto, Japan, September 6-10, 2004, pp. 198-207.
14. van Gurp, J., J. Bosch, and M. Svahnberg. "On the Notion of Variability in Software Product Lines," Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 28-31, 2001, pp. 45-54.
15. Halmans, G. and K. Pohl. "Communicating the Variability of a Software Product Family to Customers," **Software and Systems Modeling**. 2:1, 2003, pp. 15-36.
16. Hui, B., S. Liaskos, and J. Mylopoulos. "Requirements Analysis for Customizable Software Goals-Skills-Preferences Framework," 11th IEEE Int. Requirements Engineering Conference, Monterey Bay, California, USA, September 8-12, 2003, pp. 117-126.
17. IEEE Std. 830-1998. IEEE Recommended Practice for Software Requirements Specifications, Volume 4: Resource and Technique Standards, IEEE Software Engineering Standards Collection.
18. IrqA, http://www.irqaonline.com, 2005.
19. ISO/IEC Standard 9126-1 Software Engineering- Product Quality-Part 1: Quality Model, ISO Copyright Office, Geneva, June 2001
20. Jackson, M. **Problem Frames: Analyzing and Structuring Software Development Problems.** Boston: Addison-Wesley, 2000, p. 416.
21. John, I. and D. Muthig. "Tailoring Use Cases for Product Line Modelling," Int. Workshop on Requirements Engineering for Product Lines, Essen, Germany, September 10, 2002.
22. Katz, S. and A. Rashid. "From Aspectual Requirements to Proof Obligations for Aspect-Oriented, Systems," 12th IEEE Int. Requirements Engineering Conference, Kyoto, Japan, September 06-10, 2004, pp. 48-57.
23. Kavakli, E. and P. Loucopoulos. "Goal Modeling in Requirements Engineering: Analysis and Critique of Current Methods," Information Modeling Methods and Methodologies, IDEA Group, Hershey, USA, 2005, pp. 102 – 124.
24. Kelly, S., K. Lyytinen, and M. Rossi. "METAEDIT+ A Fully Configurable Multi-User and Multi-tool CASE and CAME Environment," 8th Conference on Advanced Information Systems Engineering, Heraklion, Creta, May 1996, pp. 1-21.
25. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," 11th European Conference on Object-Oriented Programming, LNCS 1241, Finland Jyväskylä, Finland, June 9-13, 1997, pp. 220-242.
26. van Lamsweerde, A. "Goal-Oriented Requirements Engineering: A Guided Tour," 5th IEEE International Symposium on RE, Toronto, Canada, August 2001, pp. 249-263.
27. Letelier, P., E. Navarro, and V. Anaya. "Customizing Traceability in a Software Development Process," Information Systems Development Advances in Theory, Practice, and Education, Springer Science+Business Media, Inc., 2005, pp. 137-148.
28. Letier, E. and A. van Lamsweerde. "Deriving Operational Software Specifications from System Goals," **ACM SIGSOFT Software Engineering Notes**, 27:6, 2002, pp. 119-128.
29. von der Maßen, T. and H. Lichter. "Modeling Variability by UML Use Case Diagrams," Int. Workshop on RE for Product Lines, Essen, Germany, Sept. 9, 2002.
30. Navarro, E., I. Ramos, and J. Pérez. "Software Requirements for Architectured Systems," 11th IEEE Int. Requirements Engineering Conference, Monterey, September 8-12, 2003, pp. 365-366.
31. Navarro, E., P. Letelier, and I. Ramos. "Goals and Quality Characteristics: Separating Concerns," Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Vancouver, Canada, October 25, 2004.
32. Navarro, E., P. Letelier, and I. Ramos. "UML Visualization for an Aspect and Goal-Oriented Approach," 5th Aspect-Oriented Modeling Workshop, Lisbon, Portugal, October 11, 2004.
33. Nuseibeh, B. and S. Easterbrook. "Requirements Engineering: A Roadmap," 22nd Int. Con. on Software Engineering, Future SE Track, Limerick Ireland, June 4-11, 2000, pp. 37-46.
34. OCL 2.0 Specification, OMG, http://www.omg.org/docs/ptc/05-06-06.pdf, 2005.
35. OCL tools & services http://www.klasse.nl/ocl/ocl-services.html, 2005.
36. Oreizy, P., N. Medvidovic, and R.N. Taylor. "Architecture-

Based Runtime Software Evolution," 20th Int. Conference on Software Engineering, Kyoto, Japan, April 1998, pp. 117-186.
37. Rashid, A., A. Moreira, and J. Araújo. "Modularization and Composition of Aspectual Requirements," 2nd Int. Conf. Aspect-Oriented Software Development, Boston, March 17 -21, 2003, pp. 11-20.
38. Rashid, A., P. Sawer, A. Moreira, and J. Araújo. "Early Aspects: A Model for Aspect-Oriented Requirements Engineering," 10th IEEE Joint International Conference on Requirements Engineering, Essen, Germany, Sept. 9-13, 2002, pp. 199-202.
39. RequisitePro, http://www.306.ibm.com/software/awdtools/reqpro/, 2005.
40. Rational Rose, http://www-306.ibm.com/software/awdtools/architect/swarchitect/, 2005.
41. Siau, K. and L. Lee. "Are Use Case and Class Diagrams Complementary in Requirements Analysis? An Experimental Study on Use Case and Class Diagrams in UML," **Requirements Engineering**, 9:4, 2004, pp. 229-237.
42. Siau, K. and X. Tan. "Evaluation Criteria for Information Systems Development Methodologies," **Communications of the AIS**, 16, 2005, pp. 856-872.
43. Siau, K. and X. Tan. "Improving the Quality of Conceptual Modeling Using Cognitive Mapping Techniques," **Data and Knowledge Engineering**, 55:3, 2005, pp. 343-365.
44. Siau, K. and X. Tan. "Technical Communication in Information Systems Development: The Use of Cognitive Mapping," **IEEE Trans. on Professional Communications**, 48:3, 2005, pp. 269-284.
45. Suzuki, J. and Y. Yamamoto. "Extending UML with Aspects: Aspect Support in the Design Phase," Aspect Oriented Programming Workshop, Lisbon, Portugal, June 1999.
46. Yijun Y., J.C.S.P. Leite, and J. Mylopoulos. "From Goals to Aspects: Discovering Aspects from Requirements Goal Models," 12th IEEE Int. Requirements Engineering Conference, Kyoto, Japan, Sept. 6-10, 2004, pp. 38-47.
47. Yu, E., L. Liu, and Y. Li. "Modeling Strategic Actor Relationships to Support Intellectual Property Management," 20th Int. Conference on Conceptual Modeling, LNCS Vol. 2224, Yokohama, Japan, November 27-30, 2001, pp. 164 - 178.
48. Zave, P. "Classification of Research Efforts in Requirements Engineering," **ACM Computing Surveys**, 29:4, 1997, pp. 315-321.

Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System

MA. EUGENIA CABELLO, CRISTÓBAL COSTA, ISIDRO RAMOS, JOSÉ A. CARSÍ
Department of Information Systems and Computer Science
Polytechnic University of Valencia
Camino de Vera s/n, 46022 Valencia
SPAIN
{mcabello, ccosta, iramos, pcarsi}@dsic.upv.es http://issi.dsic.upv.es

Abstract: This paper presents the construction of a software model (PIM following OMG standards) of an expert system for diagnostic tasks within the PRISMA framework. PRISMA allows the user to obtain a product line for this type of system, offering the advantages that PRISMA architectures provide in the construction of distributed, complex, and re-usable systems. Our architectural model is described in two levels: the type level to define the types of the architectural elements that make up the model, and the configuration level to define the instances and specify the topology of the model. The architectural elements that make up the model in different levels of abstraction (from the simplest to the most complex) have been described, diverse aspects (coordination, functional, distribution, replication, and presentation) have been mentioned, and three reasoning strategies (deductive, inductive, and differential) have been used. Since our architectural model follows the MDA approach the model is transformed automatically to .NET executable code (PSM) by means of the PRISMA Model Compiler.

Key-Words: software architectures, aspects, architectural elements, diagnosis, reusability of software, expert systems

1 Introduction

Systems for diagnosis have recently become of great interest. They identify the internal causes that lead to problems starting from a series of observable data. This need has motivated the development of systems that perform tasks of this type in different domains such as medicine, law, electronics, meteorology and agriculture. At the present time, the importance of software systems has increased along with their complexity and, therefore, there is an increased need to improve them.

In order to improve the development and maintenance of diagnostic systems, a generic architectural model of an expert system has been constructed to perform diagnostic tasks, using the modelling framework PRISMA [1]. This model offers the advantages of distributed systems, facilitating reusability and decreasing complexity.

In the proposed method, once the different architectural elements and aspects that make up the system are identified, the topology of our architectural model is obtained. This model is instantiated, obtaining a specific topology for any kind of system that is desired. In this way, generic software applications for diagnostic systems can be obtained.

This model is automatically compiled into C# code and into its abstract specification in the PRISMA ADL by means of the PRISMA Model Compiler.

The article is divided into five sections. Section 2 introduces the PRISMA model and the architecture of the expert system; the correspondence between the two gives rise to our generic PRISMA architecture for expert systems. Section 3 describes the proposed architectural model. Section 4 presents the related works. Finally, section 5 presents our conclusions and future work.

2 Architectural foundations

The architectural conceptual model presented in this work has been designed within the framework of the PRISMA model, considering the architecture and operation of a rule-based expert system.

2.1 The PRISMA model

The PRISMA model integrates two approaches: the Component-Based Software Development approach (CBSD) [2] and the Aspect-Oriented Software Development approach (AOSD) [3]. This integration is obtained by defining the architectural elements

through their aspects. The PRISMA model not only defines basic architectural elements and specifies their syntax and their semantic, it also specifies the aspects that cover the necessary properties of each element. For this reason, a PRISMA architectural element can be analyzed from two different views: the internal view and the external view (see Figure1). The internal view shows the architectural element as a prism so that each side of the prism corresponds to an aspect. The external view encapsulates the functionality of an architectural element showing only the services that it publishes through its ports.

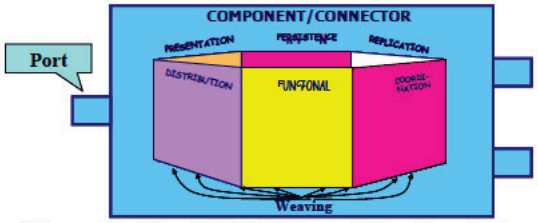


Figure 1. The PRISMA architectural element

The PRISMA model consists of three types of architectural elements: components, connectors, and systems. A component captures the functionality of the system, whereas a connector acts as a coordinator among other architectural elements. A system is an architectural element of great granularity that allows the encapsulation of a set of components, connectors, and other systems. This, in turn, allows the system to be correctly connected to the other architectural elements.

A PRISMA architectural element is formed by a set of aspects of different types, the interweaving relations between these aspects (*weavings*), and one or more ports. The services of the architectural elements are published through their ports, which establish a partial behaviour by means of a certain role (*played_role*). The ports represent the points of interaction among the architectural elements that are interconnected by channels of communication (*attachments and bindings*). The *attachments* establish the communication channel between the port of a component and the port of a connector. The *bindings* establish the communication between the port of a system and the architectural elements that the system encapsulates. This way, the *bindings* link the different levels of granularity of the architectural elements that make up the model.

The aspects carry out the protocols. The protocol defines a process by describing a set of actions whose occurrence is possible. It is made up of a set of processes that establishes the services and transitions to other possible processes. It also specifies the execution priorities of each one of the services involved. A protocol models the global process: it coordinates and synchronizes the services of the different *played_roles*. Therefore, a *played_role* is a projection of the protocol on the set of services of an interface that defines the partial behaviour of a port with a certain role. It is a partial view of the protocol that has a specific and restrictive behaviour that can later be associated to a port of a component or connector. A *played_role* allows the components to behave in a different way depending on what other components they are connected to. The *played_roles* and the protocols are specified using poliadic pi-calculus [5], which describes the process and services that are executed.

PRISMA makes the detection of the architectural elements using the functional decomposition criterion applied to software architectures [4]. This detection has resulted in architectural elements in different abstraction levels.

PRISMA defines the architectures in two levels: the type level and the configuration level. In the type level, the types of the architecture are defined: interfaces, aspects, components, connectors and systems, all of which can be reused. In the configuration level, the types are instantiated and the topology of the model is specified.

2.2 The Expert System Architecture

Expert systems capture the knowledge of experts and try to imitate their reasoning process when they solve problems in a certain domain.

The knowledge base, the inference motor, the working memory or facts base, and the user interface, are the main components of the architecture of a rule-based expert system (see Figure 2).

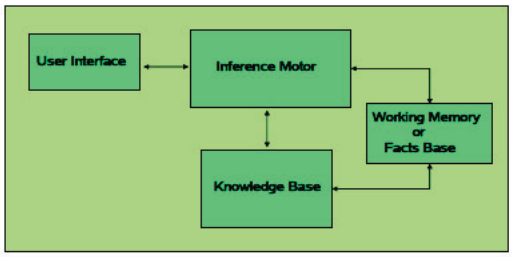


Fig. 2 Expert System architecture

These elements are independent and make up separates units. The data are grouped in the working memory (temporary warehouse of dynamic information). The representation of the knowledge uses rules of the type: IF <premise > THEN <conclusion>, which make up the knowledge base. The control is independent and is performed by the inference motor when making inference processes through reasoning strategies. The input and the output information of the system are made through the user interface.

2.3 The correspondence of the architectural elements between the PRISMA architecture and the Expert System architecture.

The design of the architectural elements of our PRISMA model for Expert Systems are based on the following criteria: the inference motor corresponds to the *Component_type InferenceMotor* of our architecture; the knowledge base and the working memory jointly correspond to the *Component_type KnowledgeBase;* and the user interface corresponds to the *Component_type User*.

Since our architecture is based on the PRISMA model, it was necessary to introduce two connectors (the *Connector_type InferenceKnowledgeConnect* and the *Connector_type UserShellConnect*) to connect the components that make up our model. The first connector *Connector_type InferenceKnowledge Connect* joins the *Component_type InferenceMotor* with the *Component_type KnowledgeBase*. The *Connector_type UserShellConnect*, the *Component_type InferenceMotor,* and the *Component_type KnowledgeBase* are encapsulated in a new architectural element called *System_type Shell*. This new element is connected to the *Component_type User* through the second connector *Connector_type UserShellConnect*.

3. The PRISMA Architectural Model for Expert Systems of Diagnosis

In this section, we present the architectural elements and the aspects in the type level. We also define the instances that make up the topology of the architectural model in the configuration level.

3.1 The definition of the types

In this section, we describe the architectural elements of the proposed model from a simpler functional grain (level 1) to a more complex grain (level 3). We also describe the aspects that are necessary to make up each architectural element.

3.1.1 The architectural elements

The simplest level of abstraction (level 1) consists of the *Component_Type InferenceMotor*, the *Component_Type KnowledgeBase,* and the *Connector_Type InferenceKnowledgeConnect*. The following level of abstraction (level 2) consists of the *System_Type Shell* (which encapsulates the three architectural elements of level 1), the *Component_Type User,* and the *Connector_Type UserShellConnect.* The greatest level of abstraction (level 3) consists of the *Architectural_Model Diagnostic.*

Component_Type InferenceMotor- This component establishes the control of the system and provides the general resolution strategy to obtain the diagnosis. The inference processes are carried out in this component through three reasoning strategies: deductive reasoning (process search led by the input data); inductive reasoning (process search led by the objective); and differential reasoning (process that discriminates between two or more possible solutions to determine which is the most reasonable). These are the strategies that are most widely used by the people when they make a diagnosis. To determine the order in which the diagnosis strategies will be executed, we considered the following process: first, the deductive reasoning is executed to make a diagnosis. If two or more diagnosis possibilities are detected, differential reasoning will be executed. This will invoke inductive reasoning to reformulate questions to the user in order to determine which of the possibilities is the most accurate for the problem under analysis. This architectural element is independent of the domain knowledge of the case study, and therefore, the same *Component_Type InferenceMotor* can be the base of several systems in different domains. This component contains two ports: one port to communicate with the *Component_Type KnowledgeBase* (to acquire the domain data) and the other port to communicate with the *Component_Type User* (to obtain the values of the observable data). The behaviour of the *Component_Type InferenceMotor* integrates two aspects: the functional aspect and the distribution aspect.

Component_Type KnowledgeBase.- This component contains the domain knowledge of the case study by means of rules of inference (Horn clauses with head, which are modelled in PRISMA as derived attributes rules) and facts (Horn clauses without head, which are modelled as attributes). The facts represent the dynamic structure of the knowledge since these facts can be increased by relating the domain rules. This architectural element includes a temporary warehouse of dynamic information. This warehouse stores all the information offered by the user to the system (answers to formulated questions) and also stores the conclusions of the rules of the inference process in the form of facts. When the diagnostic process of a particular problem has concluded, the content of the working memory is eliminated. This component requests and sends services through a single port and defines its behaviour through the functional aspect and the distribution aspect.

Connector_Type InferenceKnowledgeConnect.- This connector synchronizes requests and sends the services requested by the components that are attached through its four ports. This connector has two aspects to implement this functionality: the coordination aspect and the distribution aspect.

System_Type Shell.- This is a component that encapsulates the diagnosis system, which will interact with the environment. This complex component is made up of the aggregation of the *Component_Type InferenceMotor,* the *Component_Type KnowledgeBase,* and the *Connector_Type InferenceKnowledgeConnect.* The set of services offered by this system is provided through its single port. The *System_Type Shell* does not contain emergent aspects.

Component_Type User- This component allows the communication between the user and the system. Through it, the user offers initial data to the system or responds to questions formulated by it. This component supplies a set of communication services through a single port and contains four aspects that define its behaviour: the functional aspect, the presentation aspect, the replication aspect, and the distribution aspect.

Connector_Type UserShellConnect.- This connector joins the *Component_Type User* and the *System_Type Shell* to perform the function of repeater between them; it simply forwards the requested services. It contains two ports to communicate the components that are attached. This connector defines the distribution aspect and the coordination aspect.

3.1.2 The aspects

The different interests (*crosscuting concerns*) of the system are separated into aspects, which increase the system maintenance. Ten aspects were necessary to define each one of the architectural elements of the designed software architecture. Some aspects have been used in more than one architectural element. This fact demonstrates the degree of reusability of the PRISMA approach.

Functional_Aspect FMotor.- The functional aspect of the *Component_Type InferenceMotor* defines the reasoning process of the system. It uses three reasoning strategies (deductive, inductive, and differential). By means of this aspect, the *Component_Type InferenceMotor* acquires data from the *Component_Type KnowledgeBase* to limit the possible options of the decision tree so that the diagnosis can be made.

Functional_Aspect FBase.- The functional aspect of the *Component_Type KnowledgeBase* defines the inference rules of the domain. In this aspect, the domain of the case study is specified. If the case study is changed, it will be necessary to replace this aspect by another *Functional_Aspect* that contains the information of the new domain. Thus, the approach presented in this work could be applied to diverse fields of knowledge by simply changing the functional aspect of the *Component_Type KnowledgeBase.*

Functional_Aspect FUser.- The functional aspect of the *Component_Type User* consists of the request for services that the system offers to the user, as well as the services required by the system to obtain the observable data.

Distribution_Aspect DLocation.- This is the distribution aspect of the *Component_Type InferenceMotor,* the *Component_Type KnowledgeBase,* and the *Connector_Type InferenceKnowledgeConnect.* This aspect indicates the location of the instances of the architectural element.

Distribution_Aspect DLocationMove.- This is the distribution aspect of the *Component_Type User* and the *Connector_Type UserShellConnect*. This aspect provides the information about the location of the component which is being used. Since the system can be managed from different machines, this aspect provides the services of mobility to move from one machine to another.

Replication_Aspect RUser.- The replication aspect of the *Component_Type User* allows the simultaneous access of different users, since there may be more than one user in the system.

Presentation_Aspect PUser.- The presentation aspect of the *Component_Type User* stores the set of elements that define the graphical interface that is used by the user.

Coordination_Aspect CUser.- The coordination aspect of the *Connector_Type UserShellConnect* defines the synchronization between the architectural elements that are joined by this connector.

Coordination_Aspect CDiag.- The coordination aspect of the *Connector_Type InferenceKnowledgeConnect* involves the synchronization of the services sent and received by the architectural elements that this connector coordinates.

The *Connector_Type InferenceKnowledgeConnect* defines the general diagnosis process through its coordination aspect. Figure 3 shows the formal specification using the PRISMA ADL in the type definition level for the protocol of the coordination aspect of the *Connector_Type InferenceKnowledge Connect,* i.e. the choreography.

```
CDIAG = begin COORD

COORD =
   (
   USER_CNCT GetResults ? (Disease, Cert, Treatment) P1
   →
   MOTOR_EXTERNAL_CNCT GetResults ! (Disease, Cert, Treatment)  P2
   →
   MOTOR_INTERNAL_CNCT CleanDB ? ( )
   →
   KNOWLEDGE_CNCT CleanDB ! ( )
   →
   MOTOR_INTERNAL_CNCT AcquireNameThickGrainSignsSymptoms ? (NameTGSS)
   →
   KNOWLEDGE_CNCT AcquireNameThickGrainSignsSymptoms ! (NameTGSS)
   →
   KNOWLEDGE_CNCT AcquireNameThickGrainSignsSymptoms ? (NameTGSS)
   →
   MOTOR_INTERNAL_CNCT AcquireNameThickGrainSignsSymptoms ! (NameTGSS)
   →
   MOTOR_EXTERNAL_CNCT GetValueThickGrainSignsSymptoms ? (ValueTGSS)
   →
   USER_CNCT GetValueThickGrainSignsSymptoms ! (ValueTGSS)
   →
   USER_CNCT GetValueThickGrainSignsSymptoms ? (ValueTGSS)
   →
   MOTOR_EXTERNAL_CNCT GetValueThickGrainSignsSymptoms! (ValueTGSS)
   →
   MOTOR_INTERNAL_CNCT InferThickDiagnosis ? (NameTGSS, ValueTGSS,
   ThickDiagnosis)
   →
   KNOWLEDGE_CNCT  InferThickDiagnosis! (NameTGSS, ValueTGSS, ThickDiagnosis)
   →
   KNOWLEDGE_CNCT  InferThickDiagnosis? (NameTGSS, ValueTGSS, ThickDiagnosis)
   →
   MOTOR_INTERNAL_CNCT  InferThickDiagnosis! (NameTGSS, ValueTGSS,
   ThickDiagnosis)
   →
   MOTOR_INTERNAL_CNCT InferPossibleDiagnosis ? (ThickDiagnosis,
   PossibleDiagnosis)
   →
   KNOWLEDGE_CNCT InferPossibleDiagnosis ! (ThickDiagnosis, PossibleDiagnosis)
   →
   KNOWLEDGE_CNCT InferPossibleDiagnosis ? (ThickDiagnosis, PossibleDiagnosis)
   →
   MOTOR_INTERNAL_CNCT InferPossibleDisease ! (ThickDiagnosis, PossibleDiagnosis)
   →
   MOTOR_INTERNAL_CNCT AcquireNameFineGrainSignsSymptoms ?
   (PossibleDiagnosis, NameFGSS)
   →
   KNOWLEDGE_CNCT AcquireNameFineGrainSignsSymptoms ! (PossibleDiagnosis,
   NameFGSS)
   →
   KNOWLEDGE_CNCT AcquireNameFineGrainSignsSymptoms ? (PossibleDiagnosis,
   NameFGSS)
   →
   MOTOR_INTERNAL_CNCT AcquireNameFineGrainSignsSymptoms !
   (PossibleDiagnosis, NameFGSS)
   →
   MOTOR_EXTERNAL_CNCT GetValueFineGrainSignsSymptoms ? (ValueFGSS)
   →
   USER_CNCT GetValueFineGrainSignsSymptoms ! (ValueFGSS)
   →
   USER_CNCT GetValueFineGrainSignsSymptoms ? (ValueFGSS)
   →
   MOTOR_EXTERNAL_CNCT GetValueFineGrainSignsSymptoms ! (ValueFGSS)
   →
   MOTOR_INTERNAL_CNCT InferFineDiagnosis ? (NameFGSS, ValueFGSS, Diagnosis,
   Certainty)
   →
   KNOWLEDGE_CNCT InferFineDiagnosis ! (NameFGSS, ValueFGSS, Diagnosis,
   Certainty)
   →
   KNOWLEDGE_CNCT InferFineDiagnosis ? (NameFGSS, ValueFGSS, Diagnosis,
   Certainty)
   →
   MOTOR_INTERNAL_CNCT InferFineDiagnosis ! (NameFGSS, ValueFGSS, Diagnosis,
   Certainty)
   →
   MOTOR_INTERNAL_CNCT InferRecomendation ? (Diagnosis, Certainty,
   Recomendation)
   →
   KNOWLEDGE_CNCT InferRecomendation ! (Diagnosis, Certainty, Recomendation)
   →
   KNOWLEDGE_CNCT InferRecomendation ? (Diagnosis, Certainty, Recomendation)
   →
   MOTOR_INTERNAL_CNCT InferRecomendation ! (Diagnosis, Certainty,
   Recomendation)
   →
   MOTOR_EXTERNAL_CNCT GetResults ? (Diagnosis, Certainty, Recomendation)
   →
   USER_CNCT GetResults ! (Diagnosis, Certainty, Recomendation) ) COORD
+
   end END
```

Fig. 3 Coordination aspect protocol of the Connector_Type InferenceKnowledgeConnect

Figure 4 shows the transition graph that corresponds to the specification of the protocol in Figure 3.
Each arrow in Figure 4 corresponds to a service of the protocol in Figure 3. The arrow indicates the transition of each step in the process of the protocol. Some examples are shown below:
begin service corresponds to the transition from CDIAG to COORD,
GetResults service corresponds to the transition from COORD to P1,
CleanDB service corresponds to the transition from P2 to P3.

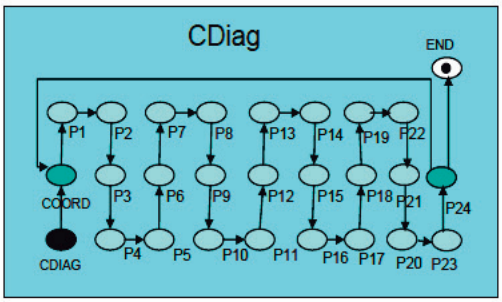


Fig. 4 Transition graph of the coordination aspect protocol of the Connector_Type InferenceKnowledgeConnect

To illustrate our work, we present a part of the conceptual model through three views of the *Connector_Type InferenceKnowledgeConnect*. Figure 5 shows the graphical metaphor of this architectural element (with its aspects and interfaces) using Microsoft DSL Tools [6]. Figure 6 shows the specification in the PRISMA ADL of this connector, and Figure 7 shows their corresponding C# code. Both were automatically generated through the PRISMA Model Compiler.

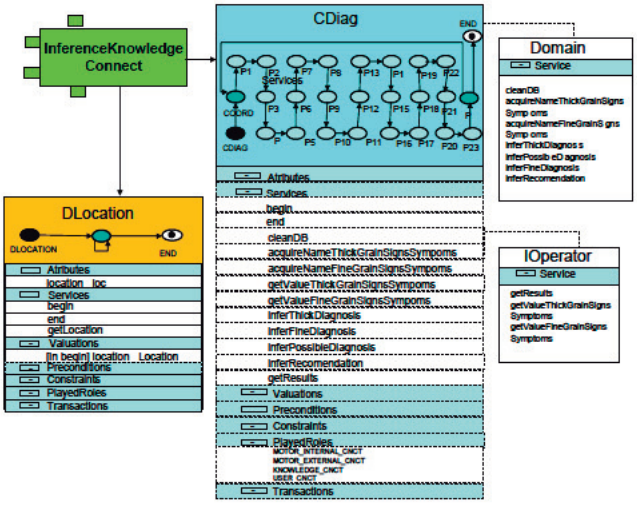


Fig. 5 Graphical metaphor of the Connector_Type InferenceKnowledgeConnect

```
Connector_Type InferenceKnowledgeConnect
   Ports
        MotorInternalPortCnct: IDomain,
             Played_Role  CDiag MOTOR_INTERNAL_CNCT;
        MotorExternalPortCnct: IOperator,
             Played_Role  CDiag MOTOR_EXTERNAL_CNCT;
        KnowledgePortCnct: IDomain,
             Played_Role  CDiag KNOWLEDGE_CNCT;
        UserPortCnct: I<operator,
             Played_Role  CDiag USER_CNCT;
   End_Ports;

   Coordination Aspect import  CDiag;
   Distribution Aspect import  DLocation;

   Initialize
         new ( )
         {    CDiag begin ();
              DLocation begin (Location);
                   }
   End_Initialize;

   Destruction
         destroy ()
         {    CDiag end ();
              DLocation end ();
                   }
   End_Destruction;
End_Component_Type   InferenceKnowledgeConnect;
```

Fig. 6 Specification (ADL PRISMA) of the Connector_Type InferenceKnowledgeConnect

```
namespaceIKCNCT
{
   [Serializable]
   public class InferenceKnowledgeConnect: ConnectorBase
   {
      public class InferenceKnowledgeConnect (string name, loc Location)
: base (name)
      {
         AddAspect (new CDiag ( ) );
         AddAspect ( new DLocation (Location)  );

         InPorts Add ("MotorInternalPortCnct", "IDomain",
                      "MOTOR_INTERNAL_CNCT");
         OutPorts Add ("MotorInternalPortcCnct", "IDomain",
                       "MOTOR_INTERNAL_CNCT");
         InPorts Add ("MotorExternalPortCnct", "IOperator",
                      "MOTORIEXTERNALCNCT");
         OutPorts Add ("MotorExternalPortcCnct", "IOperator",
                       "MOTOR_EXTERNAL_CNCT");
         InPorts Add ("KnowledgePortCnct", "IDomain",
                       "KNOWLEDGE_CNCT");
         OutPorts Add ("KnowledgePortCnct", "IDomain",
                        "KNOWLEDGE_CNCT");
         InPorts Add ("UserPortCnct", "IOperator", "USER_CNCT");
         OutPorts Add ("UserPortCnct", "IOperator", "USER_CNCT");
      }
   }
}
```

Fig. 7 Code (C#) of the Connector_Type InferenceKnowledgeConnect

3.2 Instantiation and configuration of the model

Once the types of the architectural elements and aspects have been defined, the following step is to define the topology. The instances in the topology of our model are shown in Figure 8 and their correspondence to the architectural elements types are shown in Table 1.

Architectural element type	Instance
Component_Type InferenceMotor	Control
Component_Type KnowledgeBase	Domain
Connector_Type InferenceKnowledgeConnect	DiagCnct
System_Type Shell	DiagnosticSys
Component_Type User	Operator
Connector_Type UserShellConnect	OperatorCnct

Table 1 Instantiation of the architectural elements

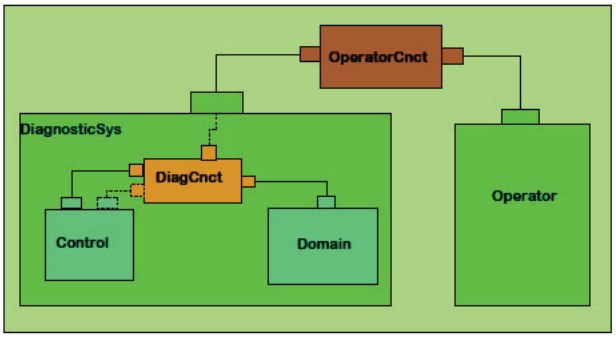


Fig. 8 The PRISMA architecture of the Expert System

Diagnostic System Architectural Model

The syntax of the specification of our architectural model is shown in Figure 9. This specification defines the PRISMA ADL in the configuration level.

```
Architectural_Model  Diagnostic
  Variables
     VarOperator: User;
     VarOperatorCnct: UserShellConnect;
     VarDiagnosticSys: Shell;
  End_Variables;
 Attachments
  VarDiagnosticSys DiagnosticalPort                              ↔
       VarOperatorCnct DiagnosticalPortCnct;
  VarOperatorCnct UserPortCnct                                   ↔
       VarOperator UserPort;
 End_Attachments;
  Initialize
   new (Location:loc, LocMin:loc, LocMax:loc, NewLocation:loc)
   {
   VarOperator = new User (Location,LocMin,LocMax,NewLocation);
  VarOperatorCnct=new UserShellConnect (Location,LocMin,LocMax);
  VarDiagnosticSys = new Shell (Location);
   }
  End_Initialize;
  Destruction
  destroy ( )
  {
   VarOperator destroy ( );
   VarOperatorCnct destroy ( );
   VarDiagnosticSys destroy ( );
  }
  End_Destruction;
End_Architectural_Model Diagnostic
```

Fig. 9 Specification of the architectural model

This specification allows the definition of the architectural model as a type. When the model is instantiated, the required initial values are given. To reuse the architectural model, a specific topology for any system that is to be executed can be obtained by giving different initial values. An example of such an instantiation is shown below:

```
DIAG = new Diagnostic (localhost, 0, 100, newLocalhost ),
```

where DIAG is the instance name of the *Architectural_Model Diagnostic*, and *localhost*, 0, 100, and *newLocalhost* are the instances of the arguments *Location*, *LocMin, LocMax,* and *NewLocation,* respectively. These values correspond to the following: the location of the machine that contains the component, the range of address mobility of the component, and the new location where the component can be replicated.

4 State of the art

There are a great variety of works related to the development of expert systems. The methodologies and applications for these systems have produced a wide variety of research products, offering suggestions and solutions to these systems in specific domains.

A study made by Liao [7] examines the methodologies of expert systems and classifies them into eleven categories. Two of these categories correspond to the systems based on rules and the systems based on knowledge. These categories have been taken into account in our work when using knowledge represented in the form of rules (Horn clauses) and facts (observable variables).

Liao also mentions that the development of expert systems has been characterized by the separation of knowledge and procedure as independent units. In our architectural model, the elements in the type level are defined taking into account this concept, specifically when there is a component that contains the domain knowledge and another component that executes the inference process of the diagnosis.

Giarratano and Riley [8] and other authors in the field of expert systems state that the architectures of these systems are based only on components. The architecture of our system has integrated two approaches combining both components and aspects. This increases the reusability and the maintenance of the system.

Expert systems have also been implemented in the development of different programming paradigms such as structured, logic, and object-oriented paradigms. These paradigms are oriented toward fourth-generation languages and visual programming methods to provide user-friendly communication. PRISMA provides an Architecture Description Language to define an architectural model that follows the MDA approach [9], which allows the automatic generation of code.

5 Conclusions and future work

This work has presented a component-based and aspect-oriented generic architectural model of an expert system for diagnostic tasks. The architectural elements that make up the model have been described in three levels of abstraction (from the simplest to the most complex). The diverse aspects (coordination, functional, distribution, replication, and presentation) have been detected. Three reasoning strategies (deductive, inductive, and differential) have been used.

It is important to note that the PRISMA model has been applied to other domains, such as tele-operation systems for the cleaning of ships by means of an arm-robot [10], as well as in systems for handling bank accounts [11]. These cases have verified the improvement of the development and maintenance of these systems through the combined approach of aspects and components.

This system architecture conserves the characteristics and the performance similar to the other diagnostic expert systems. Nevertheless, the approach presented in this work offers the advantages that PRISMA architectures provide in the construction of distributed, complex and re-usable systems. These advantages are described below:

- The complexity of the system is managed by means of different levels of granularity through the separation of functional units in architectural elements
- The architectural elements of this model are independent so that components and/or connectors, can be added or interchanged without affecting the general operation of the system or the rest of the elements.
- The *crosscuting concerns*, which are separated in aspects, help the maintenance of the system.
- The components and the aspects can be reused in the same architecture in different domains or in different architectures.
- The architecture can be used in both centralized and distributed systems. The distribution aspect specifies the characteristics that define the dynamic location of the architectural element, as well as the mobility services to implement strategies of distribution of the architectural elements.
- The architectural model is independent of the platform and is designed from the perspective of

the problem and not from the solution. This consideration allows diagnostic tasks to be applied to different domains.

To obtain the validation of the proposed model using the PRISMA Model Compiler, we have automatically generated its specification (in the PRISMA ADL) and its code (in C#), which allows this model to be executed on the middleware PRISMANET [12].

As future work, we plan to add several Knowledge-Bases (distributed knowledge) to the model. We are also studying the incorporation of a decision aspect, and other reasoning strategies for the diagnostic process. Finally, we plan to use the system in other types of similar tasks, such as interpretation and prediction.

References:

[1] J. Pérez, N. Ali, J.A.Carsí and I. Ramos, "Dynamic Evolution in Aspect-Oriented Architectural Models", *Second European Workshop on Software Architecture*, in Lecture Notes in Computer Science, Springer Verlag, Vol. 3527, Pisa, Italy, 2005.

[2] C. Szyperski, *Component software: beyond object-oriented programming*, ACM Press and Addison Wesley, New York, USA, 1998.

[3] Aspect-Oriented Software Development, in: http://aosd.net.

[4] F. Bachman, L. Bass, G. Chastek, P. Donohoe and F. Peruzzi, "The Architecture-Based Design Method", *Technical Report CMU/SEI-2000-TR-001*, Carnegie Mellon University, Pittsburgh, USA, 2000, 61 p.

[5] R. Miller, *Π-Cálculo Piliádico*, Tutorial, 1991.

[6] R. Cabedo, J. Pérez, J.A. Carsí e I. Ramos, "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", *IV Workshop DYNAMICA*, Archena, Murcia, España, 2005, pp. 79-86.

[7] S.H. Liao, "Expert Systems Methodologies and Applications- a Decade Review from 1995-2004", in *Expert Systems with Applications*, Vol. 28, Issue 1, 2005, pp. 93-103.

[8] J. Giarratano and G. Riley, *Expert Systems: design and development*, Course Technology, 4a. ed., 2004.

[9] Model Driven Architecture, in: http://www.omg.org/mda.

[10] R. Cabedo, J. Pérez e I. Ramos, "Aplicación del lenguaje de descripción de arquitecturas PRISMA a un sistema robótico industrial", *Informe técnico DSIC-II/12/05*, Universidad Politécnica de Valencia, Valencia, España, 2005, 201 p.

[11] J. Pérez, N. Ali, J.A.Carsí, I. Ramos and E. Navarro, "Designing Software Architectures with an Aspect-Oriented Language", in *Advice: Journal of Aspect Orientation,* Vol. 1, No. 1, 2004, pp. 1-18.

[12] C. Costa, J. Pérez, N. Ali, J.A.Carsí e I. Ramos, "PRISMANET Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", *X Jornadas de Ingeniería del Software y Bases de Datos*, Granada, España, 2005, pp. 27-34.

Acknowledgments. This is funded by the Spanish Department of Science & Technology, under the National Program for Research, Development and Innovation (TIN2006-15175-C05-01), and by the European Regional Development Fund.

Designing Software Architectures with an Aspect-Oriented Architecture Description Language

Jennifer Pérez, Nour Ali, Jose A. Carsí, and Isidro Ramos

Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n
E-46022 Valencia, Spain
{jeperez, nourali, pcarsi, iramos}@dsic.upv.es

Abstract. A great deal of languages have emerged and have demonstrated the advantages that Aspect-Oriented Programming offers. For this reason, the aspect-oriented approach is being introduced into the early phases (analysis and design) of the software life cycle. In this work, we present an Aspect-Oriented Architecture Description Language (AOADL) to specify software architectures of complex, dynamic and distributed software systems. This AOADL follows the PRISMA approach, which integrates the advantages of Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD). The PRISMA AOADL combines components and aspects in an elegant and novel way achieving a better management of crosscutting-concerns. In addition, it is independent of the technology, and it has great expressive power in order to facilitate the automatic code generation from its specifications. In this work, we demonstrate how PRISMA AOADL improves the management, maintainability and reusability of software architectures introducing the notion of aspect in its ADL.

1 Introduction

Nowadays, software systems are becoming more and more difficult to develop due to their complex structures, non-functional requirements and distributed and dynamic nature. Two approaches of software development have emerged to overcome these needs: Component-Based Software Development (CBSD) [7, [27] and Aspect-Oriented Software Development (AOSD) [2].

On the one hand, CBSD reduces the complexity of software development and improves its maintenance by increasing software reuse. CBSD decomposes the system into reusable entities called components. By extension, this advantage is provided by software architectures [8] due to the fact that architectural models are constructed using components. As a result, software architectures have emerged as a solution for the development process of complex software systems.

On the other hand, AOSD allows the separation of concerns by modularizing crosscutting concerns into a separate entity, the aspect. The encapsulation of the aspect permits the reusability of the same aspect in different objects, and the evolution of an aspect without affecting the other objects and aspects. The main emphasis in this approach has been made at the implementation level, by introducing the Aspect

I. Crnkovic et al. (Eds.): CBSE 2006, LNCS 4063, pp. 209 – 224, 2006.
© Springer-Verlag Berlin Heidelberg 2006
Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. Proc. of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006). 2006.

Oriented Programming (AOP) as a new paradigm of software development. A great number of aspect-oriented programming languages have been proposed and have demonstrated that AOP improves the structure and reusability of the code [10].

PRISMA is an approach to develop complex software systems that has been designed taking into account different solutions of interest: CBSD and AOSD. This approach provides a model and an Aspect-Oriented Architecture Description Language (AOADL). The model defines software architectures by integrating aspect-oriented software development (AOSD) and component-based software development (CBSD). This integration is directly reflected in its AOADL. In this work, we specially focus on the demonstration of the improvement of the reusability, the development, and the maintainability of architectures using the PRISMA AOADL.

PRISMA AOADL defines the semantics of the architectural models in a formal way in order to validate and verify PRISMA architectural models and to automatically generate source code from the PRISMA AOADL. It is important to keep in mind that the PRISMA language is independent of the technology and it has great expressive power. These properties allow us to generate code from its specifications and to choose among different technologies at the time of generating the code. For this reason, we can compile the same PRISMA architectural model into different programming languages and technologies, thereby reducing the development time and preserving the traceability between an architectural model and its application code.

The structure of the paper is the following: Section 2 presents a brief summary of related work. Section 3 gives an overview of the PRISMA approach and presents in detail the PRISMA AOADL by demonstrating its main advantages. Conclusions and further work are presented in the last section.

2 Related Work

A wide variety of models based on the separation of concerns have been proposed [21], [28], etc. However, the most widely used is the Aspect-Oriented Paradigm (AOP). Aspect-Oriented models can be classified into two different categories: static models and dynamic models. The static models are not able to change aspects and their weaving at run-time, whereas the dynamic ones offer this advantage. Examples of dynamic models are the Mask Model [22] and the Dynamic Aspect-Oriented Platform (DAOP) [18].

A well-accepted aspect-oriented programming language is AspectJ [10], which is an extension of Java, where the code is separated into aspect and non-aspect code (objects). However, based on experiments implementing different aspects such as persistence and distribution [25], this model has many drawbacks (static, limited reusability). Therefore, in [19], work has been done to provide dynamic weaving using the Java Virtual Machine Debugger Interface (JVMDI).

AOP is being transferred to other platforms such as .NET by means of extensions. However, the existing .NET approaches for supporting AOP are still in an early phase and only Rapier-Loom.Net [23] supports mechanisms for adding or removing aspects dynamically. However, it defines the weavings inside the aspects thereby losing their

reusability. EOS [20] is another dynamic approach that is able to attach aspects at the instance level by means of events. However, none of these approaches takes into account the emerging relations that result from the aggregation of various aspects at the same point of the base code (joinpoint). The JAsCo [26] approach is presented as a solution of this lack. It provides an expressive language that permits the definition of relations among aspects. JAsCo integrates AOSD and CBSD at the implementation level by extending the JavaBeans and introducing connectors to perform dynamic aspect weaving by preserving the aspect reuse. The inconvenience of this approach is that the dynamic weaving of aspects to the base code is referential but not inclusive. As a result, base code and the aspects are not inside the same entity when they are instantiated and the code mobility is limited. As mobility is an essential feature of software components, PRISMA AOADL provides all these needed properties at the same time. These properties are the following: dynamic weaving (run-time evolution), the join of the base code and the aspects inside the same entity (mobility support), and the reuse of aspects (reusability). Finally, an important difference between these AOP approaches and PRISMA AOADL is the technological independence; all these properties are going to be integrated at the architectural level of the software development process, instead of integrating them at the implementation level.

A wide variety of ADLs have been proposed at the architectural level, an interesting comparison between these languages has been made in the work by [11]. Some proposals for the integration of the software architecture and the AOSD have emerged in order to take advantage of both approaches [5]. Each one introduces aspects in their Architecture Description Languages (ADLs) in a different way: as a component [13], as a connector among components [18], as a view of the architecture [9], etc. However, PRISMA introduces the aspect in its ADL as a new concept without simulating it with any other architectural term (components, connectors, views, etc).

Our approach does not only introduce aspects as new requirements [13], it takes advantage of the notion of aspect from the beginning of the system definition. Also, the complete view of the software architecture is not lost by the use of aspects as [9]. Moreover, at the configuration level, when the architectural elements are instantiated, we do not lose the structural and architectural view of the system due to the fact that our components are connected by means of connectors instead of aspects [18]. As Shaw presents in her work [24], the specification of software systems with complex coordination protocols is too difficult without the connector architectural element. This is because the connector provides the separation of the component interaction achieving a higher level of abstraction, modularity and architectural view of the system. TranSAT [3] is another approach that incorporates aspects that refine the original component specification and generates another one that includes them without losing the black box view of the component. However, this approach is only focused on technical aspects.

In [4], the main requirements for developing aspect-oriented software architectures are presented. They are the linguistic support for weaving aspects, dynamic adaptability, and reusability. PRISMA AOADL is presented as a good solution to specify aspect-oriented software architectures because it satisfies these requirements.

3 PRISMA: An Aspect-Oriented Architecture Description Language (AOADL)

The PRISMA approach allows the definition of software architectures of complex software systems by integrating AOSD and CBSD. PRISMA uses AOSD to separate the crosscutting concerns (distribution, security, context-aware, coordination, etc.) of the architecture in aspects. In this way, the PRISMA architectural elements are defined by using aspects to define their behaviour.

A PRISMA architectural element can be analyzed from two different views, internal and external. The internal view (see Figure 1) shows an architectural element as a prism with each side of the prism being an aspect that is imported by this architectural element. This represents that an architectural element of PRISMA is formed by a set of aspects and weavings relationships among aspects. Whereas, the external view (see Figure 2) is an architectural element that encapsulates its functionality as a black box, and by means of its ports publishes and receives a set of services to and from the rest of the architectural elements.

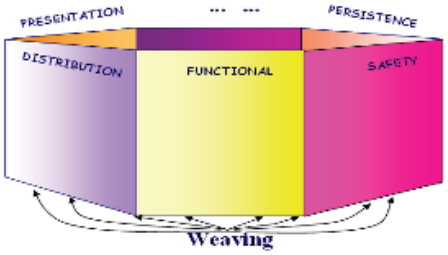


Fig. 1. Internal View of an Architectural Element

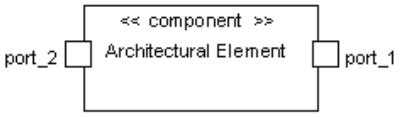


Fig. 2. External View of an Architectural Element

The PRISMA AOADL defines the architectural elements at different levels of abstraction: the type definition level and the configuration level. The type definition level defines architectural types with a high abstraction level. Its main advantages are software reuse and complexity reduction by integrating components and aspects. The PRISMA types defined at this level are stored in a PRISMA repository so that they can be reused by other types or specific architectures.

The configuration level designs the architecture of software systems by creating and interconnecting instances of the defined architectural elements in the type definition level. In other words, we specify the topology of a specific software system at this level.

In the following sections, a very simple banking system example is going to be used to illustrate the PRISMA AOADL. An information system that contains a client-server architecture between two components is defined. By this simple banking system, we present the specification of different aspects (coordination, distribution and functional aspects) and architectural types.

There are two components that are involved in the banking system and are connected with each other: an *ATM* and an *Account*. The ATM is the client component that requires *withdrawal* and *balance* services from the *Account*. The *Account* is the server component that offers the required services to the *ATM*. The

ATM stores an ATM number that identifies it (*numberId)*, the amount of money available in the ATM (*money*) and where the ATM is located (*address).* The *Account* stores an account number that identifies it (*numberId),* the amount of money (*money)*, and the postal address of its owner (*address)*. The system has a connector *ATMAccount* that controls the communication process between the two components. In order to simplify the example we do not take the persistence aspect into account and we assume that the information is stored in the main memory.

3.1 The Type Definition Level

The type definition level of PRISMA defines architectural patterns (complex components) and the first-class citizens of the language: interfaces, aspects, components, and connectors. They are stored in the PRISMA repository in order to be reused.

3.1.1 Interfaces

An *interface* publishes a set of services. It describes the signature of the services that can be invoked or requested through that interface. The arguments that define a service can be *input* or *output*. *Input* arguments are necessary to perform the execution of a service, and *output* arguments will store the result of the service execution. Interfaces publish services without taking into account the ports and aspects that are going to use them. In the following, we present some interfaces of the banking system.

```
Interface ICreditCardTransactions
   withdrawal(input Quantity: currency, output MyMoney: currency);
   balance(output MoneyBalance:currency);
   changeAddress(input NewAdd: string);
End_Interface ICreditCardTransactions;


Interface IMobility
   move(input NewLoc:loc);
End_Interface IMobility;
```

3.1.2 Aspects

An *aspect* defines the structure and the behaviour of a specific *concern* of the software system. Examples of concerns are functionality, coordination, safety, distribution, among others. A common syntax of aspects has been defined. This common syntax is going to be presented using the functional aspect.

- **Functional Aspect**

The head of an aspect specifies its name and the kind of concern it defines: *functional, distribution, coordination,* etc. Moreover, interfaces whose semantics is defined by the aspect are detailed next to the reserved word *using*. We are going to define a functional aspect which specifies the semantics of the *ICreditCardTransaction* interface (see Figure 3, nº 1).

Attributes are specified inside an aspect. These attributes are necessary to store information about the characteristics of the aspect. Attributes are preceded by the *Attributes* reserved word and they have a name and a type (see Figure 3, nº 2). This type defines the kind of values that the attribute can store. There are three kinds of attributes:

- *Constant:* Their stored values cannot change
- V*ariable:* The stored values can be modified
- *Derived:* The value is calculated on demand applying its derivation rule.

An aspect defines the semantics of services. The set of services specified in an aspect must contain the *begin* service, the *end* service, and the interface services that this aspect uses (see Figure 3, nº 3). *Begin* and *end* services do not mean that it is possible to instantiate an aspect by itself; they make a reference to the creation and destruction services of the architectural element that the aspect will belong to, and where it will be instantiated. The semantics of services is defined by means of preconditions and valuations. *Preconditions* establish the condition that must be satisfied to execute a specific service (see Figure 3, nº 5). *Valuations* specify the changes in the value of attributes and parameters by the execution of services (see Figure 3, nº 4).

With regard to services, it is important to take into account that the same service can have two different behaviors: client and server. The client behavior is when a service is invoked by an aspect. The server behavior is when a service is offered and processed by an aspect. Sometimes, it is necessary to distinguish between these two behaviors. The syntactical difference between client or server service is the reserved word *in, out* and *in/out.*

The aspect example shown in Figure 3 has a precondition to indicate when the service *withdrawal* can be executed and the valuation of the *withdrawal* indicates how the *money* attribute is updated when this service is executed. The precondition ensures that there is enough money to be able to withdraw the required quantity, and the valuation of the withdrawal updates the quantity of the available money. Services, preconditions and valuations are preceded by the reserved words *Services, Preconditions*, and *Valuations,* respectively.

It is necessary to specify the protocol to describe the order and the state in which a service could be executed (see Figure 3, nº7). The protocol is a textual specification of a transition state machine.

An aspect also defines the set of roles that can be played taking into account the semantics of the services. They are called *played_roles* (see Fig 3, nº6). A played_role is a projection of the protocol that defines the partial behaviour belonging to a specific role. This role must be compliant with the signature and the process of the protocol. As a result, the played_role is performed inside the global behaviour of the protocol, and its calculations are a subset of the calculations of the protocol.

Every service that composes a played_role belongs to the same interface. This is specified at the beginning of the played_role specification with the *for* keyword and the name of the interface. A played_role is a partial view of the protocol that has its own meaning, a specific behaviour that can be later associated to a port. This association allows us to define the behaviour of ports.The formal language that we

used to describe the *Played_Roles* and *Protocols* is the poliadic π-calculus [12]. The main advantage of this language is the fact that it allows us to describe processes and services that can be executed concurrently in a simple way.

```
1 Functional Aspect BankInteraction using
   ICreditCardTransaction
2   Attributes
        numberId: number;
        name: string;
        address: string
        money: currency;
3   Services
        begin;
        in/out withdrawal(input Quantity: currency, output
                          MyMoney:currency);
4               Valuations
                     [in withdrawal(Quantity, MyMoney)]
                       money := money - Quantity;
                        MyMoney := Quantity;
        in/out balance(output MoneyBalance:currency);
4                Valuations
                     [in balance(MoneyBalance)]
                       MoneyBalance := money;

        in/out changeAddress(input NewAdd: string);
4                Valuations
                    [in changeAddress(NewAdd)]
                      address := NewAdd;
        end;
5   Preconditions
        in withdrawal(Quantity)
            if Quantity <= money;
6   Played_Roles
       BANK for ICreditCardTransaction ::=
                  (withdrawal ?(Quantity, MyMoney)
                   → withdrawal !(Quantity, MyMoney))
                   +
                   (balance ?(Quantity, MyMoney)
                    → balance !(Quantity, MyMoney))
                   +
                   (changeAdress ?(Newadd)
                   → changeAdress !(Newadd));
       CUSTOMER for ICreditCardTransaction:: =
                  ( withdrawal !(Quantity, MyMoney)
                   → withdrawal ?(Quantity, MyMoney))
                   +
                   (balance !(Quantity, MyMoney)
                    → balance ?(Quantity, MyMoney)
                   +
                   (changeAdress !(Newadd)
                   → changeAdress ?(Newadd));
7   Protocol
        BANKINTERACTION ::= begin → TRANSACTION;
       TRANSACTION = (BANK || CUSTOMER) → TRANSACTION + end;
End_Aspect BankInteraction;
```

Fig. 3. *BankInteracation* Functional Aspect

- **Distribution Aspect**

This aspect has the same sections of a functional aspect with some predefined attributes and services. The distribution aspect [1] specifies the locations of the instances. However, if the location is specified in the distribution aspect, the same distribution aspect cannot be reused in different architectural elements if instances are distributed in different locations. This problem is solved using the PRISMA AOADL due to the fact that it is separated into the Type Definition Language and the Configuration Language. Thus, the Type Definition Language specifies that a type has a location without assigning it a value. The value of the location is assigned when architectural elements are instantiated in the Configuration Language.

```
Distribution Aspect ExtMbile using IMobility
   Attributes
      id: nat;
      location: LOC NOT NULL;
   Services
     in move(input NewLoc:LOC);
       Valuations
          [in move(NewLoc)] location:= NewLoc;
  Played_Roles
     MOVEMYSELF::= IMobility.move ? (NewLoc);
  Protocols
         EXTMBILE ≡ begin → MOVEMENT;
         MOVEMENT ≡ MOVEMYSELF  → MOVEMENT +
                    end;
End_Distribution Aspect ExtMbile;
```

Fig. 4. *ExtMbile* distribution aspect

The distribution aspect presented in Figure 4 specifies the behaviour of a mobile architectural element enabling the change of the location attribute by the *move* service *valuation*. The attribute *location* has an abstract data type called *LOC*. This data type hides the different mechanisms of locations of an architectural element at a physical level, e.g. it can be a URL, an IP, etc. The *location* attribute should have a value when an architectural element is instantiated; this is indicated by the *NOT NULL*.

- **Coordination Aspect**

This aspect presents the same sections as a functional one. The difference between both aspects is their purpose, functionality is for computation and coordination is for synchronization of architectural elements. Figure 5, shows the coordination aspect which allows the synchronization of two architectural elements whose port type is the *ICreditCardTransaction* interface.

In Figure 5 the *BankCoordination* coordination aspect does not have attributes because it does not perform computations, it only synchronizes. However, a coordination aspect can have attributes to take coordination decisions in complex protocols. Figure 5 also shows that the coordination played roles are the opposite processes of the functional played_roles (see Figure 3). This is due to the fact that the coordinator units have the opposite process view of the computational units, i.e., an output action for a computation unit is an input action for a coordinator unit and vice versa.

```
Coordination Aspect BankCoordination using ICreditCardTransaction
    Services
       begin;
        in/out withdrawal(input Quantity: currency, output
                          MyMoney:currency);
        in/out balance(output MoneyBalance: currency);
        int/out changeAddress(input NewAdd: string);
        end;
   Played_Roles
       CUSTOMER for ICreditCardTransaction ::=
                  (withdrawal ?(Quantity, MyMoney)
                  → withdrawal !(Quantity, MyMoney))
                  +
                  (balance ?(Quantity, MyMoney)
                   → balance !(Quantity, MyMoney))
                  +
                  (changeAdress ?(Newadd)
                  → changeAdress !(Newadd));
       BANK for ICreditCardTransaction:: =
                  ( withdrawal !(Quantity, MyMoney)
                  → withdrawal ?(Quantity, MyMoney))
                  +
                  (balance !(Quantity, MyMoney)
                   → balance ?(Quantity, MyMoney)
                  +
                  (changeAdress !(Newadd)
                  → changeAdress ?(Newadd));
7   Protocol
      BANKCOORDINATION ::= begin → STNCHRONIZE;
      SYNCHRONIZE ::=
        (CUSTOMER.withdrawal ?(Quantity, MyMoney)
         → BANK.withdrawal!(Quantity, MyMoney))→ SYNCHRONIZE
        +
        (CUSTOMER.balance ?(Quantity, MyMoney)
        → BANK.balance!(Quantity, MyMoney)) → SYNCHRONIZE
        +
        (CUSTOMER.changeAdress ?(Newadd))
        →(BANK.changeAdress !(Newadd)) → SYNCHRONIZE
        +
        (BANK.withdrawal ?(Quantity, MyMoney)
        → CUSTOMER.withdrawal!(Quantity, MyMoney))→SYNCHRONIZE
        +
        (BANK.balance ?(Quantity, MyMoney)
        → CUSTOMER.balance !(Quantity, MyMoney)) → SYNCHRONIZE
        +
        (BANK.changeAdress ?(Newadd))
        →(CUSTOMER.changeAdress !(Newadd)) → SYNCHRONIZE
        + end;
   End_Coordination Aspect BankCoordination;
```

Fig. 5. *BankCoordination* Coordination Aspect

3.1.3 Components and Connectors

A simple architectural element is specified with the set of ports, the aspects it is formed of, and the aspect weavings. It can be noticed from the aspects specification in section 3.1.2, that an aspect definition does not include the points where an aspect needs to coordinate with the rest of other aspects (aspect weavings). In this way,

aspects are completely maintainable and reusable. Therefore, when architectural elements are defined they import the aspect types from the PRISMA repository and define their weavings. An aspect weaving is specified by determining the aspects that participate in the weaving, the services of the aspects where they are weaved, and the weaving methods. A weaving that relates service s1 of aspect A1 and service s2 of aspect A2 can be specified using the following operators:

- A2.s2 **after** A1.s1: A2.s2 is executed after A1.s1
- A2.s2 **before** A1.s1: A2.s2 is executed before A1.s1
- A2.s2 **instead** A1.s1: A2.s2 is executed in place of A1.s1

```
Component Account

  Functional Aspect Import BankInteraction;
  Distribution Aspect Import ExtMbile;

  Weavings
    BankInteraction.changeAddress(NewAdd: string)
    before
    ExtMbile.move(NewAdd: string);
  End_Weaving;

  Ports
    AccountCnct: ICreditCardTransactions
                 Played_Roles BankInteraction.CUSTOMER;
    AccountSys: ICreditCardTransactions
                 Played_Roles BankInteraction.CUSTOMER;
  End_Ports;
End_Component Account;

Component ATM

  Functional Aspect Import BankInteraction;
  Distribution Aspect Import ExtMbile;

  Ports
    ATMCnct: ICreditCardTransactions
             Played_Roles BankInteraction.BANK;
  End_Ports;
End_Component ATM;
```

Fig. 6. Definition of Components

Simple architectural elements in PRISMA are components and connectors. A component is an architectural element that captures the functionality of the information system and does not act as a coordinator between other architectural elements; whereas, a connector is an architectural element that acts as a coordinator between other architectural elements. In order to better understand how to specify components and connectors, we show their syntax by means of the specification of the *ATM* and *Account* components and the connector that connects them.

The specified aspects in section 3.1.2 are going to be reused to define the architectural elements of the example. The same *ExtMbile* distribution aspect (see Figure 4) specifies the *Account* and *ATM* components (see Figure 6), and the

ATMAccount connector (see Figure 7). The *BankInteraction* functional aspect (see Figure 3) is reused in the *Account* and *ATM* components (see Figure 6). The *BankCoordination* coordination aspect (see Figure 5) is used to define the connector *ATMAccount* (see Figure 7). However, it is important to note that these aspects could be reused to create other architectural elements. We present the synchronization between aspects in the *Account* component; for example, when the customer address changes (*ChangeAddress*), the account of this customer must be moved to another place nearer to his/her new address (see weaving section in the *Account* component in Figure 6).

```
Connector ATMAccount

   Coordination Aspect Import BankCoordination;
   Distribution Aspect Import ExtMbile;

   Port
      ATM: ICreditCard_Transactions
           Played_Roles BankCoordination.BANK;
      Account: ICreditCard_Transactions
           Played_Roles BankCoordination.CUSTOMER;
   End_Port;
End_Connector ATMAccount;
```

Fig. 7. Definition of Connectors

3.1.4 Systems, Attachments and Bindings

PRISMA components can be simple or complex. The complex ones are called systems. A PRISMA system is a component that includes a set of connectors, components, and other systems that are correctly linked.

A system is specified as a pattern so that it can be reused in any software architecture that could be necessary. The difference between a system and a simple component is that it needs attachments and bindings:

-Attachments: They are connection relationships that establish the connection among ports of components and connectors.

- Bindings: They are connection relationships that establish the connection among the system (complex component) and the architectural elements it contains. The bindings allow the system to define its exterior behaviour (ports) by means of the architectural elements it contains.

Figure 8 shows the definition of the *SimpleBank* type. The set of architectural elements, that are necessary to define the system, are imported, and the number of instances that can be specified at configuration time are constrained. In the specification of the *SimpleBank* system (see Figure 8), this number is not specified and the default value (min=1, max=n) is applied to each type. In addition, the connections among the different types of architectural elements are specified in order to define the architectural pattern. The *SimpleBank* type has two types of attachments. They establish the connections among ports of components (*ATM, Account*) and ports of connectors (*ATMAccount*). Each type constrains the cardinality of the attachment at configuration time. The *SimpleBank* bindings establish the connection among the systems ports (*SimpleBank*) and the connectors or/and components ports (*Account*) that the system is composed of (see bindings section). Each type constrains the

cardinality of the binding at configuration time. The cardinality constrains of the system allow us to define a specific pattern of communication that must be satisfied at the configuration time.

```
System  SimpleBank
   Ports
      Banksystem: ICreditCardTransactions;
   End_Port;

   Import Architectural Elements ATM, Account, ATMAccount;

   Attachments
      Account.AccountCnct (1..n) ↔ (1..1) ATMAccount.Account;
      ATM.ATMCnct (1..n) ↔ (1..1) ATMAccount.ATM;
   End_Attachments;

   Bindings
      SimpleBank.Banksystem (1..1) ↔ (1..n) Account.AccountSys;
   End_Bindings;
                        End System SimpleBank;
```

Fig. 8. *SimpleBank* system specification

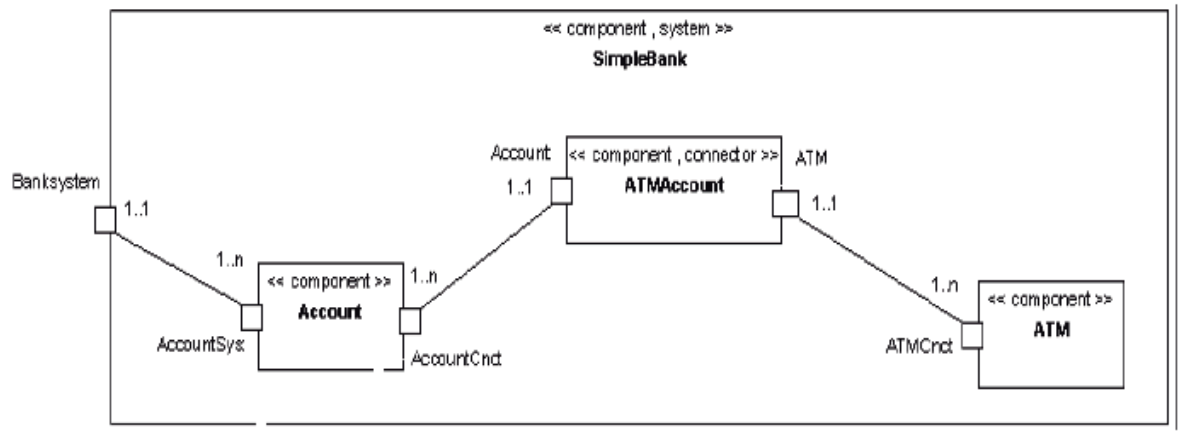


Fig. 9. *SimpleBank* system graphical representation[1]

Figure 9 illustrates the graphical view of the architecture that has been specified in Figure 8.

3.2 The Configuration Level

The configuration level is used to define a specific architectural model for a software system. In order to do this all required connectors, components and systems types should be instantiated, and attachment and binding instances should be added among them. At this moment, constraints that have been defined in systems are validated in order to ensure the pattern satisfaction. An example of a configuration is to define a

[1] The figure has been designed using the Poseidon Tool, http://www.gentleware.com

specific architectural model for a bank by reusing the system type that we have defined in the previous section (see Figure 10).

```
Architectural Model
   MySimpleBank = new SimpleBank(){
      MyFirstCustomer = new Account (0000001, Jose, Main Street 20,
                                     100, mainhost);
      MainATM = new ATM{ATM1, MainBranchATM, London Road, 1000000,
                        localhost);
      BankController = new ATMAccount()
      Attachments
       MyFirstCustomer.AccountCnct ←→ BankController.Account;
       MainATM.ATMCnct ←→ BankController.ATM;
      End_Attachments;
      Bindings
       MySimpleBank.Banksystem ←→ MyFirsCustormer.AccountSys;
      End_Bindings;
   };
End_Architectural Model;
```

Fig. 10. Architectural Model of the *MySimpleBank* Bank System

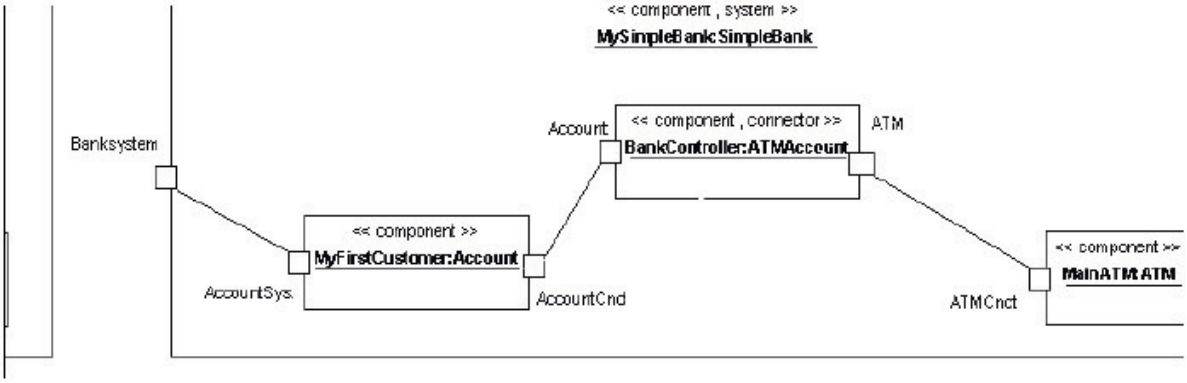


Fig. 11. Graphical representation of the *MySimpleBank* Architectural model[1]

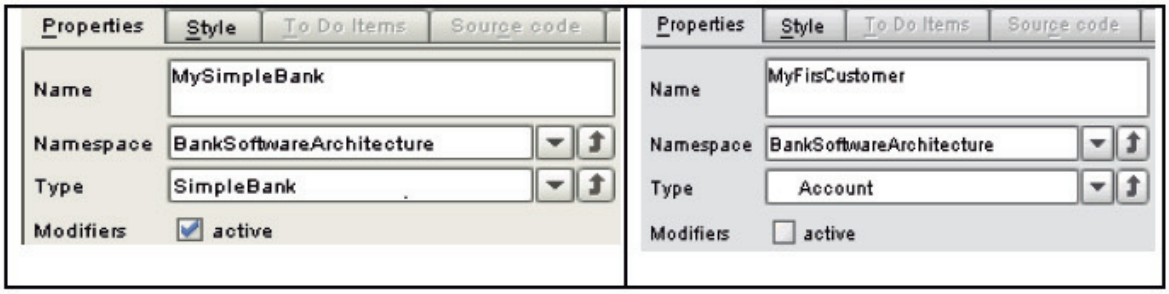


Fig. 12. Information related to the instantiation from the types of PRISMA library[1]

Figure 11, shows the graphical representation of the *MySimpleBank* system instance defined in Figure 10. Underlined names indicate that they are instances of a type (see Figure 12).

4 Conclusions and Further Work

In this paper, an AOADL to specify complex, dynamic and distributed information systems has been presented in detail. This language allows us to define PRISMA architectural models. The structure, design and maintainability of architectures specified in the PRISMA AOADL are improved by reusing entities at different levels of granularity (interfaces, aspects, components, connectors and systems). This reusability is achieved by means of the division of the language into two levels of abstraction and the integration of AOSD and CBSD into the language. The stored types defined at the type definition level can be reused by the configuration level to define a specific software architecture. In addition, the fact that interfaces and aspects are first-class citizens of the language increases the reusability because an interface can be used by several aspects and an aspect can be used by several architectural elements. The example of the paper has demonstrated this high level of reusability by reusing an interface to define two aspects, a distribution aspect to define two components and one connector, and other functional aspect to define two components.

We have used a simple example to present the language in order to facilitate the understanding of the language capabilities to the reader instead of using a complex one. However, it is important to keep in mind that PRISMA does not specify simple architectural systems for academic projects such as: pipelines, filters, blackboards, etc. PRISMA AOADL is being used to specify industrial projects where the software systems are complex, open, and active such as the *TeachMover* robot [16] and EFTCoR [17]. EFTCoR is a robot family that cleans the hulls of ships.

The PRISMA AOADL provides a better management of evolution and maintenance of crosscutting-concerns and software architectures. The maintenance of crosscutting-concerns is improved due to the fact that if we want to change the features of a specific concern, we only need to modify or change the aspect that defines the concern, and every architectural element that imports it will be updated. However, other approaches that use non-aspect-oriented ADLs need to look for each statement that is related to the concern in the tangled code of every architectural element of the system. The maintenance of software architecture is improved because the PRISMA approach supports evolution by means of a meta-level which provides a set of evolution services to evolve software architectures at run-time [16].

The PRISMA AOADL has a graphical notation that is based on a UML profile. As a result, PRISMA reduces the complexity in software development by providing a graphical notation and a modelling tool to support more intuitive and friendly software architecture modelling [14].

It is important to take into account that most ADLs only allow us to specify the skeleton of architectures and the services that are interchanged among their different architectural elements. However the PRISMA AOADL has a great expressive power to specify more features and requirements related with the software system by means of aspects in order to facilitate the code generation. We are currently developing the model compiler using DSL tools [6]. This is going to permit the compilation of the same PRISMA architectural model into different programming languages and technologies, thereby reducing the development time and preserving the traceability between an architectural model and its application code. We are improving the graphical modelling tool using DSL tools and we are starting to generate C# code and

PRISMA specifications from graphical PRISMA architectural models. Despite the fact that .NET framework does not provide support for the Aspect-Oriented approach, we are able to execute the C# code generated using our model compiler by developing a .NET middleware for our PRISMA approach called PRISMANET [15]. PRISMANET extends the .NET technology by the execution of aspects on the .NET platform, the reconfiguration of software architectures (local and distributed) and the addition and removal of aspects from components at run-time.

As future work, we are going to introduce validation and verification techniques in our modelling tool. Currently, we support cardinality constrains to define architectural patterns (systems). We are going to extend the language to support other kinds of constrains. In addition, we want to measure the benefits of the language with several case studies.

Acknowledgements

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, DYNAMICA project TIC2003-7804-C05-01.

References

1. Ali, N., Ramos, I., Carsí, J.A.: A Conceptual Model for Distributed Aspect-Oriented Software Architectures. International Conference on Information Technology Coding and Computing (ITCC), IEEE Computer Society, Las Vegas, NV, USA. (2005)
2. AOSD. Aspect-Oriented Software Development, http://aosd.net (2005)
3. Barais, O., Cariou, E., Duchien, L., Pessemier, N., Seinturier L.: "Transat: A framework for the specification of software architecture evolution". *In* ECOOP First *International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04)*, Oslo, June. http://wcat04.unex.es/. bib. (2004)
4. Constantinides, C.A., Elrad, T.: On the Requirements for Concurrent Software Architectures to Support Advanced Separation of Concerns. In proceedings of OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems. Available at: http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/constantinides.pdf. (2000)
5. Cuesta C.E., Romay M.P., De la Fuente P., Barrio-Solórzano M., Architectural Aspects of Architectural Aspects, Second European Workshop on Software Architecture (EWSA), Springer LNCS 3527, Pisa, June (2005)
6. Domain-Specific Language (DSL) Tools, http://lab.msdn.microsoft.com/ teamsystem/ workshop/dsltools/default.aspx
7. D'Souza, D., Wills, A. : "Objects, Components and Frameworks with UML: The Catalysis approach"; Addison-Wesley. (1999)
8. Garlan D., Perry D.: Introduction to the Special Issue on Software Architecture, IEEE Transactions on Software Engineering, 21(4), April (1995)
9. Katara, M., Katz, S.: Architectural Views of Aspects. International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, March, (2003)
10. Kiczales, G., Hilsdale, E., Huguin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag. (2001)

11. Medvidovic, N., Taylor, R. N.: A classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions of SW Engineering, Vol. 26, nº 1, January. (2000)
12. Milner, R.: π- Cálculo Poliadico: A tutorial. (1991)
13. Navasa, A., Perez, M.A., Murilllo J.M.: .Aspect Modelling at Architecture Desing. 2nd European Workshop in Software Architectures, Pisa, Italy, June, LNCS 3527, Springer Verlang. (2005)
14. Pérez, J., Navarro, E., Letelier, P., Ramos, I.: Graphical Modelling for Aspect Oriented SA, Proceedings on the 21st Annual ACM Symposium on Applied Computing (SAC), ACM ,Dijon, France, April 23 -27, 2006. (short paper)(accepted, to appear)
15. Pérez, J., Ali, N., Costa, C., Carsí, J. A., Ramos, I.: Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. 3rd International Conference on .NET Technologies, Plzen, Pilsen, Czech Republic, 30 May-1 June. (2005)
16. Pérez, J. Ali, N., Carsí, J.A., Ramos, I.: Dynamic Evolution in Aspect-Oriented Architectural Models. 2nd European Workshop in Software Architectures, Pisa, Italy, June, LNCS 3527, Springer Verlang. (2005)
17. Pérez, J., Ali, N., Ramos, I., Pastor, J.A., Sánchez, P., Álvarez, B. : Tele-operated Systems Development using the PRISMA approach. VIII conference on Software Engineering and Databases, Alicante, Spain. (2003) (in spanish).
18. Pinto, M., Fuentes, L., Troya, J. M.: DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. Generative Programming and Component Engineering: Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, Springer Verlag Computer Science, ISSN: 0302-9743. (2003)
19. Popovici, A., Gross, T., Alonso, G.: Dynamic Weaving for Aspect-Oriented Programming. In proceedings of the 1st international conference on Aspect-oriented software development, Enschede, The Netherlands, April. (2002)
20. Rajan, H., Sullivan, K., Eos: Instance-Level Aspects for Integrated System Design. In the proceedings of the 2003 Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Helsinki, Finland, September. (2003).
21. Rashid, A.: A Hybrid Approach to Separation of Concerns: The Story of SADESK. Proc. Reflection conf., Springer-Verlang, LNCS, 2192, pp. 231-249. (2001)
22. Sánchez, F.: Mask Model: Towards adaptability of synchronization restrictions in LCOO. PhD. dissertation, Extremadura University, Spain. (1999)
23. Schult, W. , Polze, A.: Aspect-Oriented Programming with C# and .NET. In 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing, (Washington, DC), IEEE Computer Society Press, pp.241-248. (2002)
24. Shaw M.: Procedure Calls Are the Assembly Language of Software Interconection: Connectors Deserve First-Class Status. Proceedings of Workshop on Studies of Software Design, January. (1994)
25. Soares, S., Laureano, E., Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. In proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, *OOPSLA'02*, Seattle, WA, USA, 174-190. (2002)
26. Suvee, D., Vanderperren, W., Jonckers, V.: JasCo: an Aspect-Oriented approach tailored for Component Based Software Development. In proceedings of the 2nd international conference on Aspect-oriented software development. Boston Massachusetts, March. (2003)
27. Szyperski C.: Component software: beyond object-oriented programming, ACM Press and Addison Wesley, New York, USA. (1998)
28. Tarr, P., Ossher, H., Harrison, W.H., Sutton, S. M.: "N Degrees pf Separation: Multi-Dimensional Separation of Concerns". Internacional Conference on Software Engineering (ICSE), ACM, pp. 1907-119. (1999)

A Modelling Proposal for Aspect-Oriented Software Architectures

Jennifer Pérez
Department of Information Systems and Computation, UPV
jeperez@dsic.upv.es

Elena Navarro
Department of Computer Science, UCLM
enavarro@info-ab.uclm.es

Patricio Letelier
Department of Information Systems and Computation, UPV
letelier@dsic.upv.es

Isidro Ramos
Department of Information Systems and Computation, UPV
iramos@dsic.upv.es

Abstract

Aspect-Oriented Programming (AOP) has emerged in recent years as a new paradigm for software development. PRISMA is an approach for developing complex and large software systems. It combines the Aspect-Oriented Software Development (AOSD) and the Component-Based Software Development (CBSD) in an elegant and novel way achieving a better management of crosscutting-concerns and software reusability. PRISMA approach proposes the separation of concerns from the very beginning of the software life-cycle in order to introduce them as reusable aspects of software architecture. PRISMA provides a framework to guide the development of complex and large software systems. The framework is composed by a modelling tool, a .NET middleware, and a compiler to automatically generate the application source code. In this paper, we present the modeling tool of the PRISMA framework and how it supports the reuse and maintenance improvements of the PRISMA model and its Aspect-Oriented Description Language (AOSD). We illustrate our proposal and modelling tool using a real-life case, the TeachMover robot.

1. Introduction

The complexity of current software systems and the fact that their non-functional requirements have become very relevant for the end-user are to challenges to be faced on every software development. Two new approaches of software development have emerged to overcome these needs: Component-Based Software Development (CBSD) [38] and Aspect-Oriented Software Development (AOSD) [5].

On the one hand, CBSD reduces the complexity of software development and improves its maintenance by increasing software reuse. CBSD decomposes the system into reusable entities called components. By extension, this advantage is provided by software architectures [11] due to the fact that architectural models are constructed using components. As a result, software architectures have emerged as a solution for design and development processes of complex software systems.

On the other hand, AOSD improves the quality of software and accurately satisfies the needs of the end-user giving equal importance to functional and non-functional requirements from the first stages of the software life-cycle. AOSD also reduces the complexity of software development and improves its maintenance by increasing the reusability of software. This approach allows the separation of concerns by modularizing crosscutting-concerns in a separate entity called *aspect*.

PRISMA is an approach that combines the CBSD and the AOSD to model software architectures. PRISMA provides the advantages of both approaches by means of its model and its Aspect-Oriented Architecture Description Language (AOADL). PRISMA takes advantage of the aspect-oriented approach to achieve more reusable and maintainable software architectures [33]. In this sense, it specifies the different *concerns* (safety, distribution, coordination, etc.) that *crosscut* a software system as aspects; these aspects are used to define PRISMA architectural elements (component, connector).

The PRISMA AOADL has a graphical notation that is based on a UML profile [26]. In this paper, we present how PRISMA reduces the complexity in software development by providing a graphical notation and a modelling tool to support more intuitive and friendly software architecture modelling. Moreover, we show how the reusability of aspects and components is supported by the modelling tool and facilitates the development of complex architectures.

The structure of the paper is the following: Section 2 presents the tele-operation domain and the *TeachMover* case study which is used through out the paper. Section 3 details the advantages that the PRISMA model obtains from the aspect-oriented approach. Section 4 presents how PRISMA allows the graphical modelling of software architectures taking advantage of its maintenance and reuse improvements; the graphical notation of the PRISMA AOADL, part of PRISMA UML profile and their implementation in a graphical editor are explained in detail. Section 5 compares related works on aspect-oriented software architectures. Finally, conclusions and further work are presented in section 6.

Pérez, J., Navarro, E., Letelier, P., Ramos, I. A Modelling Proposal for Aspect-Oriented Software Architectures. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). 2006.

2. Tele-operation Systems: The TeachMover Robot

There are many kinds of software systems whose large and complex software architectures must be reusable and maintainable. An example of software systems of this kind are the so-called tele-operation systems. Tele-operation systems perform risky activities, substituting human operators in dangerous areas. They are usually robots that are controlled by a human operator from a safe area. Specifically, we are working on the development of a tele-operation robot called *TeachMover* [9], we designed and implemented the PRISMA architecture of the *TeachMover* robot [39] (shown in Figure 1). We are going to illustrate the advantages of the PRISMA model and the facilities of the graphical editor using the *TeachMover* robot case study.

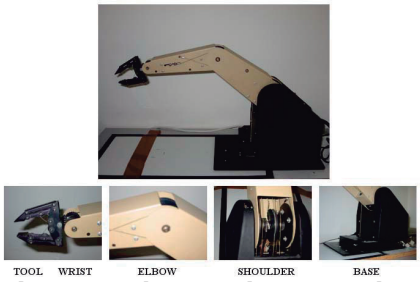


Figure 1. The *TeachMover* robot

The *TeachMover* is formed by a set of joints that permit the movement of the robot. These joints are: *Base, Shoulder, Elbow* and *Wrist*. In addition, it has a *Tool* to perform different tasks. In this case, the *Tool* is a gripper whose open and close actions allow the robot to pick up objects.

The functionality of the *TeachMover* robot is to move itself by means of *half-steps* or inverse cinematics (moving to a specific point in the space) at a specific speed. This functionality, together with the functionality of the *Tool*, allows the robot to move objects from an initial position to a final one. The movements of the robot are ordered by an operator from a computer. In addition, safety directives require that the *TeachMover* movements be checked in order to control that they are safe for the robot, the operator, and the environment which surrounds them.

The *TeachMover* architecture has a lot of components, connectors and their interactions at different levels of abstraction. The lowest level of the robot architecture has sensors and actuators as basic components. They are the software components that are communicated with the hardware joints of the robot. Actuators send commands to a joint to be performed; sensors are listeners of the results of the command executions. An actuator and a sensor are coordinated by means of a connector (see Figure 3). These three architectural elements (*actuator, sensor* and *SUCconnector*) (see Figure 3) are encapsulated inside a *system* called *Simple Unit Controller* (SUC) (see Figure 2). As a result, a SUC is a complex component (system) that manages the basic interactions of the joints of the *TeachMover*. Due to the complexity of this software system, in this paper, we only focus on the low levels of the hierarchical architecture of the *TeachMover* robot, i.e, on SUC. For more details about the specification see [32].

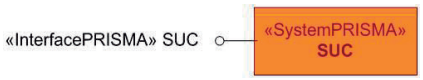


Figure 2. The external view of the SUC system

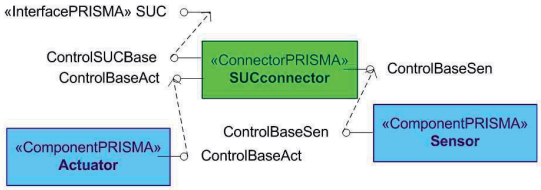


Figure 3. The internal view of the SUC system

3. A PRISMA overview

The main effort in the AOSD has been made at the implementation level, from which Aspect-Oriented Programming (AOP) has emerged as a new paradigm of software development. The AOSD was introduced in [15], where aspects were defined as properties that crosscut the software system, thereby achieving a reduction in tangled code. The concept of aspect, its advantages, and how it is weaved with the class (kernel) by means of the *pointcuts*, *joinpoints* and *advice* methods (after, before, around) are clear at the implementation level. However, not much work has been done concerning aspects at an architectural level. As a result, there is no consensus about what is an aspect in a software architecture and how aspects should weaved architectural elements.

3.1. PRISMA Approach

PRISMA approach has a formal model that uses AOSD principles to describe software architectures. A PRISMA aspect represents a specific *concern* for the software architecture under development. In PRISMA, from the very beginning of the specification all the behaviour is modularized in aspects, so that they can be imported by one or more architectural elements in order to properly manage *crosscutting-concerns*. A PRISMA architectural element (components and connectors) is formed by a set of aspects that describe it from the different *concerns* of the architecture. The kind of aspect (safety, coordination, distribution, etc.) that forms an architectural element depends on the *concerns* of the software system that we are specifying. In the case of the *TeachMover* software system, the concerns that crosscut the *SUC* system are the following:

- **Functional aspect:** This kind of aspect is necessary to specify the computation of the joints and *Tool* of the *Teachmover* system.
- **Coordination aspect:** This kind of aspect is necessary to specify the synchronization between the different architectural elements that form the *TeachMover* architecture.
- **Safety aspect:** This kind of aspect is necessary because safety properties are extremely important to establish a safety control of the robot movements. The movements must be safe for the robot itself and for the rest of the elements of its environment.

Each architectural element of a PRISMA model is formed by a set of aspects that describe it from the different *concerns* of the architecture. Architectural elements can be analyzed from two different views: internal (*white box view)* and external (*black box view*). The *white box view* shows an architectural element as a prism, each side of which is an aspect of this architectural element. The *black box view* encapsulates the architectural element functionality and publishes a set of services that it offers to other architectural elements.

There are three kinds of architectural elements: components, connectors and systems. All of them share the fact that they are formed by a set of aspects and their synchronization relationships (weavings).

- A *component* is an architectural element that captures the functionality of the system and does not act as a coordinator among other elements. It is formed by an identifier, a set of aspects together with their synchronization relationships (*aspect weavings*), and one or more ports. These ports represent the interaction points among architectural elements and are typed by interfaces. An interface publishes a set of services that can be required or provided by the component.
- A *connector* is an architectural element that acts as a coordinator among components. It is formed by an identifier, a set of aspects together with their synchronization relationships (*aspect weavings*), and one or more ports. These ports represent interaction points among architectural elements and are typed by interfaces. An interface publishes a set of services that can be required or provided by the connector.
- A *system* is a complex component that includes a set of architectural elements (connectors, components, systems) that are connected among them and a set of ports to publish the services required or provided by the system. A system definition must specify the connection relationships (*attachments*) and the composition relationships (*bindings*) among the architectural elements that it includes. *Attachments* establish the connection among ports of components and roles of connectors, whereas *bindings* define the composition between the ports of the system and the ports of the architectural elements which the system contains.

3.2. PRISMA AOADL

PRISMA has an AODL to define reusable architectural elements and aspects; as well as specific architectures by creating and interconnecting the instances of the defined reusable types. It is based on the OASIS language [18] in order to define its semantics in a formal way and to preserve its main advantages, i.e., the validation and the verification of architectural models and the automatic generation code. A full description of the language is out of the scope of this paper. Here, we are going to focus on how PRISMA takes advantage of the aspect specification.

3.2.1. Reusing Aspects

PRISMA aspects are first-order citizens of the language and are specified without considering which architectural elements will import them in the future. When an aspect is defined, it is stored in the PRISMA repository to be reused by different architectural elements of the same or of different architectural models.

Table 1. The SMotion safety aspect

```
Safety Aspect SMotion
  Attributes
    Constant   minimum, maximum: integer,
                                  NOT NULL;
  Services
    begin( input InitialMinimum: integer,
           input InitialMaximum: integer);… …
    in check(input Degrees: integer,
           output Secure: boolean);… …
    in controlSpeed(input CurrentSpeed: integer,
           output Secure: boolean);… …
  Operations
  transaction
    DANGEROUSCHECKING(input Degrees: integer,
           input CurrentSpeed: integer,
           output Secure: boolean): … …
  Protocol
    SMOTION = begin.CHECKING;
    CHECKING= check (Degrees, Secure)+
       DANGEROUSCHECKING(Degrees, Secure) + end;
End_Safety Aspect SMotion;
```

Most aspect-oriented approaches have been influenced by the aspect definition of aspect-oriented programming languages such as AspectJ [15]. For this reason, these approaches often associate aspects to non-functional requirements. However, not only does PRISMA consider non-functional requirements as aspects, but it also specifies functionality as an aspect in order to facilitate its reuse. In fact, some research works point out the fact that the functionality can be subdivided into different aspects, such as works [21] and [22]. An example of a functional aspect in the *TeachMover* case study is the interaction with the robot; therefore, this aspect can be reused for defining components of any architectural model of a robotic system.

The fact that PRISMA architectural elements do not have a kernel class (asymmetrical model [12]) provides many advantages to the model. One of these advantages is that it facilitates the construction of an architectural model

architectures. Specifically, the SUCConnector is imported to define the SUC system, whose specification is not provided by space reasons. In this way, the SUC system is instantiated for each joint and tool of the robot to form the *TeachMover* architecture (*configuration level*). An example of this instantiation is the *Base* joint that is shown next:

```
BaseSUC = new SUC(){
   BaseActuator= new Actuator(localhost);
   BaseSensor= new Sensor(localhost);
   BaseConnector = new
         SUCconnector(0,0,localhost,90,-90);};
```

4. Modelling PRISMA Architectures

The specification of software architectures of complex and large software systems from the early stages of the software life-cycle is the main goal of the PRISMA framework. It provides support to the PRISMA model [30] and its AOADL to define and generate software architectures.

The PRISMA framework is composed of the EGV-PRISMA graphical editor, a .NET middleware and a compiler that automatically generates the source code of applications. Our graphical editor has not been developed from scratch. We have used the Microsoft Office Visio 2003 [40] along with its UML template to support our PRISMA UML profile and to take advantage of the functionality that is already provided by this element. EGV-PRISMA allows us to specify, in a graphical way, PRISMA architectures and to generate automatically their PRISMA textual specification and/or their XML document.

4.1. Profile PRISMA-UML

The PRISMA textual language could be used directly to model software architectures. However, for any modern modelling approach, it is important to have a visual representation of the specifications. This makes the application of PRISMA in the context of a modelling tool easier and more practical. Obviously, a textual representation is more suitable for some details of a PRISMA specification; for instance, to formulate the changes in the value of attributes by the execution of aspect services. Thus, only the main concepts and their relationships are visually specified; the rest of the concepts are represented in textual form and are included in the definition of the corresponding symbols. We consider UML to be relevant as a graphical notation for PRISMA because it is a standard and popular notation. Although UML is intended for object-oriented modelling, thanks to its extension mechanisms that are associated to the definition of a UML profile, it can be customized to the particular needs of PRISMA specifications. These extension mechanisms are stereotypes, tagged values, and constraints, which are all used to define new derived concepts (metaclasses) from the standard UML metaclasses. Thus, a set of specific extensions is called a UML profile. Most CASE tools are based on UML and allow us to work with these kinds of extensions through UML profiles; however, there is currently no full support for all the functionality that might be expected. For software architecture modelling, UML 2.0 [26] includes the following concepts: component, connector, port and interface (required or provided). However, the provided expressivity is too basic in comparison with PRISMA. Furthermore, UML 2.0 does not include the aspect and weaving concepts.

We have established a PRISMA profile that includes all the necessary extensions for using UML as a visual notation for PRISMA specifications. This profile has been implemented by extending an existing one as we describe in the following section. Throughout the definition process of the PRISMA profile, especially for AOSD elements, we took into account the satisfaction of the requirements that Aldawud et al [2] stated for defining a UML profile for AOSD.

Following, we summarize the part of the PRISMA profile that is most related to aspect orientation. In this case, we are basically interested in representing aspects and weaving relationships. With this aim, we have derived these concepts extending the UML metaclasses. Table 3 shows the graphical notation used for representing aspect-oriented modelling in PRISMA. With respect to reuse, PRISMA allows us to define aspect types. There are some predefined stereotypes for modelling these aspect types, such as: «Quality», «Presentation», «Contex-awareness», «Coordination», «Navegation», «Functional», «Distribution», «Safety» and «Replication». As shown in Table 3, they are specializations of the stereotype «AspectPRISMA» and they have a common notation.

Similar to other proposals that extend UML with aspect-oriented modelling, we allow the visualization of aspects and weaving relationships. We have used the metaclass UML *Classifier* to derive the Aspect stereotype. Thus, the Aspect stereotype takes advantage of the structural and behaviour features that are available for UML metaclasses. We have chosen *Dependency* as the base class to represent weaving relationships. These selected metaclasses are the common choice in UML extensions for modelling aspects [2][37]. In this sense, the main differences refer to the context in which aspect orientation is used.

Table 3. The UML PRISMA profile related to aspect-oriented modeling

Stereotype	Base Class	Parent	Notation
ComponentPRISMA «ComponentPRISMA»	Component		«InterfacePRISMA» «ComponentPRISMA»
Connector «ConnectorPRISMA»	Component		«InterfacePRISMA» «ConnectorPRISMA»
AspectPRISMA «AspectPRISMA»	Classifier		Not represented explicitly
AspectTypePRISMA «*AspectType*»		AspectPRISMA	«AspectType» Name
WeavingPRISMA «WeavingPRISMA»	Dependency		«WeavingPRISMA»

In our case, our concern is software architecture modelling according to PRISMA expressivity; however, in [2][37], for example, the extensions are made in the context of aspect design and programming.



Figure 4. External view of a connector

Figures 4 and 5 show diagrams that use the PRISMA profile. The external view shows components, connectors and their attachments, as an example Figure 4 shows the connector. For simplicity, we have decided to visualize in a separate view (internal view) the aspects of an architectural element. Thus, Figure 5 shows the internal view of the connector SUCConnector with its two aspects: CProcessSUC and SMotion, whose stereotypes are «Coordination» and «Safety», respectively.

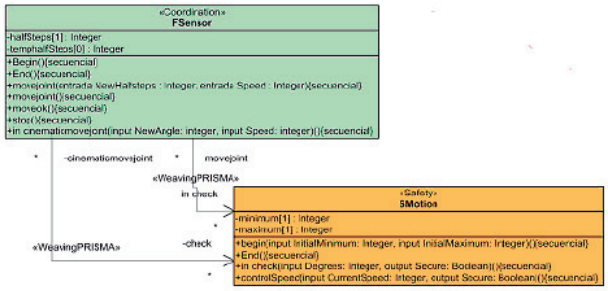


Figure 5. Internal view of the SUCconnector

4.2. Graphical Editor of PRISMA

One of the advantages of defining a UML profile is that tool support can be easily obtained by means of available tools based on UML, such as Rational Rose [36], Argo UML [3], Visio [40], etc. Most of them provide mechanisms to extend their functionality. This means that there is no need to develop a tool from scratch but extending any of the existing ones. We considered and reviewed several of these tools as support for PRISMA modelling, but, we finally selected Visio for two reasons:

- Visio allows straightforward management, both for using and modifying, shapes. This characteristic is highly relevant for our purposes because all the kinds of concepts that are included in our profile can easily have different shapes related to their functionality. A different colour is assigned to each aspect in order to improve the legibility of the model. Since there is no support in UML 1.5. for the port and role concepts, graphical shapes to model components and connectors have to be modified to describe them.
- An add-in is also available from Visio to translate a UML model to XMI. This means that other tools such as those listed in [27] can be used for semantic analysis. This allows for simple consistency checks and type checking in terms of defined OCL constraints of the PRISMA profile.

A lightweight extension of Visio that we have called EGV-PRISMA has been developed by means of a template to support the above–mentioned needs of the PRISMA profile. Therefore, whenever a new system has to be modelled, the EGV-PRISMA template is instantiated to create a new document and make the GUI appear as in Figure 6, where three main areas can be observed from right to left: the developed *Stencils*, the *Work Area* and the *Model Explorer*. In addition, a *PRISMA menu* has been added to those provided by default.

Visual modelling in Visio is accomplished by means of *Stencils*. They are galleries of shapes which are organized according to their purposes, uses, etc. In the development of the EGV-PRISMA, two stencils were specifically developed using the shapes that the Visio UML template provides. They are *Aspect and interface PRISMA* and *Main Structure PRISMA*. As can be observed in Figure 7 (a), the former has been defined to allow the modelling of those aspects that are currently supported by PRISMA

and the *weaving* relationships. The latter shows the stencil for modelling *components*, *connectors* and *systems* along with the relationships made available to them, i.e., *binding* and *attachment*, to be properly glued together (Figure 7 (b)).

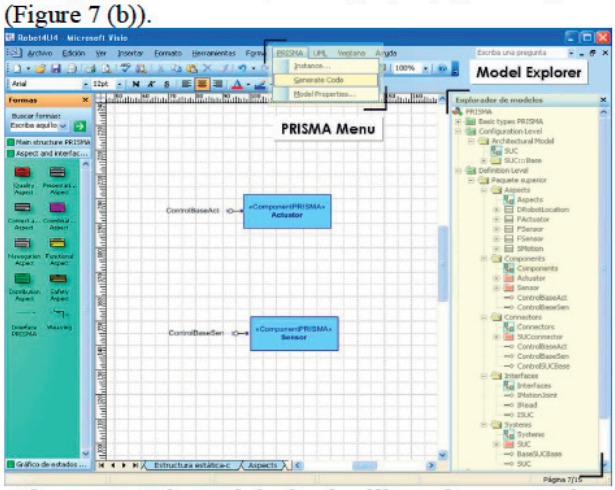
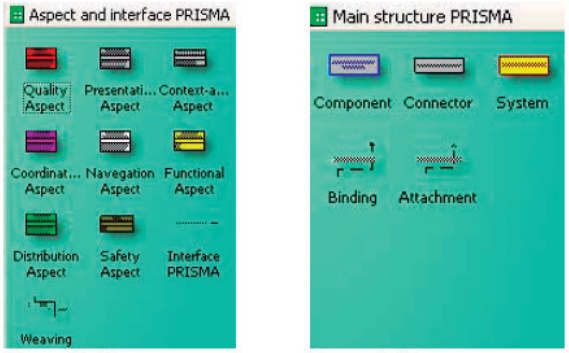


Figure 6. What Visio looks like whenever the EGV-PRISMA template is loaded

Each shape on these stencils is the visual representation of a stereotype that is defined in the PRISMA profile. This means that it inherits all the properties from the base class used for the stereotype in conjunction with others that are intrinsic. In this sense, most of the shapes in the *Aspect and Interface PRISMA* stencil allow us to model aspects: quality, presentation, context-awareness, coordination, navigation, functional, distribution and safety. As *classifier* is the selected base class, each aspect uses the utilities provided by the Visio UML template to specify the attributes and services of each aspect. Additionally, a new utility has been developed in the EGV-PRISMA to allow the architect to establish certain properties that are only made available to aspects, i.e., to type each service as *in*, *out* or *in/out*, to describe their preconditions, triggers and played_roles (subprocesses) (Figure 8).

In addition, a shape for representing the weaving relationship is shown in the same stencil that uses *Dependency* as the base class. This shape also has a specific form, shown in Figure 7 (a), which not only allows the specification of both the initial and final services and the kind of weaving relationship (*after*, *before* and *instead*), but also allows an obligation semantic if it is needed. In this way, *afterIf*, *beforeIf* or *insteadIf*, as described above, can be easily specified according to a specific condition (Figure 9).



(a) Modeling PRISMA aspects and interfaces

(b) Modeling components, connectors and systems

Figure 7. Developed stencils for PRISMA

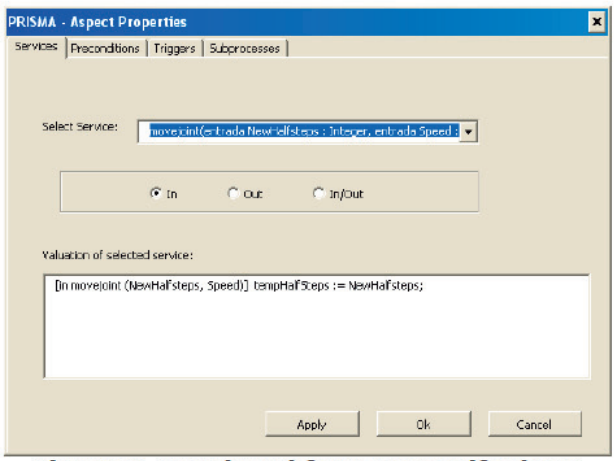


Figure 8. Developed form to specify those specific properties to aspects: kinds of service, preconditions, triggers and subprocesses.

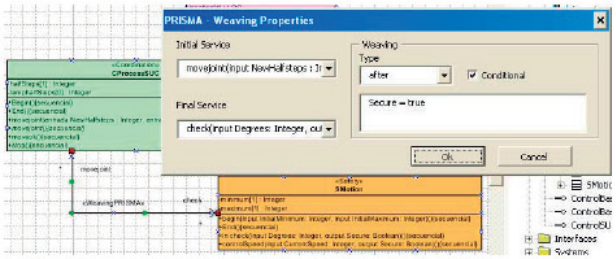


Figure 9. EGV-PRISMA provides support to model weaving relationships.

The *Main Structure PRISMA* stencil (Figure 7 (b)) works in a similar manner to the one previously described. The main difference is its purpose, because it is intended to model the main elements in PRISMA. Therefore, *component*, *connector* and *system* are visually represented by means of shapes specified on this stencil which are the visual representation of the specified stereotypes having *Component* as the base class. Other modifiable properties have been added to those inherited

from the Visio UML template. For instance, a connector needs the definition of its roles by specifying its name, the associated interface, and the pair aspect-subprocess (Figure 10) which provides the role with the semantic.

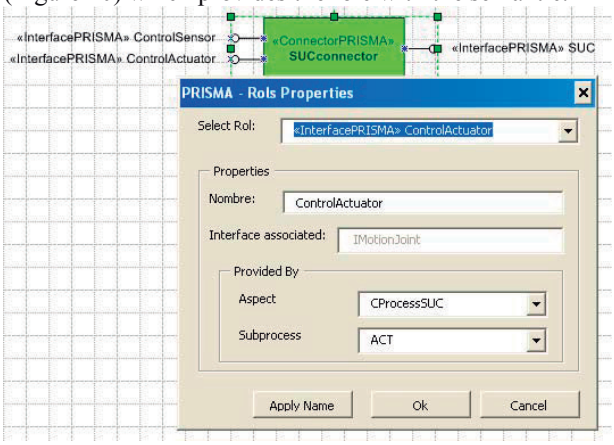


Figure 10. Describing roles when a connector is defined

As stated above, another main area in the EGV-PRISMA is the *Model Explorer* (see Figure 6) that is reused from the Visio UML template. It provides a tree-view of the PRISMA model being defined to facilitate the navigation through the model, i.e., a hierarchy in which each PRISMA element or diagram is represented by an icon. The Model Explorer has been customized to provide the architect with guidance throughout the process in a similar way to how she/he would have fulfilled the PRISMA AOADL. In this sense, two levels can be distinguished:

- *Definition Level* which provides the view of the PRISMA repository where every PRISMA element is defined. For this reason it is structured in several packages: Aspects, Components, Connectors and Systems. For instance, we can observe in Figure 9 the *SMotion aspect* that has been defined by dragging and dropping a Safety Aspect from the *Aspect and Interface* stencil onto the *Aspect* page. In a similar way, *Actuator* and *Sensor* components and the *SUCConnector* connector were defined (see Figure 6). It also facilitates the reuse of every element defined on the repository by just dragging it to the locations where it is needed.
- *Configuration Level* which allows the user to access to the instances of the PRISMA model, i.e., the Architectural Model. For instance, in our case study we can see on the Model Explorer where the *Base* system, an instance of the SUC system, appears. This instance has been defined by means of the *Instance* item available on the PRISMA menu.

We would like to point out another functionality that has been added to Visio by means of the EGV-PRISMA. It is related to the translation from the graphical model to the textual PRISMA ADL. It can be observed in Figure 6 that a new item, called *Generate code,* has been added to the *PRISMA menu*; in such a way that the currently visually defined model is translated into the PRISMA AODL by just clicking on it. This means that the on-going compiler will allow the generation of C# code from the graphical notation.

5. Related Work

A wide variety of ADLs have been proposed, an interesting comparison between these languages has been made in the work by [19]. Some proposals for the integration of the software architecture and the AOSD have emerged in order to take advantage of both approaches [7]. Each one introduces aspects in their Architecture Description Languages (ADLs) in a different way: as a component [23], as a connector among components [35], as a view of the architecture [14], etc. However, PRISMA introduces the aspect in its ADL as a new concept without simulating it with another architectural term (components, connectors, views, etc). Our approach does not force us to have an architectural specification of the system to introduce aspects as new requirements [23], we take advantage of the notion of aspect from the beginning of the system definition. Also, the complete view of the software architecture is not lost by the use of aspects as [14]. Moreover, at the configuration level, when the architectural elements are instantiated, we do not lose the structural and architectonical view of the system due to the fact that our components are connected by means of connectors instead of aspects [35]. As Shaw presents in her work [29], the specification of software systems with complex coordination protocols is too difficult without the connector architectural element. This is because the connector provides the separation of the component interaction achieving a higher level of abstraction, modularity and architectural view of the system. TranSAT [6] is another approach that incorporates technical aspects that refine the original component specification and generates another one that includes them without losing the black box view of the component.

Another difference of PRISMA when comparing it with other ADLs is that its aim is beyond of providing only a language for modelling software architectures. The PRISMA framework has a purpose based on the Model-Driven Architecture (MDA) approach, proposed by the OMG group [25] for system interoperability and integration. MDA encourages modelling by using UML extensions that are independent from the final implementation platforms, by separating concerns and establishing different abstraction levels during the modelling process. Thus, transformation techniques and inter-level relationships are used to produce the final implementation. Furthermore, in the context of a tool, a

graphical notation for the PRISMA AOADL is provided in order to make easier the specification of aspect-oriented in software architecture. There are several works that study the use of UML for the description of software architectures [10] [16]. UML 1.5 version is not a whole solution as a graphical notation for ADLs, because there are some elements (ports, connectors, etc.) that do not have a direct representation. For this reason, the UML extension mechanisms [17][20] behind the concept of UML profile are needed to provide a UML graphical notation. UML 2.0 include some of these elements, but regarding aspect and weaving concepts a UML profile is the usual option.

On the other hand, in [8], the main requirements for developing aspect-oriented software architectures are presented. They are the linguistic support for weaving aspects, dynamic adaptability, and reusability. PRISMA and its ADL are presented as a good solution to specify aspect-oriented software architectures because they satisfy these requirements. It is important to keep in mind that PRISMA does not specify simple architectural systems for academic projects such as: pipelines, filters, blackboards, etc. PRISMA AOADL is being used to specify industrial projects where the software systems are complex, open, and active such as the *TeachMover*.

Finally, there are some tool that give support to architecture description languages such as ACME Studio [1], Software Architect's Assistant[24] or CommUnity WorkBench [28]. In addition, there is a wide variety of tools under construction following the Aspect Oriented Paradigm to provide support at different stages of the software development and following different technologies [4]. However, there are not many tools which provide support for AOADL as EGV-PRISMA does. In this sense, tools DAOP-ADTool or ConcernBase [13] for supporting DAOP-ADL [35] or Perspectival Concern-Space Framework, respectively, could be clear examples for evaluation purposes. However, these tools are currently under development and almost no information is available. DAOP-ADTool provides some help for visual modelling and a full view of the XML description of components and aspects. However, this tool does not include any utility similar to the Model Explorer, which is very helpful to facilitate an improved view of and access to the architecture under description.

6. Conclusions and Further Work

In this paper, we present the modelling tool of the PRISMA framework and how it supports the reuse and maintenance improvements of the PRISMA model and its Aspect-Oriented Architecture Description Language.

We have presented how the AOADL that supports the PRISMA model allows the reusability of aspects due to the fact that aspects are first-order citizens and the weavings are established outside them. In addition, the formal specification of PRISMA AOADL contributes to facilitate the automatic code generation of software architectures. We have demonstrated the reuse capabilities of the PRISMA model using the specification of the *TeachMover* case study, as well as the flexible properties of modelling by means of its tool and graphical notation (EGV-PRISMA). The case study has been totally specified and some of its components are being reused in a more complex tele-operated system called EFTCoR [9], to clean the hulls of ships. The case study has also been implemented by extending the classes provided by the PRISMA middleware. Therefore, PRISMA is a mature and well supported approach for developing large and complex software systems.

We are currently developing the PRISMA compiler in order to automatically generate C# code from graphical specification. In addition, the evolution capabilities of the middleware are being improved to add and remove components and aspects from architectural models at runtime. In addition, the EGV-PRISMA will be extended to include specific editors to manage formulae in a more intuitive way.

7. Acknowledgments

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, DYNAMICA project TIC2003-07776-C02-02.

8. References

[1] AcmeStudio, http://www.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio html

[2] Aldawud, O., Elrad T. and Bader A. UML profile for Aspect Oriented Software Development, In the Aspect-Oriented UML Workshop, Collocated to the Aspect Oriented Software Development Conference, Boston, USA March 18, 2003.

[3] ArgoUML, http://argouml.tigris.org/

[4] AOSD-EUROPE network of excellence, http://www.aosd-europe.net/

[5] Aspect-Oriented Software Development http://aosd net

[6] Barais, O., Cariou, E., Duchien, L., Pessemier, N. Seinturier, L. Transat: A framework for the specification of software architecture evolution, In ECOOP First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04), Oslo, June. http://wcat04.unex.es/. bib

[7] Cuesta C.E., Romay M.P., De la Fuente P., Barrio-Solórzano M., Architectural Aspects of Architectural Aspects, Second European Workshop on Software Architecture (EWSA), Springer LNCS 3527, Pisa, June (2005)

[8] Constantinides, C. A. and Elrad, T. On the Requirements for Concurrent Software Architectures to Support Advanced Separation of Concerns, In Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, http://trese.cs.utwente nl/Workshops/OOPSLA2000/papers/constantinides.pdf

[9] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794. 2002
[10] Garlan D. and Kompanek, A.Reconciling the Needs of Architectural Description with Object-Modeling Notations, In Proceedings «UML» 2000,LNCS from Springer Verlag, York, UK, October , 2000, 498-512.
[11] Garlan D., Perry D.: Introduction to the Special Issue on Software Architecture, IEEE Transactions on Software Engineering, 21(4), April (1995)
[12] Harrison W.H., Ossher H.L., Tarr P.L., Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition, IBM Research Report RC22685 (W0212-147),Thomas J. Watson Research Center IBM, December 2002.
[13] Kande, M. M., "A concern-oriented approach to software architecture," in Computer Science, vol. PhD. Lausanne,Switzerland: Swiss Federal Institute of Technology (EPFL), 2003.
[14] Katara, M., Katz, S.: Architectural Views of Aspects. International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, March, (2003)
[15] Kiczales, G., Hilsdale, E., Huguin, J., Kersten, M., Palmand, J., Griswold, W. G. An Overview of AspectJ, In proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 2001.
[16] Kobrym C. Modeling Components and Frameworks with UML. Comunications of ACM, 43(10) 2000, 31-38.
[17] Kruchten, P. Selic, B. and Kozaczynski W. Describing Software Architecture with UML, In Proceedings of ICSE 2001, Toronto, Ontario, Canada, May 2001, 715-716
[18] Letelier, P., Sánchez, P., Ramos, I. and Pastor, O. OASIS 3.0, A formal language for the object oriented conceptual modeling, Polytechnic University of Valencia, SPUPV-98.4011, ISBN 84-7721-663-0, 1998,(In Spanish).
[19] Medvidovic, N. and Taylor, R. N. A classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions of SW Engineering, Vol. 26, nº 1, January, 2000
[20] Monge, R., C. Alves, and A. Vallecillo, A Graphical Representation of COTS-based Software Architectures, In Proc IDEAS, La Habana, 2002.
[21] Moreira, A. M. D., Araújo, J., Rashid, A. A Concern-Oriented Requirements Engineering Model. Proceedings 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005: 293-308
[22] Navarro, E., Letelier, P., Ramos, I. "Goals and Quality Characteristics: Separating Concerns", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, collocated to OOPSLA 2004, Monday, October 25, 2004, Vancouver, Canada.
[23] Navasa A., Pérez M.A., Murillo J.M.: Aspect Modelling at Architecture Design, 2nd European Workshop on Software Architecture, Springer LNCS 3527, Pisa, June (2005)
[24] Ng K., Kramer J., Automated Support for Distributed Software Design, Proceedings of the 7th International Workshop on Computer-Aided Software Engineering (CASE '95), Toronto, July 1995
[25] OMG Model Driven Architecture, http://www.omg.org/mda/
[26] Object Management Group (OMG), Unified Modeling Language: superstructure specification, http://www.uml.org/
[27] OCL tools & services http://www.klasse.nl/ocl/ocl-services html
[28] Oliveira, C. and Wermelinger, M. The CommUnity Workbench. In Proceedings of the 26th international Conference on Software Engineering (May 23 - 28). ICSE '04. IEEE Computer Society, 709-710. 2004 - Conference Formal Demo
[29] M. Shaw, Procedure Calls Are the Assembly Language of Software Interconection: Connectors Deserve First-Class Status, *Proceedings of Workshop on Studies of Software Design*, Enero, 1994.
[30] Pérez J. Ali N., Carsí J.A., Ramos I., Dynamic Evolution in Aspect-Oriented Architectural Models, Second European Workshop on Software Architecture (EWSA), Springer LNCS 3527, Pisa, June (2005)
[31] Pérez J., Ali, N., Costa, C., Carsí, J.A., Ramos, I., Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology, International Conference on .NET Technologies, Plzen, Pilsen, Czech Republic,may-june, 2005.
[32] Perez, J., Ali, N., Carsí J.A., Ramos I., The PRISMA architecture of the 4u4 robot case study, *Technical Report DSIC II/13/04, Polytechnic University of Valencia*, 72 pages, Spain, 2004 (in Spanish)
[33] Perez, J., Ramos, I., Jaén, J., Letelier, P and Navarro E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures, *In proceedings of 3rd IEEE International Conference on Quality Software (QSIC 2003)*, Dallas, Texas, USA, November, 2003.
[34] Pérez, J., Ali., N.H., Ramos, I., Pastor, J. A., Sánchez, P.and Álvarez B. Tele-operated Systems Development using the PRISMA approach, *VIII Conference on Software Engineering and Databases*, Alicante, Spain, 2003 (In Spanish)
[35] Pinto, M., Fuentes, L., Troya, J.M. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development, *Generative Programming and Component Engineering: Second International Conference, GPCE 2003*, Erfurt, Germany, September 22-25, Springer Verlag Computer Science, ISSN: 0302-9743
[36] Rational Rose, http://www-306.ibm.com/software/rational/
[37] Suzuki, J., Yamamoto, Y. Extending UML with Aspects: Aspect Support in the design phase, 3rd Workshop on Aspect-oriented Programming en European Conference on Object Oriented Programming(ECOOP), 299-300, 1999.
[38] Szyperski, C. Component software: beyond object-oriented programming, New York, USA: *ACM Press and Addison Wesley*, 1998.
[39] The *TeachMover* Robot, http://www.questechzone.com/microbot/teachmover htm
[40] Visio 2003, http://office microsoft.com/es-es/FX010857983082.aspx

A Goal-Oriented Approach for Safety Requirements Specification

Elena Navarro[†], Pedro Sánchez[‡], Patricio Letelier[∫], Juan A. Pastor[‡] and Isidro Ramos[∫]
[†]*Computer Science Department, UCLM*
[‡]*Information Technology and Communication, UPCT*
[∫]*Department of Information Systems and Computation, UPV*
[†]*enavarro@info-ab.uclm.es*
[‡]*{pedro.sanchez| juanangel.pastor }@upct.es*
[∫]*{letelier|iramos@dsic.upv.es*

Abstract

Robotic systems are developed to execute tasks with several types of risks associated. The possible damages that can affect both the working environment and the self system lead us to consider that these systems are safety critical, i.e., systems where the strict management of safety aspects is vital. In this work we introduce our proposal for the consideration of safety related requirements and their consequent trace to the desired final system architecture. For this reason, this paper gives a procedure for the identification and specification of safety requirements based on a goal oriented framework. Moreover, in this work other approaches have been considered and integrated to deal with well known safety standard recommendations. By means of an industrial case study, we show how this proposal can be used to consider safety requirements in tele operated robotic systems and, by extrapolation, in other critical domains.

1. Motivation

Robotics systems are substantially different from other software applications because it is mandatory to consider aspects such as interaction with the environment, presence of perturbations, etc. Moreover, if their use has an important impact on persons and equipment when errors arise then safety aspects must be considered during their development. Besides the risks inherent to their use, work places are higher risk areas. Douglass [2] gives, among others, the following list of damage sources: errors in the execution of the control system (hardware and software), people who access to forbidden walking areas, human errors, broken mechanical parts, liberation of stored energy, and so on.

Nowadays, when we analyze the development of tele operated systems, we can observe that there is no well known integration of safety requirements within the process of requirements specification as whole, nor are there methodologies supporting it in an integrated way. To meet this deficiency, we have considered the *goal oriented* framework ATRIUM [16] for guiding the requirement engineering process. ATRIUM offers a methodology for defining both the requirement specification and the software architecture of the system. This framework has been extended for considering *safety* requirements by following the ANSI/RIA R15.06 1999 [1] standard. Furthermore, our proposal has been enriched both with the ideas by Lemos [11] for the division of operation mode of systems, and the patterns and heuristics given by Douglass [2] for the consideration of this kind of requirement.

This work has been validated in the context of the EFTCoR project [4] funded by the European Union. This kind of system, by its nature, entails a greater probability of dangers than others. Thus a precise identification, specification and trace of safety requirements turn out essential.

The remainder of the paper is organized as follows: In Section 2, we present the conceptual framework considered. In Section 3, we describe the process of specifying a complete requirements specification, integrating safety requirements, by following a goal oriented approach. The case study is introduced in Section 4 by explaining the real situation where these ideas have been put into practice. Section 5 gives an example of pattern use for the materialization of safety requirements. Section 6 looks at some related work and finally the paper is concluded in Section 7 with future work.

Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)
0-7695-2546-6/06 $20.00 © 2006 IEEE
Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. A Goal-Oriented Approach for Safety Requirements Specification. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06). 2006.

2. Conceptual Framework

In order to clarify the following discussion, the main concepts used are defined in this section. Hence, it is mandatory to define which meaning of *Safety requirement* has been used from the set of definitions in the bibliography. ANSI/RIA R15.06 1999 standard, which this work is based on, defines Safety requirements as those to be satisfied for any industrial robotic system to assure the safety of personnel associated with its use. In this work, this definition has been extended, with Levesons' ideas [15], by including damage or destruction of property or injury or damage to any living being, especially, human beings.

When the behaviour of a system is being described, *tasks* to be provided by any system component, whether software or hardware, have also to be described. They mainly refer to that functionality expected from the system, as for instance, motions of any element of the robot, coordinated motions, use of tools, etc. The execution of these tasks is the potential source of damage or injuries the system can cause, so that the most serious risks for safety arise from deficiencies of functionality, reliability or usability as ISO/IEC 9126 standard [9] states.

The early detection of *Hazards* is a challenge during the specification of Safety requirements. A hazard is any potential source of damage or injury to an entity of the system, from an operator to an entity of the environment or the self system. In this way, during the gathering and specification of Safety requirements, the analyst has to identify the likelihood of the system hazards and analyze them so as to determine which strategy is the most appropriate for their management. For this analysis, the risks related to each hazard have to be established, i.e., the damage or injury to any entity.

These concepts indicate that although the process of gathering and identifying this kind of requirement is quite similar to that applied in other contexts, it does show some meaningful differences. Furthermore, the fact that tele operated systems are Safety Critical emphasizes how important it is to provide the analyst with a both specific process and specific notation.

A *Goal* describes why a system is being developed, or has been developed, from the point of view of the business, organization or the system itself. In order to identify it, both functional goals (expected services of the system) and non functional goals (quality of service, constraints on the design, etc.) should be determined. A *Goal Model* is built as a directed graph by means of a refinement from the systems goals (or concerns). This refinement lasts until goals have enough granularity and detail so as to be assigned to an agent (software or environment) so that they are verifiable within the system to be. This refinement process is performed by using AND/OR/XOR refinement relationships. In addition, operationalizations are also specified during the Goal Model definition. Operationalizations are the lowest level refinements introduced to describe the design alternatives associated to the requirements by means of contribution relationships.

This approach offers two advantages that make it especially appropriate for analyzing the specification. The first advantage is related to its ability to specify and manage positive and negative interactions among goals, which allow the analyst to reason about different design alternatives and to validate the Goal Model by means of its animation. The second advantage is related to its capabilities concerning traceability from low level details to high level goals (concerns), which make it especially suitable to bridge the gap between architectural and requirements models.

3. Identifying and Specifying Safety Requirements

Both ANSI/RIA R15.06 1999 standard and Douglass' and Lemos' approaches have been integrated to establish a process for identification and specification of Safety Requirements in Tele operated environments. The Goal Model of ATRIUM is the notation used during the process. This model exploits the standard ISO/IEC 9126 as a starting point to organize the requirements specification.

The established process entails several steps. The first ones (I II) are related to the behaviour specification of the system:

I. *To identify system operation modes*, according to Lemos' recommendations for control systems. These operation modes are specified as system goals and have a refinement relationship towards the characteristic ISO/IEC 9126 Suitability. Each goal is related to its child sub goals by means of AND/OR/XOR refinement relationships. The relationship to be applied depends on whether all sub goals, some of them or only one sub goal, respectively, have to be included in the system to satisfy the parent goal. This intentional refinement is applied over and over again until goals have enough granularities to allow the identification of tasks.

II. *To identify Tasks* (T_i) *associated to each operation mode*. In the Goal Model, these tasks are specified

IEEE COMPUTER SOCIETY

as requirements which have an AND/OR/XOR refinement relationship towards their goal parent. In addition, it provides an improved visibility of which hardware/software components of the system are involved in which operation modes because of their traceability to operationalizations, i.e., scenarios which describe how these components interact to fulfil a requirement.

Both the identification of operation modes and tasks provide the system behaviour specification. This identification must be performed before Safety requirements are identified, so that any likely injury or damage is caused when the system is performing these tasks. Once this view of the system has been established, Safety requirements are determined by using the ISO/IEC 9126 Safety category. Therefore, the following steps are added to the process:

III. *Determine Safety Goals of the system*. For each identified task T_i its capacity to cause any damage is evaluated. If this happens, a Safety goal, *Safe* (T_i), will be specified so as *to achieve safeguard* T_i. This implicitly means a composition relationship between a Safety requirement *Safe* (T_i) and a Functional requirement T_i at the task level.

IV. *Determine System Hazards*. For each safety goal *Safe* (T_i) its related hazards are identified and specified as a sub goal *Manage* (Hz_j). An AND/OR/XOR relationship is established between *Safe* (T_i) and its set of hazards to be managed. The relation to apply depends on whether the whole set of hazards, some of them or only one, respectively, have to be managed to safeguard the task T_i. We have to bear in mind that the same hazard can be specified as refinement of several Safety goals.

V. *Identify Risks* (R_k) *associated to each pair Safe* (T_i) *Manage* (Hz_j). The associated risk R_k has to be identified, so that the strategy appropriate for the management of Hz_j is selected. A Hazard can be related to several risks depending on which task is to be evaluated. Hence, according to the framework of traceability [13], the risk specification is represented by introducing a *rationale* associated to the refinement relationships between *Safe* (T_i) *Manage* (Hz_j). In addition, according to Douglass' evaluation of risks, this rationale also entails some necessary information for the risk specification: identification of causes (software, hardware, human, …) which can result in a hazardous situation, reaction required for its management and maximum deadlines of exposure, detection and tolerance.

VI. *Determine the Risk Reduction Category (RRC) to apply to each relation Safe* (T_i) *Manage* (Hz_j), taking into account its associated risks R_1 ... R_n. With this aim, the following three attributes must be evaluated:

a) *Severity*. Level of damage that an entity of the environment or the self system can suffer. The table which is provided by ANSI/RIA R15.06 1999 for evaluation has been modified so as to deal with the widest sense of the term, i.e., including not only damage to the health or the environment but also to the self system. Therefore, the meanings associated to both S1 and S2 categories are described now as follow:

S2	Serious injury to the operator requiring more than first aid. *Damage of a system component which is irreplaceable both in time and cost.*
S1	Serious injury to the operator only requiring first aid. *Damage of a system component which is replaceable both in time and cost.*

b) *Exposure.* Frequency of exposure to the hazard. ANSI/RIA R15.06 1999 defines two categories: *E2* as *frequent* and *E1* as *infrequent*.

c) *Avoidance*. Likelihood of avoiding the exposure to the hazard. ANSI/RIA R15.06 1999 defines two categories: *A2* as *not likely* and *A1* as *likely*.

ANSI/RIA R15.06 1999 describes eight combinations of these values, which are specified in Table 1 as Risk Reduction Categories (RRCs) along with the recommended actions to manage the Hazard. Hence, a evaluation of the values (Severity, Avoidance, Exposure) is carried out on each pair Safe (Ti) Manage (Hzj) and its risks Rk. According to the RRC an action has to be selected to eliminate, substitute, prevent, isolate or cease the Hzj. If different RRCs are applicable for the same pair Safe (Ti) Manage (Hzj), the most severe will be the selected one.

Once each pair *Safe* (T_i) *Manage* (Hz_j) has been dealt with along with its Strategy, it is mandatory to re evaluate the full set of RRC for the Goal Model as the set of Hazards could have changed. This procedure must be repeated until all the hazards are considered "tolerable", i.e., an acceptable risk level for the system. These hazards are to be considered residual risks of the system.

Table 1 Risk Reduction Categories

Exposure	Avoidance	Severity	RRC	Action on Hz
E2	A2	S2	R1	Eliminate/Substitute
		S1	R2C	Prevent/
	A1	S2	R2A	Cease
		S1	R3A	Isolate
E1	A2	S2	R2B	Prevent/Cease
		S1	R3A	Isolate
	A1	S2	R3B	
		S1	R4	Warning/Training/Protect

4. Case Study

This section illustrates how we have applied our proposal to gather the safety requirements of the industrial project EFTCoR. The aim of this project is to design a family of robots for performing ship hull maintenance operations such as coating removal, washing and re painting.

The identified robotic tele operation platform is integrated by different subsystems (illustrated in Figure 1). The Robotic Devices Control Unit (RDCU) integrates all the required functionality to manage the EFTCoR. Cleaning Tools and Positioning Systems, both Primary and Secondary, are the mechanical components of the EFTCoR. Its architectural definition is highly relevant because of the constraints that have to be satisfied in order to allow a safe behavior of the system.

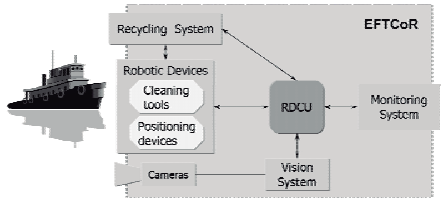


Figure 1. EFTCoR System

During the elaboration of the specification, 6 operation modes were detected (step I): *working*, *calibration*, *learning*, *diagnosis* and *configuration* (these four are *maintenance* modes) and *safe stop*. For each operation mode the involved subsystems and associated task were defined (step II). Due to limited space, in this work we only include the Working Mode and the Primary Positioning sub system of the EFTCoR system.

Once the operation modes of the system were established, their safety requirements were detected. With this aim, those tasks to be safeguarded were specified as Safety goals, *Safe* (T_i), of the system (step III). Figure 2 (whose symbols are described in Table 5), shows a part of the Goal Model where tasks (Table 2) to be safeguarded are identified. The Goal Model for the EFTCoR, partially shown in Figure 2, has been made by means of MetaEdit+[10], a metaCase for domain specific modelling. Each Safety Goal, *Safe* (T_i), was refined in *Manage* (Hz_j) sub goals by applying step IV. A summary of Hazards for the Primary Subsystem is shown in Table 3.

Table 2. Summary of Tasks to safeguard

Task	Description
T1	Motion of the arm joint
T2	Motion of the joint on tracks
T3	Stop of the primary motion

Table 3. Summary of Hazards for the Primary Subsystem

Task	Description
H1	Primary Positioning subsystem is moving and finds an obstacle on the rail.
H2	End of range of the Primary joint is overrun (arm).
H3	End of range of the Primary joint is overrun (on rails).
H4	Joint of Secondary touches the hull of ship.
H5	Primary (arms or tracks) does not stop.
H6	Robot places the tool point out of the cleaning stereo radio when the robot is in working mode.

IEEE COMPUTER SOCIETY

Applying step V, the risks (described in Table 4 and as comments in Figure 2) of each pair *Safe* (T_i) *Manage* (Hz_j) were established. Along with their identification, a description of other information is also required for the later evaluation activity. For instance, R8 and R10 were established as likely risks for the pair Safe (H6) Manage (T2) and, *Sofware Error of the robot control* as a possible cause of hazards; the realization of an *Emergency Stop* as a possible reaction when hazard H6 appears; probability of occurrence *medium*; etc.

Table 4. Summary of Risks for the Primary Subsystem

Risk	Description
R1	Damage of the tool or any mechanical component of the Secondary Positioning subsystem.
R2	Mechanical damage to the arm joint or joint on tracks.
R3	Damage to the hull surface.
R4	Damage to operator, primary or any object of the environment.
R5	Mechanical damage of the primary with joint on tracks and overturned of the robot.
R6	Mechanical damage to Primary Joint either on tracks or arm.
R7	Damages to Secondary.
R8	Damages to operator.
R9	Mechanical damages.
R10	Damage to objects of the environment.

Table 5. Explanation for symbols used in Figure 2

Symbol	Goals on Figure 1
Ti	Safe (Ti)
Hzj	Manage (Hzj)

As a result of step VI, several reduction categories were selected for use in each specific case. For instance, the evaluation of severity, exposure and avoidance was (S2, E2, A2) for the pair Safe (H6) Manage (T2), taking into account the risks (R8, R10). By applying the RRCs shown in Table 1, the category R1 was established and *eliminate* was the selected action. Once the hazard H6 is eliminated the re evaluation of the graph it is necessary in order to determine if the residual risks are tolerable for the system.

5. Safety Patterns For Operationalizations

Table 6 describes the strategy for the entry R2A of the safeguard selection matrix (Table 1). This recommendation is not enough to develop concrete systems such as EFTCoR and in consequence more detail is needed in order to facilitate this process. Thus, we propose to establish a catalogue of patterns for the reduction or elimination of risks, classified by the ANSI RRC. These patterns, in a similar way to the proposal by Gamma [6], describe the solution to recurrent situations. In order to facilitate its comprehension, we give a concrete example of a pattern considered in the context of the EFTCoR system.

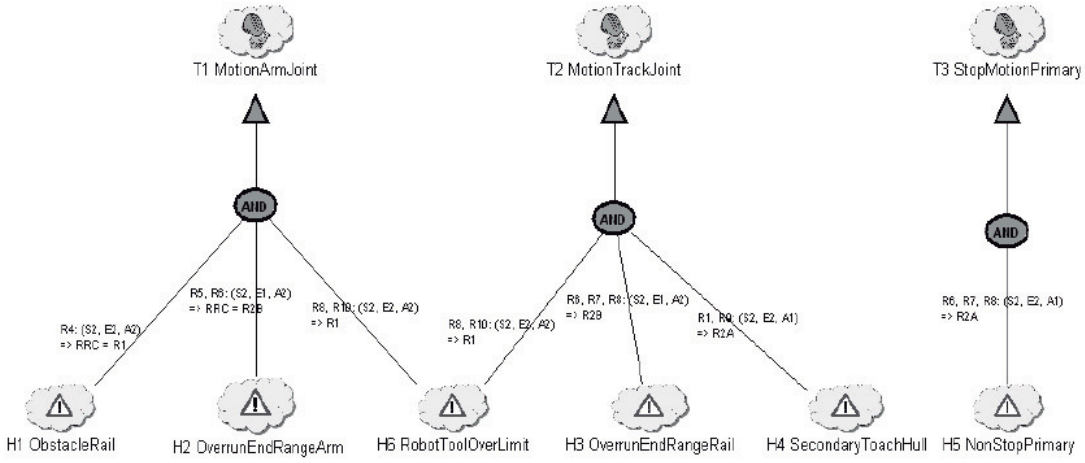


Figure 2. Goal Model (Part of)

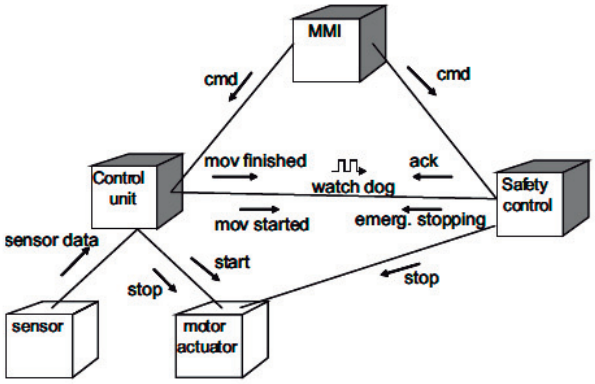


Figure 4. Deployment considering safety for hazard H5

Table 6. Description of the ANSI Risk Reduction Category R2A

R2A: Control reliable safety circuitry (based on hardware or software controller or firmware) The monitoring shall generate a stop signal if a fault is detected. A warning shall be provided if a hazard remains after cessation of motion Safe state shall be maintained until the fault is cleared. Common mode failures shall be taken into account. The single fault should be detected at time of failure.

The primary positioning system (see Figure 3) has a height of twelve meters and a weight of twenty tons which make inevitable the movement of the robot without the consideration of safety requirements. The crane has in its central zone an articulated arm of two tons with a secondary positioning system at its end (an XYZ table which includes a cleaning tool). It is indispensable that the system ensures a safe movement of the arm according to the received commands from the operator. A detailed analysis of the hazard H5 ("the arm of the primary system does not stop") leads us to associate the following sources of error:

- Any sensor integrated with the motors that move the arm fails.
- The electrical power is off.
- The control unit does not run correctly (a hardware fail or a software error).



Figure 3. Primary Positioning System with both arm joint (yellow) and joint on tracks (green) of the EFTCoR system

The hazard H5 may imply the breakage of mechanical parts, the precipitation of components to the floor or damages to the human operator (R6, R7, R8). Taking into account the severity of the injury, the frequency of the exposure and the probability of avoidance, the RRC is R2A. Figure 4 shows the deployment partitioning of the system (using an extension of the standard UML notation), which accomplishes the R2A risk reduction factor for the hazard H5. The limitation of space in this paper does not allow us to give all the details related to the real implementation of the safeguard for this hazard (for instance, the concurrent state machines which we have used to model the concurrent monitoring of the system, etc.). Even so, the following description gives a good idea of the way in which the hazard has been considered:

- When a movement command is received, the *Man Machine Interface* (MMI) node forwards it

simultaneously both to the *Control Unit* node (which will process it) and to the redundant node dedicated to monitor possible hazards (*Safety Control* node).

- The control node reads from a sensor the current position of the joint and controls directly the functioning of the motor. The safety control node will stop the motor when it detects a malfunction of the motor.
- Just before the execution of any command, the control node sends a message to the monitoring node authorizing the starting of the movement. From this instant, the control unit sends to the monitoring node, by means of pulses (*watch dog*), the current value just read from the sensor. The monitoring node answers this *tick* with an acknowledgement signal (*ack*) which includes as a parameter the estimated value of the motor position. Both nodes compute the curve of the discrete positions that must be reached by the robot arm. This timed discrete calculus is done by taking into consideration the initial value of the sensor and the command to be executed. Any difference between the calculated values implies an anomaly in the function of the robot movement. When a node detects a discrepancy in this value with respect to the estimation of the position values, an emergency signal is generated.

The different error conditions which could lead to a safe stopping of the robot in the previous example are the following: (1) Both the design and construction of the robot are done in such a way that, if a global fail of the system occurs then the robot will be mechanically fixed and returned to a safe mechanical state; (2) If any computing node does not work well (due to software or hardware errors) or the communication link fails then the other one will detect the discrepancy in the values. In this last case, it is essential that the control unit periodically reads the sensor data, although there is no current movement command in execution.

6. Related Works

One of the most popular approaches to identify, evaluate and manage safety requirements is the technique named *fault trees* [8]. These trees provide a graphical notation and a formal support that facilitate the analysis from the perspective of the system fails and its origins. However, they do not offer a global framework for requirement specification as a discipline. From the point of view of requirement refinement, our proposal is analogous to the use of fault trees; however, in our work the analysis of safety requirements is integrated and derived from the set of functional requirements of the system.

Letier et al [14] have proposed the use of KAOS for safety related requirement specification. They have introduced the concept of *obstacle* as *a set of non desirable behaviours*, the presence of those obstacles imply the obstruction in the fulfilment of the objective. At the same time, the negation of the obstacle generates the preconditions needed for the satisfaction of the requirements. The safety goals of the system being developed are formally specified by using temporal logic, and the obstacles (similar to hazards) are automatically obtained by the negation of the safety objectives and following the patterns given in their proposal [10]. However, the KAOS proposal does not provide a specific process for dealing with safety goals in the context of safety requirement specifications, nor does it consider factors such as severity, exposition time, etc, to be exploited during the analysis of the safety specification.

7. Conclusions

In tele operated systems the specification of safety requirements is a major challenge due to both the combination of hardware/software components and the presence of potential injury to people or equipment. In this work, we have introduced a unified proposal for requirement specification in this domain, which provides the integration and derivation of safety requirements with the remainder. We have adopted a goal oriented approach due to the obvious benefits for analysis and evaluation of alternatives. Moreover, the formal base of the approach allows the verification of several properties (not mentioned for space limitations).

The consideration of very relevant standards and practical approaches (ANSI, Douglass and Leveson) for the domain of safety requirements has been the source to establish some methodological guidelines for safety requirement specification of tele operated robotic systems. In these systems, it has been demonstrated that the use of catalogues and the perspective of traceability throughout the development process could improve the reuse of the knowledge based on a software product line approach [7]. In this paper, we have illustrated the application of these ideas in the scope of the EFTCoR research project.

Currently, we have two open issues: The first is related to the application of our proposal into other projects; and, the second is devoted to the extension of MetaEdit+ for the development of analysis tools, which help, for instance, to automatically determine the RRCs.

8. References

[1] ANSI/RIA R15.06 1999. *American National Standard for Industrial Robots and Robot Systems, Safety Requirements. Robotic Industries Association,* 1999.

[2] B. P. Douglass, *Real Time Design Patterns. Robust Scalable Architecture for Real Time Systems.* Reading, Addison Wesley. 2003.

[3] EFTCoR. Requisitos del Sistema. Anexo I: Resumen de Requisitos Funcionales de los sistemas robóticas. Subproyecto ANCLA, Abril 2004. http://dynamica.dsic.upv.es:8090/

[4] EFTCOR: Environmental Friendly and cost effective Technology for Coating Removal. (GROWTH G3RD CT 00794) EU Project within the 5th Framework Program, 2003.

[5] C. Fernández, J.A. Pastor, P. Sánchez, B. Álvarez, A. Iborra: "Co operative Robots for Hull Blasting in European Ship repair Industry. Robotics and Automation Magazine (RAM)", special issue on Industrial Robotics Applications, September 2005.

[6] E. Gamma, R. Helm, R. Johnson, J.Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison Wesley, 2003.

[7] J. van Gurp, J. Bosch and M. Svahnberg "On the notion of variability in software product lines", Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), 2001, pp 45 54.

[8] K.M. Hansen, A.P. Ravn, V. Stavridou, "From Safety analysis to software requirements", IEEE Transactions on Software Engineering, 24 (7), Jul 1998, 573 584.

[9] *ISO/IEC Standard 9126 1 SE Product Quality Part1: Quality Model*, ISO Co. Office, Geneva, June 2001.

[10] S. Kelly, K. Lyytinen, M. Rossi: "METAEDIT+ A fully configurable Multi User and Multi tool CASE and CAME Environment". Proceedings of 8th International Conference on Advances Information System Engineering, LNCS1080, Springer Verlag, 1996, 1 21.

[11] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal Oriented Requirements Engineering", IEEE Transactions on SE, Special Issue on Exception Handling, 26 (10), October 2000, 978 1005.

[12] R. Lemos, A. T. Saeed, "Analyzing Safety Requirements for Process Control Systems", IEEE Software, 12(3), 42 53, May 1995.

[13] P. Letelier, E. Navarro, V. Anaya, "Customizing Traceability in a Software Development Process", 13th International Conference on Information System Development, Vilnius, Lithuania, September, 2004.

[14] E. Letier and A. van Lamsweerde, "High Assurance Requires Goal Orientation", Proceedings of International Workshop on Requirements for High Assurance Systems, Essen, Septembre 2002.

[15] N. Leveson. *Safeware: System Safety & Computers*, Addison Wesley, 1995.

[16] E. Navarro, P. Letelier, I. Ramos, "Goals and Quality Characteristics: Separating Concerns", Early Aspects 2004: Aspect Oriented Requirements Engineering and Architecture Design Workshop, Oct. 25, 2004, Vancouver.

Designing Aspectual Architecture Views in Aspect-Oriented Software Development*

Rogelio Limón Cordero[1], Isidro Ramos Salavert[1], and José Torres-Jiménez[2]

[1] Department of Information Systems and Computation,
Polytechnic University of Valencia, Camino de Vera s/n E-46071 Valencia - Spain
{rlimon, iramos}@dsic.upv.es
[2] Computer Science Department, CENIDET,
Int. Internado Palmira s/n, col. Palmira. C.P. 62490, Cuernavaca, Morelos, México
jose.torres.jimenez@acm.org

Abstract. Aspect-Oriented Software Development (AOSD) is an area that is becoming important in software engineering. Currently it is focused on how to deal with aspects from the early phases of the software development process, in order to reduce the complexity produced by these aspects in these first phases. Software Architecture (SA) is one of the first steps in the software development process; the SA design requires a support framework to represent, identify, and manage aspects. In this paper, a method to represent and design SA is presented. The proposed method allows: (a) the detection and separation of the architectural aspects and concerns, and (b) the building of the architectural design of the modular and component-connector-aspect views.

Keywords: Sotware Architectures, Aspect Oriented Software Development.

1 Introduction

Aspect-Oriented Software Development (AOSD) is a novel paradigm that aims to improve the separation of concerns [1]. The concerns that (at the design or implementation phases) are spread over several modular units (crossing the limits of these units) are named *crosscutting-concerns*. This crosscutting complicates the development activities, the evolution, and the maintenance of the software. For this reason, a special modular unit that contains the *crosscutting-concerns* (known as *aspects*) was proposed in [1].

With regard to Software Architecture (SA), the aspects involved are called *early aspects* [2]. The identification of *early aspects* may ensure that appropriate decisions are made in the first stages of the software development. However, the techniques to identify and separate aspects are just beginning to be used in an extensive way (not always in an effective way) in the phases of requirement determination and design elaboration [2]. The handling of *aspects* at an architectural level still is in an immature phase.

* This research was supported in part by the **CICYT-PRISMA** project.

M. Gavrilova et al. (Eds.): ICCSA 2006, LNCS 3983, pp. 726–735, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Limón, R., Ramos, I., Torres, J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development. Proc. of the International Conference the Computational Science and Its Applications (ICCSA 2006) . 2006.

In this paper, a method to generate the aspectual architecture by means of a mechanism of separation of concerns and *early aspects* in a descending and iterative way is proposed. The method is supported by two proposed architectural meta-structures. These allow the representation of the architecture by means of the modular and component-connector-aspect views. With this method, it is possible to design the software architecture and at the same time, identify and specify the *early aspects* of the software development. The method uses the requirements and scenarios given by the users as inputs. The method starts with a tentative high-level architecture and performs a set of activities by breaking the architecture into its elements (functions and components) and establishing the elements' relationships. The aim of the method is to shape the aspectual software architecture by means of the identification and decomposition of the concerns and *aspects*. Furthermore, architectural patterns and a single tool are used, which makes the method more agile. The method is presented using a case study.

The rest of this paper is organized as follows. Section 2 introduces the basic architectural elements in the AOSD context and some related works. Section 3 presents the architectural meta-structures. Section 4 gives an overview of our method. Section 5 develops the method in an actual case study. Finally, section 6 presents some conclusions.

2 Software Architecture and Aspect Design

Software architecture is considered to be a set of system structures [3]. These structures are called views, which include software elements and their external visible properties and the interrelations among them through connectors. The architecture can either be represented in a schematic way or by means of some architectural description language (ADL) for analysis and evaluation proposals.

The architecture design method depends on the approach of the software development process that is followed. For example in the Rational Unified Process (RUP) an object-oriented approach is followed, and the architecture is designed and refined at every stage of the process; it is represented by means of an adaptation of the "4+1" view model.

Another approach proposed in [3] is based on the quality attributes that the software must fulfill; this method is called "Attribute Driven Design" (ADD). The ADD approach can be used with different software development methodologies (for example, the object-oriented and the component-oriented methodologies) where architectural aspects are still barely being considered.

In AOSD, the SA provides a twofold contribution: (a) to deal with early aspect by binding the ones that appear in the requirements with those that are manifested in the design, i.e. aspects of a phase that can reverberate in another phase; and (b) to focus the aspects of the SA taking in account the possible effects on other software development phases.

Thus, to be able to deal with the *aspects* at the architectural level, it is necessary to build a structure that allows for the representation and management of the *aspects*, starting with the separation of the concerns.

Proposals that deal with early aspects appear in [4], [5] and [6]. A method that deals with the software architecture to identify aspects using architectural tactics is proposed in [4]; nevertheless, this method does not deal with the component-connector view. A strategy for aspect decomposition in a top down fashion is proposed in [5]; however it does not deal with software architecture. A method called theme for aspect separation is proposed in [6] (the proposal uses UML notation); however the method does not discuss how to deal with software architecture.

The method proposed in this paper for designing the SA is intended to be part of the AOSD process. The method uses a top-down decomposition strategy and simultaneously makes the aspectual architectural views (in an iterative way), using the nota-tion of UML 2.0.

3 Aspectual Software Architectural Views Specification

We propose two meta-structures for dealing with aspectual architectural views, which are based on the architectural description model analyzed in [7]. The intention is to deal with the architectural aspects by means of the two views: the modular and the component-connector-aspect views, which are adaptations of the views analyzed in [2]. Figure 1(a) shows the meta-structures of these views. The first view is used to indicate the system static structure, and the second view expresses the system behavior, which is a realization (represented by the symbol $--->$) of the modular view.

The modular view contains the *AspArqMod*, *function*, and *aspect* stereotyped classes. The first stereotype is composed of the other two stereotyped classes, i.e. an aspectual architecture is "shaped" by the *function* and *aspect* classes, and their relationship is built by means of an association relation labeled as *Joint Points*.

Since the elements in the modular view are classes (modules) that describe the functional requirements, this kind of module can have its own relationships inside the view such as: decomposition (is a sub-module of), uses, layers, and generalizations [4]. The *aspects* are bound with the functional modules through a prototype relation called *joint-points*, whose cardinality indicates that there are functional modules that do not have a relationship with any *aspect*; but each *aspect* must have at least one relationship with one functional module.

The component-connector-aspect view (C-C-A) expresses dynamism at the component level and has three kinds of components: *functional*, *aspect*, and *connector*. The first two are precisely a realization of the classes with the same name as the modular view. The *connectors* are used to make interactions among the components, and interactions between components and their *aspects*. One of the strategies proposed in [8] is used to specify these components, by means of UML 2.0 notation. This strategy uses a *connector* component type to bind the components through their input (pIn) and output (pOut) ports. An extension of this strategy was required to incorporate *aspects*, which are represented by means of the *aspect* component class; these are bound to functions through *connectors*.

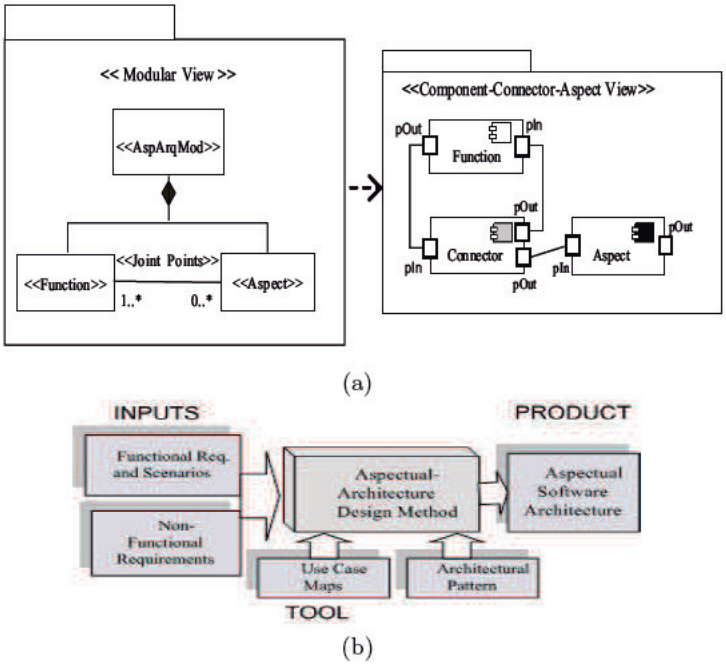


Fig. 1. (a) Meta-structures for aspectual architecture, the two proposed views. (b) The elements involved in the architectural design.

The *connector* is a realization of the association relationship of the modular view called *joint-points*. A *connector* can have as many ports as necessary (in accordance with its cardinality) and depending on how many components and aspects are attached by it.

4 An Overview of the Aspectual-Architecture Design Method

The method for aspectual-architecture design is based on three important features: (a) the use of an iterative top-down decomposition strategy, supported by the meta-structures of the aspectual architectural views described previously; (b) the use of architectural patterns; and (c) the use of some behavior analysis tools. The last two are adaptations of the proposals made in [3], [4] and [9].

Figure 1(b) shows the main elements that participate in the method. The inputs come from users and designers; the user contributes the functional requirements, and the designer contributes the non-functional requirements. In this paper, the requirements are expressed by means of use cases, and the scenarios are given in textual form.

A CASE tool based on maps is useful for finding *aspects*, specifying the module behavior, and understanding complex systems [9]. The use of architectural

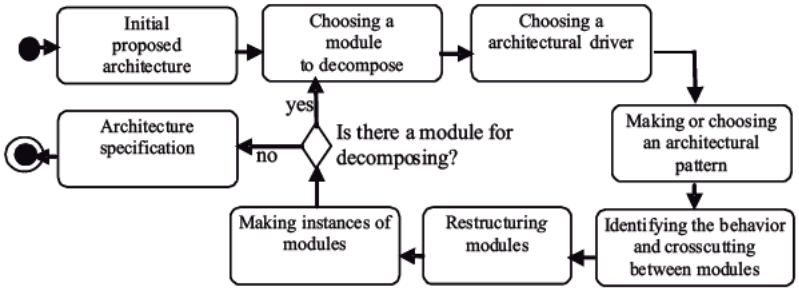


Fig. 2. The architecture design method activities

patterns makes the reuse of the architectures possible [3], which simplifies and speeds up the design process. There are some generic patterns, called architectural styles, such as pipe-line and filters, data flows, client-servers, and blackboards, which are the base of many systems.

The separation of concerns proposed in this paper is based on an iterative top-down decomposition process, since this kind of decomposition has proven to be effective in attacking the complexity at the problem-space level. This process allows the functional division of the system and the identification of aspects. The process enables the determination of hierarchical, association, and composition relationships among the functional modules. The joints between the modules and the aspects are also established. The final product of this process is the aspectual software architecture of the system, expressed through its modular and component-connector-aspect views, in UML 2.0 notation. Figure 2 shows the sequence of activities in the method; a description of each activity with an example will be presented in the next section.

5 The Aspectual-Architecture Design Method

5.1 Case Study

We intend to design a system of software for reservation and purchase of tickets in a bus terminal, which has several buses lines; each one of these has a route that passes through different cities.

A typical passenger can perform several functions: make his/her itinerary; choose the city of origin and destination; select a bus line; specify the date and time to travel; and select one or more seats, depending on the availability. All of these activities can be performed through the internet using a credit card or directly at a ticket window at the bus station. For simplicity's sake only one non-functional requirement will be considered; system security, which involves access to users. The user can be a passenger or an employee of the bus company. Security also implies the access to bank accounts to made queries and to apply charges on a credit card (when a passenger purchases a ticket).

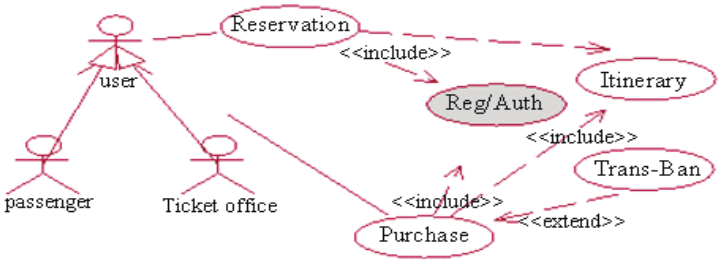


Fig. 3. Use cases for bus ticket reservation and purchase

Table 1. Scenarios Considered

Name	Scenario Description
Purchasing	make itinerary/register and authenticate user / make purchase / validate purchase/ reserve a bus seat/ print tickets
Tickets Reservation	make itinerary/register and authenticate user / make reservation

5.2 Method Inputs

Method inputs are constituted by the requirements and the scenarios; the inputs are given through use cases, as shown in Figure 3, which includes four functional requirements and one non-functional requirement. The figure 3 shows also two types of users, the passenger and the employee of the bus company. The scenarios are described in text form, (see Table 1). For reasons of simplicity only two scenarios are presented.

5.3 Method Activities

Initial proposed architecture. As a starting point, a possible architecture sketch is made, taking into account the specifications given in the functional requirements as well as the non-functional requirements. These requirements are taken at the highest level possible. The initial concerns for the architecture design are derived from these requirements. Figure 4(a) shows the concerns detected in the bus case study. These are only modules, which match a function-stereotype package level in UML.

Choosing a module to decompose. The first time that a module is chosen, any of the modules of the initial proposed architecture can be selected, since they belong to the packet level. In the subsequent times the selection must take into account the unsatisfied requirements with higher priority. In our example, any package can be chosen, since it is the first time.

Choosing an architectural driver. An architectural driver is considered in [2] as “a combination of requirements that shape the architecture or the particular

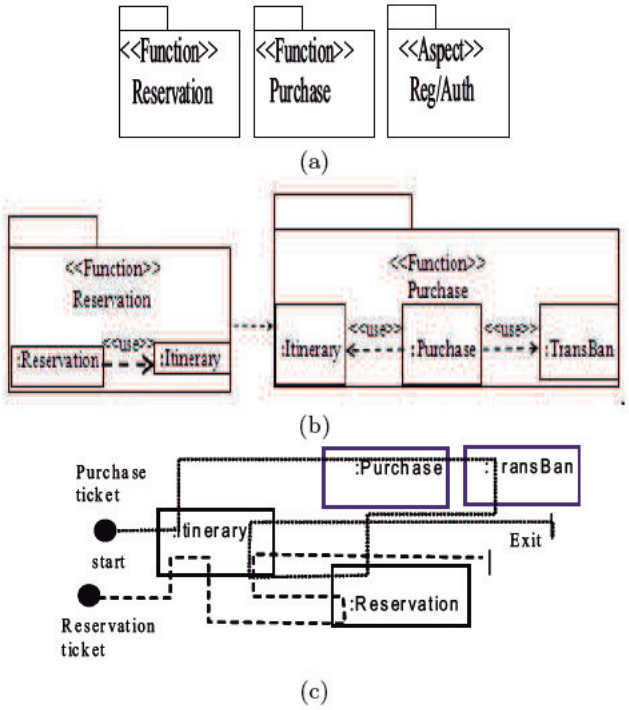


Fig. 4. (a) The initial proposed architecture. (b) Decomposition at the class level, and the application of an architectural pattern. (c) Applying use case maps to the ticket purchase and reserve scenarios.

module under consideration". In our example, we can consider the requirements associated to the purchase and reservation of a ticket, whose scenarios are shown in Table 1. Both of these requirements are used as architectural drivers, and have the same priority. Making or choosing an architectural pattern. An architectural pattern that corresponds or fits the expectation of the architectural driver must be found. This will be used as a base to refine the modules. For the bus example, the architectural drivers previously selected use a client-server pattern for the modules considered above, because the reservation and purchase modules (clients) use the services of the reg/Auth module (server). Next, the reservation and purchase modules are decomposed into sub-modules (classes). The first module will contain the reservation and itinerary classes, creating the ≪use≫ relationship between them. The second module will contain the itinerary, purchase, and transBan classes. The purchase class will use to the itinerary and trans-Ban classes, (≪use≫ relationship), as shown in Figure 4(b).

Identifying the behavior and crosscutting between modules. This task is performed through the use case maps, which use the scenarios as inputs and deal with the modules previously identified, i.e. at the class level. When using the use case maps, a crosscutting can be detected in the two scenarios considered,

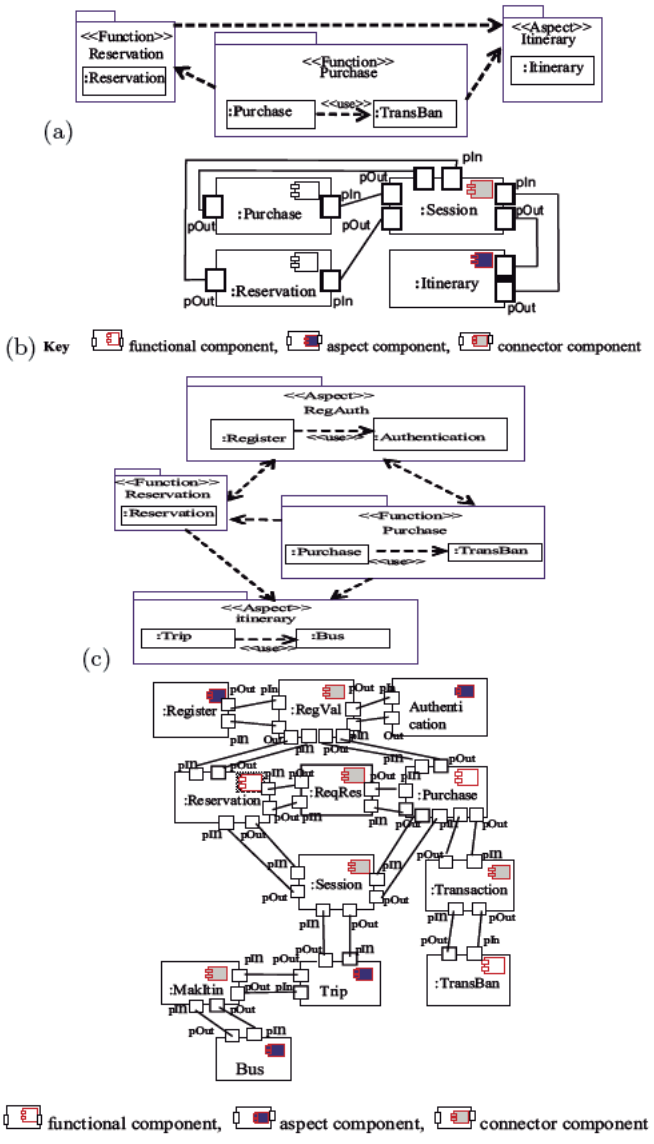


Fig. 5. (a) Modular view of the restructured architecture. (b) Component-connector-aspect view of the restructured architecture. (c) Modular view of the software architecture of the ticket purchase and reservation. (d) Component-connector-aspect view of the software architecture of the ticket purchase and reservation.

since the itinerary class is spread over two classes (*:Purchase* and *:Reservation*). The itinerary class is crossed by the two scenarios, and this class is linked with different classes of distinct modular units (Purchase and Reservation) so the

itinerary class becomes a *crosscutting concern*, i.e. an architectural aspect; as shown in Figure 4(c). Now the implicated modules will have to be restructured.

Restructuring modules. When the crosscutting inside a module has been found, it is necessary to change the classes that cause it, moving them to other modules. At the same time the relationship between them is modified. For example, the itinerary class is moved to ≪**aspect**≫ module, as shown in Figure 5(a).

Making instances of modules. In order to make instances for the functional modules, it is necessary to identify their relationships since these relationships must also be instanced. First, the instances of the functional module through components must be made. Then, each relationship must have an instance by means of a special component called a connector. For example, in the bus case, the attachment between the components is made by means of the session connector, as shown in Figure 5(b).

This process continues iterating the activities previously described above (except the first one), until all the modules are defined, as shown in Figure 5(c). Figure 5(c) shows that the modular view contains two functional stereotype packages: the purchase function and the reservation function. The purchase function depends on the reservation function. Both functions contain only atomic modular elements (classes), i.e. modules that can not be decomposed. The functional modules maintain a dependence relationship with the aspectual modules; the regAuth and itinerary *aspects* contain classes used by the functional packages. Figure 5(d) shows the modular view implementation. Each and every class in the modular view corresponds to a component in the Figure5(d). The Figure 5(d) clearly show the relationships between classes contained in the packages through a connector-component type. The information flow in this architectural view is graphically shown by the attachments among the ports (pIn and pOut) of the components. This indicates the dynamic nature of the software architecture.

Architecture specification. At this point, the architectural views have been designed and every module can be specified by means of an ADL. The PRISMA language [10] can be used because it has both a specification language as well as a configuration language; however, this topic is not covered in this paper.

6 Conclusions

This paper proposes a method to deal with *aspects* at the architecture level in the early phases of aspect-oriented development process. The output of the proposed approach is the aspectual-architecture design. The advantages of this approach can be highlighted by the reduction of the complexity inside the problem space, which is this result of the use of a top-down modular iterative decomposition strategy. This strategy separates concerns and *aspects* (*cross-cutting* concerns) in an easy, agile, and effective way, while at the same time composing both the modular view and the component-connector-aspect view, this achieves the aspectual-architecture design. The proposed meta-structures are the framework

that allows us to represent, identify, and manage aspects in the software architecture. They also make both a static analysis and a dynamic analysis of the architecture possible. Additionally, the method proposed contributes to: (a) making possible the use of patterns, which allow the architecture to be reused; (b) achieving the traceability between two view types, between the modular view and component-connector-aspect view, and (c) contributing to an improved beginning of the AOSD process.

References

1. R.E. Filman, T. Elrad, M. Aksit M., and S. Clarke. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
2. A. Rashid, P. Sawyer, A. Moreira, and J. Araújo. Early aspects: Aspect-oriented requirements engineering and architecture design. In *IEEE Joint International Conference on Requirements Engineering*, pages 199–203, 2002.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.
4. L. Bass, M. Klein, and L. Northrop. Identifying aspects using architectural reasoning. In *Proceedings Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop*, pages 51–57, Lancaster, 2004.
5. C.K. Chang and K. Tae-hyung. Distributed systems design using function-class decomposition with aspects. In *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems*, pages 148–153, 2004.
6. E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering (ICSE04) 0270-5257/04, IEEE*, pages 158–167, 2004.
7. IEEE Architecture Working Group. Ieee recommended practice for architectural de-scription of software-intensive systems. Technical report, IEEE, 2000.
8. J. Ivers and et. al. Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004, School of Computer Science, Carnegie Mellon University, 2004.
9. R.J.A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Trans-actions on Software Engineering*, 24(112):1131–1155, 1998.
10. J. Perez, I. Ramos, J. Jaén, and P. Letelier. Prisma: Towards quality, aspect oriented and dynamic soft-ware architectures. In *IEEE Proceedings of the Third International Conference On Quality Software (QSIC'03)*, pages 59–66, 2003.

Using Styles to Improve the Architectural Views Design*

Rogelio Limon Cordero, Isidro Ramos Salavert
Department of Information Systems and Computation, Polytechnic University of Valencia,
Camino de Vera s/n E-46071 Valencia – Spain
{rlimon,iramos}@dsic.upv.es

Abstract—**Software architecture has become more important in software engineering because it allows earlier decision making in systems design by defining the structures that will compose it. Each structure is in itself an architectural view that allows the isolation, analysis and modeling of different aspects of the system. The design of each view is based on the identification and specification of its elements and the relationships between them. This paper proposes a way to make a better architectural design through the management of the relationships present in a view type, using their styles as bases for the architectural structure, which leads to a more precise analysis in the design of architectural views.**

Keywords- Software Architecture; Views; Architetural Styles.

I. INTRODUCTION

Architectural view design implies identifying and specifying their components and establishing their relationships. This last task is perhaps one of the most important because the relationships constitute the basis of the architectural structure.

The design methods used in [1][2][3] establish architectural relationships by using a set of prototypical structures called architectural patterns as a reference [1], also known in other references as architectural styles. However, no analysis of these styles has been made in these methods nor have they been used in a systematic way in the design process.

The aim of this paper is research this area, by considering the architectural style as a relationship instead of a type of view. In addition, a proposal is presented to model the relationships in order to make the styles operative in the design phase of an architectural view.

This paper is structured as follows: Section II proposes a meta-model of the type of view component-connector in order to establish the relation between the styles and the other architectural elements. Section III describes, the fundamentals of how the proposed mechanism operates. Section IV proposes an adaptation of these fundamentals to the architectural styles. Section V shows how the architectural styles are applied in the design method, which it is illustrated in a case study. Finally, the conclusions are presented.

II. VIEW-TYPE AND ITS ARCHITECTURAL STYLES

From the definition presented in [1], the architectural view is considered as a specific perspective of each structure of the architecture, whose formation rules are contained in a viewpoint [4]. We will use the term view-type as the specification that defines the elements that an architectural view contains and the type of relations that can be established among them.

There are several approaches about what the architectural view should be; among the more relevant, we can enumerate the following: SEI[1], SIEMES[2], Kruchten [5]. Even though there are some differences among them about how they interpret a view, there are coincidences in the type of elements and relations that are considered in each one. In all of these approaches, the components and connectors are considered as basic building blocks in some of their views. For this reason, we have chosen the component-connector view-type [1]

* This research was supported in part by the CICYT-PRISMA project

Limón, R., Ramos, I. Using Styles to Improve the Architectural Views Design. Proc. Of the International Conference on Software Engineering Advances (ICSEA 2006). 2006.

as the most appropriate approach to apply to our proposal. In this approach the basic blocks (components and connectors) are considered in a more general way.

The relationships among the elements of a view-type adopt either some of the pre-established shapes known as architectural styles (or simply styles) [6] or similar shape. Styles define a set of features and constraints that typify the relationship among the architectural elements. We use the term basic style to denote a specific and well-delimited style. We use the term heterogeneous style to denote when the style is composed with two or more basic styles.

The Basic styles used in software architecture are studied in [6][7][10]. In [6] each style is associated to a specific type of view, which means that an architectural style cannot be inside of more than one type of view. This is derived from the fact that each type of view deals with different concerns. In this way, the styles associated to a type of view must be specified in some way, in the architectural design phase.

Fig. 1 shows the meta-model proposed for designing a component-connector view type (<<C-C-VT>>) specifying its relation with styles. This meta-model shows the elements and relationships that make up this type of view. The primary elements are <<Component>> and <<Connector>>, which are linked by their <<Ports>>. These ports can contain <<Interfaces>> to allow several modes of access.

Both the components and connectors are determined by the style types shown (<<Style>>) (Pipe&Filter, ClientServer, PeerToPeer, PublishSubscribe); more styles can be included. Note that the composition relationship in <<Component>> allows for the integration of self-contained style.

III. FUNDAMENTALS OF STYLE TREATMENT

Considering that an architectural style is a specific type of a relationship established between at least two architectural elements, a mechanism must be established to identify, represent and analyze such relationships in order to improve the architectural design process. In this paper we propose a strategy that creates a basis for constituting each style from the relationship between two elements.

Our approach is based on the design rules of [8], where a dependency matrix for analyzing relations among elements is proposed. This matrix has been used for the hierarchical and dependency relationships. In this paper, we will focus on the component-connector view-type in order to: a) model the communication relations among components, and b) use these previous relations that represent with matrices to obtain the architectonic model.

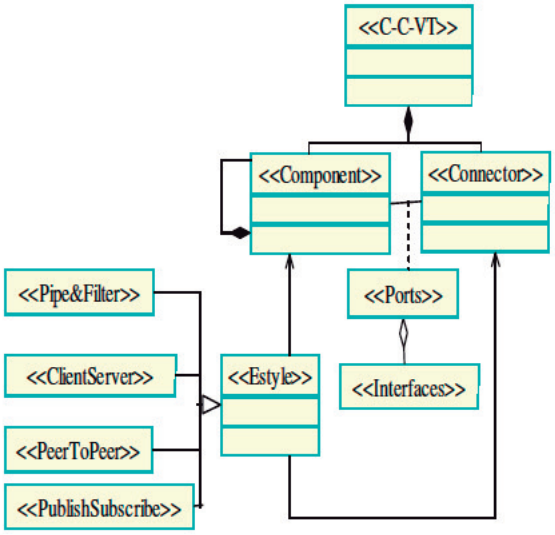


Figure 1. Meta-model for component-connector view.

A component can be related to another component in the following interactions: by means of procedure calls to invoke its services, to share information, and to send messages or information between the two components [7]. This concept is called as a communication relationship, where the connectors assume the role of coordinating and controlling the communication.

Let's consider that a relation between two components (named A and B) can be represented by a relationship matrix (see Fig. 2).

↓→	A	B
A	0	V_x
B	V_y	0

Figure 2. Relationship matrix between two components

In the relationship matrix, Vx, and Vy represents values that indicate the relation between their corresponding row and column elements. Each value will have a different interpretation for indicating the type of relation. The range of values will vary from 0 to R-1, where R is the cardinality of that set of different relationships that can be obtained. For instance, for R=3, the range of values for Vx, and Vy will be {0,1,2}; in this case, the value 0 indicates that there is not a relationship. Note that the matrix does not include self-relationships among the components.

0-7695-2703-5/06/$20.00 (c) IEEE

The interpretation of a relationship is as follows: starting from the first row, the element of each row (A or B) communicates something to the element of its corresponding column (A or B). The values Vx, and Vy in one cell are used for specifying the type of communication (for instance, the value 1 can represent a unidirectional communication, and the value 2 will represent a bi-directional one).

Fig. 3 shows three possible cases for R=2, whose values of Vx, and Vy will be {0,1}. In a general way, 0 means no communication, and 1 indicates that a communication exists.

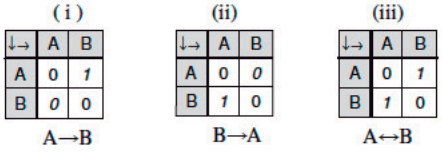


Figure 3. Cases of relations with R=2

For case (i), which is represented as A→B, the interpretation will be that A communicates something to B. Case (ii) is the reciprocal of the case (i), and case (iii) indicates that there is a communication from A to B, and another one from B to A (A↔B). Note that there exist only two types of relationships (i, iii).

When R=3, the relationships values will be {0,1,2}, where 1 and 2 will indicate a given type of communication, represented by (→) and (=>) respectively. Fig. 4, shows the possible cases, excluding the cases presented previously in Fig.2. Note that only five of the eight cases (Fig. 2 and Fig. 3) are different.

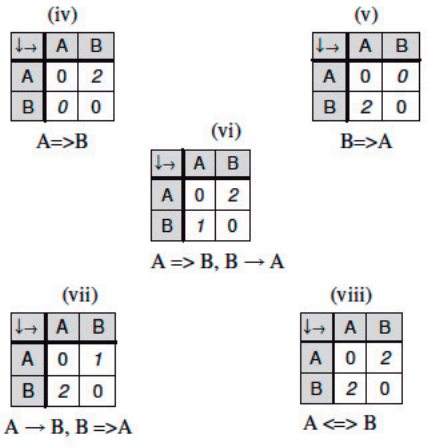


Figure 4. Cases of relations with R=3 (complement)

The following annotations complement this proposal: a) If R is the number associated to the relationship type, then $R^2 - R$ different communication cases among two elements can be generated. b) R will depend from the number of styles that will be considered in the type of view. Each matrix is used both, for analyzing and conforming relationships, and for keeping those relationships with evolution and maintenance aims.

Each matrix is used to analyze and constitute relationships, and to keep these relationships within the evolution and maintenance goals of the architecture. These matrixes will be adapted to be used in the architectonic styles.

IV. From the Matrix to the Architectural Style

The component-connector view shows the dynamism of software architecture, links of communication are established in the relationship between their elements for this reason. These links are typified by the architectural styles that belong to some type of view. Therefore, the styles set the standard of the element relationships, and the style determines the behavior of the elements.

Each style is associated to a relationship matrix, where its elements are the components and the cells of that relation are the connectors of the class that corresponds to its style. The architecture is represented in UML 2.0, using the proposal of [9]. The basic notation employed here is shown in Fig. 5.

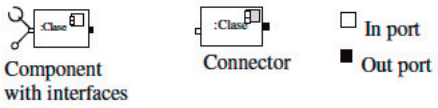


Figure 5 Basic notations for architectural representation.

The architectural styles included in Fig. 1 are represented by means of matrices and their corresponding UML representation, as it is shown in Fig. 6.

In this case, R=3, and therefore, the range of values is {0,1,2}, which it represents the following: 0 means no communication; 1 means a communication is sent; 2 means that a intercommunication is established (This communication can be: a message, data, a request, or a event).

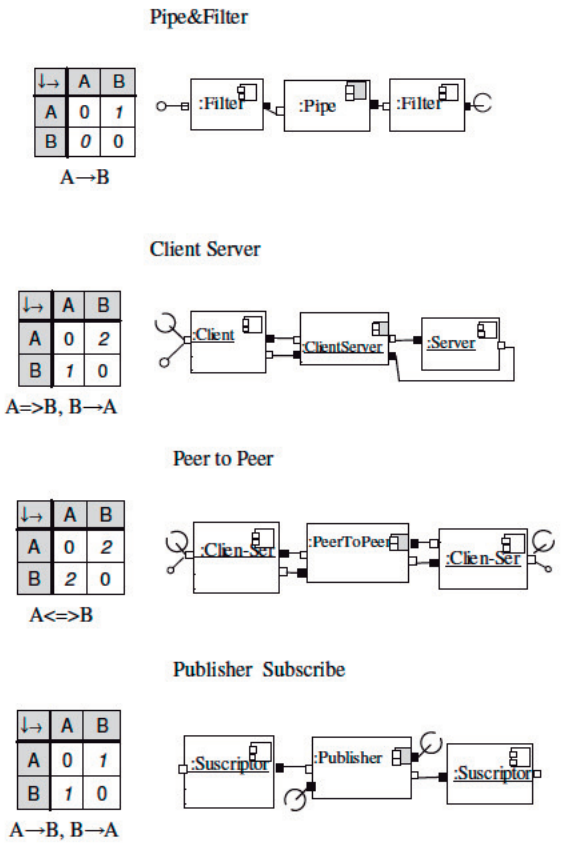


Figure 6. Matrixes for architectural basic styles and their corresponding UML representation

V. APPLYING THE STYLES IN THE DESIGN OF THE ARCHITECTURAL VIEW

A. *Using architectural styles in the view design method.*

Designing a view-type means applying a method to constitute its elements and the architectural relationships, which are expressed in UML 2.0 in this paper.

The design method proposed in [10] is used. The activities that are related to our proposal are shown in figure 7(a). Our attention is focused on: "Shaping architectural relationships" which is the key to improving the design process by means of style relationships.

The start point is the initial architecture derived from the functional requirements (expressed by the use cases) and the module to be decomposed (using the specification of each use case). The module contains some components, so the task now consists in finding the relation among the components that will constitute the architecture. The architectural relationships that are established with the architectural basic styles are applied. Their activities are detailed in Figure 7(b).

The first activity consists of identifying a pair of components that interact with each other by using the scenarios as reference. The interaction is recorded in the relationship matrix, using the corresponding style. In the case of a first time link of these elements, a basic style will be constituted; otherwise, it a heterogeneous

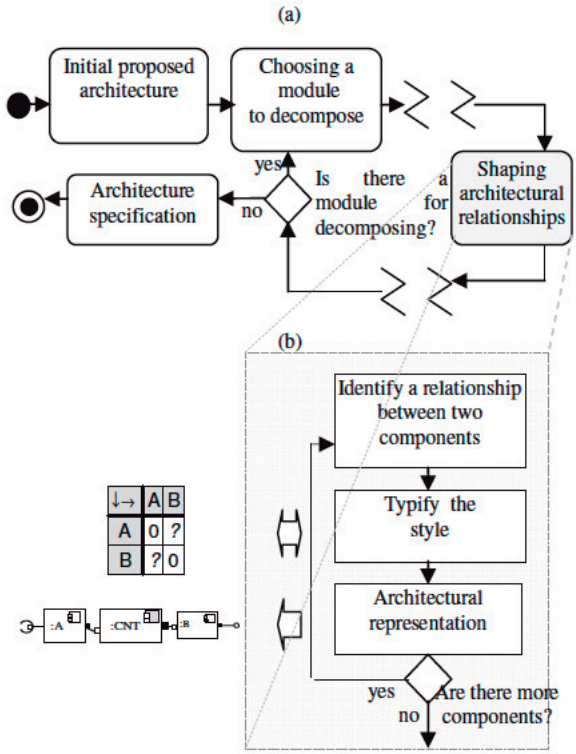


Figure. 7. (a) The architectural design method activities, (partially), (b) The activities to shape architectural

style will be made, which will imply a style reconstruction. Once that style is typified, the architectural representation is carried out.

The activities described above are repeated until there are no components left to relate. These activities are illustrated by means of a case study in the following section.

B. *Case study*

The case of study is an extension of the case presented in [10]. This study consists of designing an architecture for a purchase and reservation ticket

0-7695-2703-5/06/$20.00 (c) IEEE

system for any kind of passengers transport (bus, air plane, train, etc), where the passengers can buy and reserve a ticket, or just search in Internet. After introducing the query, the system will generate a tentative itinerary as a first step.

In order to focus on the use of the architecture styles proposed here, let's suppose the detection of components in a first architecture are derived from the use cases, which are shown in the Table 1.

TABLE 1. COMPONENTS DETECTED.

	Description	Component
1	Query formation	QuerForm
2	Generating itinerary alternatives	GeneItin
3	Making the reservation	Reservation
4	Purchasing a ticket	Purchase

The first identified interaction appears between QuerForm and GeneItin components, where the former transmits the information of a query to the latter, which also generates alternative routes. The relationship matrix is shown and its architectural representation are shown in Fig. 8.

↓→	QuerForm	GeneItin
QuerForm	0	*1*
GeneItin	*0*	0

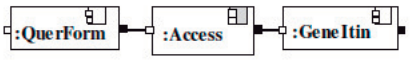


Figure 8. Matrix for the QuerForm and GeneItin components, and its architectural representation (Pipe&Filter Style)

Note that the architectural structure includes a connector of the "Access" class, which links the QuerForm and GeneItin components.

The following elements to relate are Reservation and Purchase, which communicates with the earlier described components (QuerForm and GenItin), generating the relationship matrix shown in Fig. 9 (a), and its corresponding architectural view of the ticket reservation-purchase system is shown in Fig. 9 (b).

(a)

		Reservation	Make-Itinerary		Purchase
			QuerForm	GeneItin	
Reservation		0	2	0	*0*
Make-Itinerary	QuerForm	*0*	*0*	*1*	0
	GeneItin	*1*	0	0	1
Purchase		*0*	2	0	0

(b)

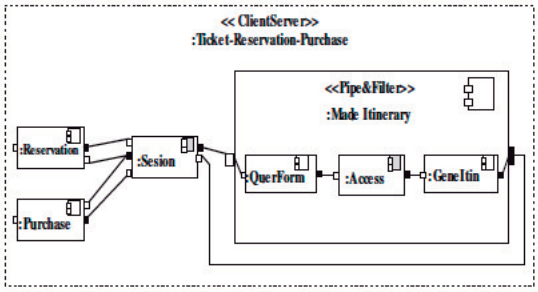


Figure 9. Ticket purchase-reservation system. (a) relationship matrix, (b) architectural view representation

VI. CONCLUSIONS

By modeling the architectural style as a type of relation, we have proposal a novel way to design an architectural view.

The identification and analysis of the basic relationships proposed here can be used in other view types, by making the analysis of relations in the architectonic design process operational.

The use of the relationship matrixes allows us to: a) adapt some view types where there is dynamism among the element relationships; b) make the maintenance of the architecture easier since the element relationships are recorded; c) generate a graphical architectural representation from the matrix (in UML 2.0 notation).

REFERENCES

[1] Bass L., Clements P., Kazman R., Software architecture in practice, Addison-Wesley, 2ª Ed. , ISBN 0-321-14595-9, 2003.

[2] Hofmeister, C., Robert Nord, and Dilip Soni. Applied Software Architecture. Boston, MA: Addison Wesley, 2000.

[3] Rozanski Nick, Woods Eoin, Software Systems Architecture, Working With Stakeholders Using Viewpoints and Perspectives, Addison-Wesley, ISBN 0-321-11229-6, 2005

[4] IEEE Architecture Working Group, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[5] Kruchten Philippe. The 4+1 View Model of Architecture, Paper published in IEEE Software Vol.12, No.6, pp. 42-50 Nov. 1995.

[6] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R. Stafford J. Documenting Software Architecture, Views and Beyond, Addison-Wesley, 2002.

[7] Mehta N. R., Medvidovic N., Phadke S., Towards a Taxonomy of Software Connectors, Proceedings of the International Conference on Software Engineering- ICSE'00 (2000).

[8] Baldwin, C.,Clark, K. Design Rules: Volume 1. The Power of Modularity. Cambridge, MA: The MIT Press. 2000.

[9] Ivers J., Clements P., Garlan D., Schmerl B., Silva J. R. O., Documenting Component and Connector Views with UML 2.0, School of Computer Science, Carnegie Mellon University, TECHNICAL REPORT CMU/SEI-2004

[10] Limon C. R., Ramos S.I. Designing Aspectual Architecture Views in Aspect-Oriented Software Development , Computational Science and Its Applications, LNCS 3983, pp, 726-735, 2006.

0-7695-2703-5/06/$20.00 (c) IEEE

Generic Architecture of an Expert System for Diagnosis

MA. EUGENIA CABELLO, CRISTÓBAL COSTA, ISIDRO RAMOS, JOSÉ A. CARSÍ
Department of Information Systems and Computer Science
Polytechnic University of Valencia
Camino de Vera s/n, 46022 Valencia
SPAIN
{mcabello, ccosta, iramos, pcarsi}@dsic.upv.es http://issi.dsic.upv.es

Abstract: This paper presents the construction of a model software (PIM following OMG standards) of an expert system for diagnostic tasks within the framework PRISMA. PRISMA allows the user to obtain a product line from this type of system, offering advantages that PRISMA architectures provide in the construction of distributed, complex, and re-usable systems. Our architectural model is described in two levels: the type level to define the types of the architectural elements that make up the model, and the configuration level to define the instances and specify the topology of the model. By means of the PRISMA Model Compiler, the model is transformed automatically to .NET executable code (PSM).

Key-Words: software architectures, aspects, architectural elements, diagnosis, reusability of software, expert systems

1 Introduction

Systems for diagnosis have recently become of great interest. They identify the internal causes that lead to problems starting from a series of observable data. This need has motivated the development of systems that perform tasks of this type in different domains such as medicine, law, electronics, meteorology and agriculture. At the present time, the importance of software systems has increased along with their complexity and, therefore, there is an increased need to improve them.

In order to improve the development and maintenance of diagnostic systems, a generic architectural model of an expert system has been constructed to perform diagnostic tasks, using the modelling framework PRISMA [1]. This model offers the advantages of distributed systems, facilitating reusability and decreasing complexity.

In the proposed method, once the different architectural elements and aspects that make up the system are identified, the topology of our architectural model is obtained. This model is instantiated, obtaining a specific topology for any kind of system that is desired. In this way, generic software applications for diagnostic systems can be obtained.

This model is automatically compiled into C# code and into its abstract specification in the PRISMA ADL by means of the PRISMA Model Compiler.

The article is divided into five sections. Section 2 introduces the PRISMA model and the architecture of the expert system; the correspondence between the two gives rise to our generic PRISMA architecture for expert systems. Section 3 describes the proposed architectural model. Section 4 presents the related works. Finally, section 5 presents our conclusions and future work.

2 Architectural foundations

The architectural conceptual model presented in this work has been designed within the framework of the PRISMA model, considering the architecture and operation of a rule-based expert system.

2.1 The PRISMA model

The PRISMA model integrates two approaches: the Component-Based Software Development approach (CBSD) [2] and the Aspect-Oriented Software Development approach (AOSD) [3]. This integration is obtained by defining the architectural elements through their aspects. The PRISMA model not only defines basic architectural elements and specifies their syntax and their semantic, it also specifies the aspects that cover the necessary properties of each element.

PRISMA makes the detection of the architectural elements using the functional decomposition criterion applied to software architectures [4]. This detection has resulted in architectural elements in different abstraction levels.

PRISMA defines the architectures in two levels: the type level and the configuration level. In the type level, the types of the architecture are defined: interfaces, aspects, components, connectors and

systems, all of which can be reused. In the configuration level, the types are instantiated and the topology of the model is specified.

2.2 The Expert System Architecture

Expert systems capture the knowledge of experts and try to imitate their reasoning process when they solve problems in a certain domain.
The knowledge base, the inference motor, the working memory or facts base, and the user interface, are the main components of the architecture of a rule-based expert system (see Figure 1).

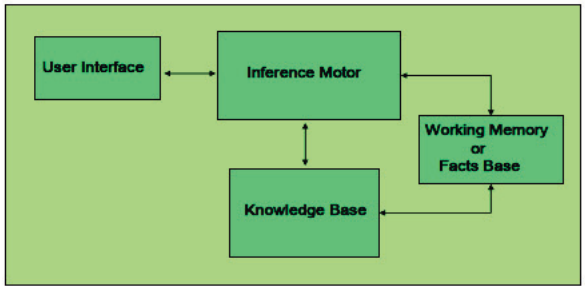


Fig. 1 Expert System architecture

These elements are independent and make up separates units. The data are grouped in the working memory (temporary warehouse of dynamic information). The representation of the knowledge uses rules of the type: IF <premise > THEN <conclusion>, which make up the knowledge base. The control is independent and is performed by the inference motor when making inference processes through reasoning strategies. The input and the output information of the system are made through the user interface.

2.3 The correspondence of the architectural elements between the PRISMA architecture and the Expert System architecture.

The design of the architectural elements of our PRISMA model for Expert Systems are based on the following criteria: the inference motor corresponds to the *Component_type InferenceMotor* of our architecture; the knowledge base and the working memory jointly correspond to the *Component_type KnowledgeBase;* and the user interface corresponds to the *Component_type User*.
Since our architecture is based on the PRISMA model, it was necessary to introduce two connectors (the *Connector_type InferenceKnowledgeConnect* and the *Connector_type UserShellConnect*) to connect the components that make up our model. The first connector *Connector_type InferenceKnowledge Connect* joins the *Component_type InferenceMotor* with the *Component_type KnowledgeBase*. The *Connector_type UserShellConnect*, the *Component_type InferenceMotor,* and the *Component_type KnowledgeBase* are encapsulated in a new architectural element called *System_type Shell*. This new element is connected to the *Component_type User* through the second connector *Connector_type UserShellConnect*.

3. The PRISMA Architectural Model for Expert Systems of Diagnosis

In this section, we present the architectural elements and the aspects in the type level. We also define the instances that make up the topology of the architectural model in the configuration level.

3.1 The definition of the types

In this section, we describe the architectural elements of the proposed model from a simpler functional grain (level 1) to a more complex grain (level 3). We also describe the aspects that are necessary to make up each architectural element.

3.1.1 The architectural elements

The simplest level of abstraction (level 1) consists of the *Component_Type InferenceMotor*, the *Component_Type KnowledgeBase,* and the *Connector_Type InferenceKnowledgeConnect*. The following level of abstraction (level 2) consists of the *System_Type Shell* (which encapsulates the three architectural elements of level 1), the *Component_Type User,* and the *Connector_Type UserShellConnect*. The greatest level of abstraction (level 3) consists of the *Architectural_Model Diagnostic*.
Component_Type InferenceMotor- This component establishes the control of the system and provides the general resolution strategy to obtain the diagnosis. The inference processes are carried out in this component through three reasoning strategies: deductive reasoning (process search led by the input data); inductive reasoning (process search led by the objective); and differential reasoning (process that discriminates between two or more possible solutions to determine which is the most reasonable). These are the strategies that are most widely used by the people when they make a diagnosis. To determine the order in which the diagnosis strategies will be executed, we considered the following process: first, the deductive reasoning is executed to make a diagnosis. If two or more diagnosis possibilities are detected, differential

reasoning will be executed. This will invoke inductive reasoning to reformulate questions to the user in order to determine which of the possibilities is the most accurate for the problem under analysis. This architectural element is independent of the domain knowledge of the case study, and therefore, the same *Component_Type InferenceMotor* can be the base of several systems in different domains. This component contains two ports: one port to communicate with the *Component_Type KnowledgeBase* (to acquire the domain data) and the other port to communicate with the *Component_Type User* (to obtain the values of the observable data). The behaviour of the *Component_Type InferenceMotor* integrates two aspects: the functional aspect and the distribution aspect.

Component_Type KnowledgeBase.- This component contains the domain knowledge of the case study by means of rules of inference (Horn clauses with head, which are modelled in PRISMA as derived attributes rules) and facts (Horn clauses without head, which are modelled as attributes). The facts represent the dynamic structure of the knowledge since these facts can be increased by relating the domain rules. This architectural element includes a temporary warehouse of dynamic information. This warehouse stores all the information offered by the user to the system (answers to formulated questions) and also stores the conclusions of the rules of the inference process in the form of facts. When the diagnostic process of a particular problem has concluded, the content of the working memory is eliminated. This component requests and sends services through a single port and defines its behaviour through the functional aspect and the distribution aspect.

Connector_Type InferenceKnowledgeConnect.- This connector synchronizes requests and sends the services requested by the components that are attached through its four ports. This connector has two aspects to implement this functionality: the coordination aspect and the distribution aspect.

System_Type Shell.- This is a component that encapsulates the diagnosis system, which will interact with the environment. This complex component is made up of the aggregation of the *Component_Type InferenceMotor,* the *Component_Type KnowledgeBase,* and the *Connector_Type InferenceKnowledgeConnect*. The set of services offered by this system is provided through its single port. The *System_Type Shell* does not contain emergent aspects.

Component_Type User- This component allows the communication between the user and the system. Through it, the user offers initial data to the system or responds to questions formulated by it. This component supplies a set of communication services through a single port and contains four aspects that define its behaviour: the functional aspect, the presentation aspect, the replication aspect, and the distribution aspect.

Connector_Type UserShellConnect.- This connector joins the *Component_Type User* and the *System_Type Shell* to perform the function of repeater between them; it simply forwards the requested services. It contains two ports to communicate the components that are attached. This connector defines the distribution aspect and the coordination aspect.

3.1.2 The aspects

The different interests (*crosscuting concerns*) of the system are separated into aspects, which increase the system maintenance. Ten aspects were necessary to define each one of the architectural elements of the designed software architecture. Some aspects have been used in more than one architectural element. This fact demonstrates the degree of reusability of the PRISMA approach.

Functional_Aspect FMotor.- The functional aspect of the *Component_Type InferenceMotor* defines the reasoning process of the system. It uses three reasoning strategies (deductive, inductive, and differential). By means of this aspect, the *Component_Type InferenceMotor* acquires data from the *Component_Type KnowledgeBase* to limit the possible options of the decision tree so that the diagnosis can be made.

Functional_Aspect FBase.- The functional aspect of the *Component_Type KnowledgeBase* defines the inference rules of the domain. In this aspect, the domain of the case study is specified. If the case study is changed, it will be necessary to replace this aspect by another *Functional_Aspect* that contains the information of the new domain. Thus, the approach presented in this work could be applied to diverse fields of knowledge by simply changing the functional aspect of the *Component_Type KnowledgeBase.*

Functional_Aspect FUser.- The functional aspect of the *Component_Type User* consists of the request for services that the system offers to the user, as well as the services required by the system to obtain the observable data.

Coordination_Aspect CUser.- The coordination aspect of the *Connector_Type UserShellConnect* defines the synchronization between the architectural elements that are joined by this connector.

Coordination_Aspect CDiag.- The coordination aspect of the *Connector_Type InferenceKnowledge*

Connect involves the synchronization of the services sent and received by the architectural elements that this connector coordinates.

Distribution_Aspect DLocation.- This is the distribution aspect of the *Component_Type InferenceMotor,* the *Component_Type KnowledgeBase,* and the *Connector_Type InferenceKnowledgeConnect.* This aspect indicates the location of the instances of the architectural element.

Distribution_Aspect DLocationMove.- This is the distribution aspect of the *Component_Type User* and the *Connector_Type UserShellConnect*. This aspect provides the information about the location of the component which is being used. Since the system can be managed from different machines, this aspect provides the services of mobility to move from one machine to another.

Replication_Aspect RUser.- The replication aspect of the *Component_Type User* allows the simultaneous access of different users, since there may be more than one user in the system.

Presentation_Aspect PUser.- The presentation aspect of the *Component_Type User* stores the set of elements that define the graphical interface that is used by the user.

To illustrate our work, we present a part of the conceptual model through three views of the *Component_Type InferenceMotor*. Figure 2 shows the graphical metaphor of this architectural element (with its aspects and interfaces) using Microsoft DSL Tools [5]. Figure 3 shows the specification in the PRISMA ADL of this component, and Figure 4 shows their corresponding C# code. Both were automatically generated through the PRISMA Model Compiler.

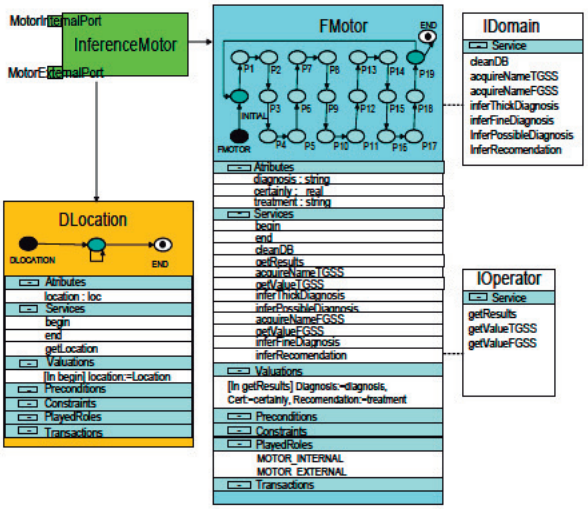


Fig. 2 Graphical metaphor of the Component_Type InferenceMotor

```
Component_Type InferenceMotor
   Ports
        MotorInternalPort: IDomain,
             Played_Role  FMotor.MOTOR_INTERNAL;
        MotorExternalPort: IOperator,
             Played_Role  FMotor.MOTOR_EXTRERNAL;
   End_Ports;

   Functional Aspect import  FMotor;
   Distribution Aspect import  DLocation;

   Initialize
         new ()
         {   FMotor.begin ();
             DLocation.begin (Location);
                  }
   End_Initialize;

   Destruction
         destroy ()
         {   FMotor.end ();
             DLocation.end ();
                  }
   End_Destruction;
End_Component_Type  InferenceMotor;
```

Fig. 3. Specification (ADL PRISMA) of the Component_Type InferenceMotor

```
namespace IM
{
  [Serializable]
  public class InferenceMotor: ComponentBase
  {
    public class InferenceMotor (string name, loc Location) : base (name)
    {
      AddAspect (new FMotor ( ) );
      AddAspect ( new DLocation (Location) );

      InPorts.Add ("MotorInternalPort", "IDomain",
"MOTOR_INTERNAL");
      OutPorts.Add ("MotorInternalPort", "IDomain",
"MOTOR_INTERNAL");
      InPorts.Add ("MotorExternalPort", "IOperator",
"MOTOR_EXTERNAL");
      OutPorts.Add ("MotorExternalPort", "IOperator",
"MOTOR_EXTERNAL");
    }
  }
}
```

Fig. 4 Code (C#) of the Component_Type InferenceMotor

3.2 Instantiation and configuration of the model

Once the types of the architectural elements and aspects have been defined, the following step is to define the topology. The instances in the topology of our model are shown in Figure 5 and their correspondence to the architectural elements types are shown in Table 1.

Architectural element type	Instance
Component_Type InferenceMotor	Control
Component_Type KnowledgeBase	Domain
Connector_Type InferenceKnowledgeConnect	DiagCnct
System_Type Shell	DiagnosticSys
Component_Type User	Operator
Connector_Type UserShellConnect	OperatorCnct

Table 1 Instantiation of the architectural elements

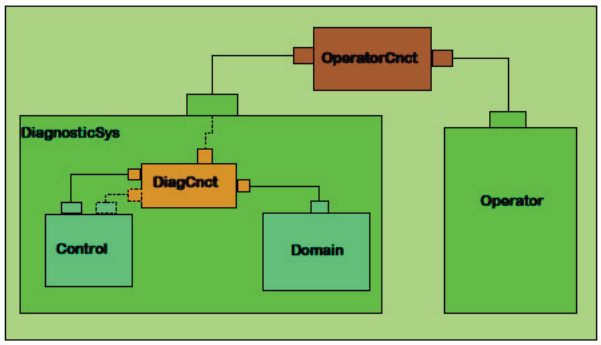


Fig. 5 The PRISMA architecture of the Expert System

Diagnostic System Architectural Model

The syntax of the specification of our architectural model is shown in Figure 6. This specification defines the PRISMA ADL in the configuration level.

```
Architectural_Model Diagnostic
 Variables
   VarOperator: User;
   VarOperatorCnct: UserShellConnect;
   VarDiagnosticSys: Shell;
 End_Variables;
Attachments
 VarDiagnosticSys.DiagnosticalPort                    ↔
      VarOperatorCnct.DiagnosticalPortCnct;
 VarOperatorCnct.UserPortCnct                         ↔
      VarOperator.UserPort;
End_Attachments;
 Initialize
  new (Location:loc, LocMin:loc, LocMax:loc, NewLocation:loc)
  {
 VarOperator = new User (Location,LocMin,LocMax,NewLocation);
 VarOperatorCnct=new UserShellConnect (Location,LocMin,LocMax);
 VarDiagnosticSys = new Shell (Location);
  }
 End_Initialize;
 Destruction
 destroy ( )
 {
  VarOperator.destroy ( );
  VarOperatorCnct.destroy ( );
  VarDiagnosticSys.destroy ( );
 }
 End_Destruction;
End_Architectural_Model Diagnostic
```

Fig. 6 Specification of the architectural model

This specification allows the definition of the architectural model as a type. When the model is instantiated, the required initial values are given. To reuse the architectural model, a specific topology for any system that is to be executed can be obtained by giving different initial values. An example of such an instantiation is shown below:

DIAG = **new** Diagnostic (localhost, 0, 100, newLocalhost),

where DIAG is the instance name of the *Architectural_Model Diagnostic*, and *localhost*, 0, 100, and *newLocalhost* are the instances of the arguments *Location*, *LocMin*, *LocMax*, and *NewLocation*, respectively. These values correspond to the following: the location of the machine that contains the component, the range of address mobility of the component, and the new location where the component can be replicated.

4 State of the art

There are a great variety of works related to the development of expert systems. The methodologies and applications for these systems have produced a wide variety of research products, offering suggestions and solutions to these systems in specific domains.

A study made by Liao [6] examines the methodologies of expert systems and classifies them into eleven categories. Two of these categories correspond to the systems based on rules and the systems based on knowledge. These categories have been taken into account in our work when using knowledge represented in the form of rules (Horn clauses) and facts (observable variables).

Liao also mentions that the development of expert systems has been characterized by the separation of knowledge and procedure as independent units. In our architectural model, the elements in the type level are defined taking into account this concept, specifically when there is a component that contains the domain knowledge and another component that executes the inference process of the diagnosis.

Giarratano and Riley [7] and other authors in the field of expert systems state that the architectures of these systems are based only on components. The architecture of our system has integrated two approaches combining both components and aspects. This increases the reusability and the maintenance of the system.

Expert systems have also been implemented in the development of different programming paradigms such as structured, logic, and object-oriented paradigms. These paradigms are oriented toward fourth-generation languages and visual programming methods to provide user-friendly communication. PRISMA provides an Architecture Description Language to define an architectural model that follows the MDA approach [8], which allows the automatic generation of code.

5 Conclusions and future work

This work has presented a component-based and aspect-oriented generic architectural model of an expert system for diagnostic tasks. The architectural elements that make up the model have been described in three levels of abstraction (from the simplest to the most complex). The diverse aspects (coordination,

functional, distribution, replication, and presentation) have been detected. Three reasoning strategies (deductive, inductive, and differential) have been used.

It is important to note that the PRISMA model has been applied to other domains, such as tele-operation systems for the cleaning of ships by means of an arm-robot [9], as well as in systems for handling bank accounts [10]. These cases have verified the improvement of the development and maintenance of these systems through the combined approach of aspects and components.

This system architecture conserves the characteristics and the performance similar to the other diagnostic expert systems. Nevertheless, the approach presented in this work offers the advantages that PRISMA architectures provide in the construction of distributed, complex and re-usable systems. These advantages are described below:

- The complexity of the system is managed by means of different levels of granularity through the separation of functional units in architectural elements
- The architectural elements of this model are independent so that components and/or connectors, can be added or interchanged without affecting the general operation of the system or the rest of the elements.
- The *crosscuting concerns*, which are separated in aspects, help the maintenance of the system.
- The components and the aspects can be reused in the same architecture in different domains or in different architectures.
- The architecture can be used in both centralized and distributed systems. The distribution aspect specifies the characteristics that define the dynamic location of the architectural element, as well as the mobility services to implement strategies of distribution of the architectural elements.
- The architectural model is independent of the platform and is designed from the perspective of the problem and not from the solution. This consideration allows diagnostic tasks to be applied to different domains.

To obtain the validation of the proposed model using the PRISMA Model Compiler, we have automatically generated its specification (in the PRISMA ADL) and its code (in C#), which allows this model to be executed on the middleware PRISMANET [11].

As future work, we plan to add several Knowledge-Bases (distributed knowledge) to the model. We are also studying the incorporation of a decision aspect, and other reasoning strategies for the diagnostic process. Finally, we plan to use the system in other types of similar tasks, such as interpretation and prediction.

References:

[1] J. Pérez, N. Ali, J.A.Carsí and I. Ramos, "Dynamic Evolution in Aspect-Oriented Architectural Models", *Second European Workshop on Software Architecture*, in Lecture Notes in Computer Science, Springer Verlag, Vol. 3527, Pisa, Italy, 2005.

[2] C. Szyperski, *Component software: beyond object-oriented programming*, ACM Press and Addison Wesley, New York, USA, 1998.

[3] Aspect-Oriented Software Development, in: http://aosd.net.

[4] F. Bachman, L. Bass, G. Chastek, P. Donohoe and F. Peruzzi, "The Architecture Based Design Method", *Technical Report CMU/SEI-2000-TR-001*, Carnegie Mellon University, Pittsburgh, USA, 2000, 61 p.

[5] R. Cabedo, J. Pérez, J.A. Carsí e I. Ramos, "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", *IV Workshop DYNAMICA*, Archena, Murcia, España, 2005, pp. 79-86.

[6] S.H. Liao, "Expert Systems Methodologies and Applications- a Decade Review from 1995-2004", in *Expert Systems with Applications*, Vol. 28, Issue 1, 2005, pp. 93-103.

[7] J. Giarratano and G. Riley, *Expert Systems: design and development*, Course Technology, 4a. ed., 2004.

[8] Model Driven Architecture, in: http://www.omg.org/mda.

[9] R. Cabedo, J. Pérez e I. Ramos, "Aplicación del lenguaje de descripción de arquitecturas PRISMA a un sistema robótico industrial", *Informe técnico DSIC-II/12/05*, Universidad Politécnica de Valencia, Valencia, España, 2005, 201 p.

[10] J. Pérez, N. Ali, J.A.Carsí, I. Ramos and E. Navarro, "Designing Software Architectures with an Aspect-Oriented Language", in *Advice: Journal of Aspect Orientation,* Vol. 1, No. 1, 2004, pp. 1-18.

[11] C. Costa, J. Pérez, N. Ali, J.A.Carsí e I. Ramos, "PRISMANET Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", *X Jornadas de Ingeniería del Software y Bases de Datos*, Granada, España, 2005, pp. 27-34.

Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico

María Eugenia Cabello Espinosa[1], Cristóbal Costa Soria[2], Isidro Ramos Salavert[3].

Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, España.

{mcabello[1], ccosta[2], iramos[3]} @dsic.upv.es

Resumen: Este trabajo ofrece una aportación al desarrollo de los sistemas expertos que realizan tareas de diagnóstico, mediante la construcción de una arquitectura software altamente reutilizable, orientada a aspectos y con comportamiento multirazonamiento. Se definen los distintos elementos arquitectónicos que conforman el modelo, de un gránulo funcional más simple al más complejo, y se especifica la topología del modelo.

Palabras clave: arquitecturas software, diseño orientado a aspectos, reutilización de software, sistemas expertos multirazonamiento, diagnóstico.

1. Introducción

Es notable el interés que han cobrado los sistemas expertos en los últimos años. Dentro de las diversas tipologías de estos sistemas destaca la del diagnóstico, que consiste en identificar las causas internas que provocan un problema o disfunción a partir de una serie de datos (variables observables). Han sido muchos los dominios de interés que han servido de escenario para el desarrollo y la aplicación de los sistemas de diagnóstico, entre los que se encuentran: agricultura, derecho, electrónica y medicina

Al respecto, es conveniente resaltar que este tipo de aplicaciones han ido adquiriendo una mayor importancia a la vez que una mayor complejidad, y por consiguiente, existe la necesidad de mejorarlas. Por ello, en este trabajo se pretende mejorar el desarrollo de los sistemas expertos de diagnóstico de dos formas. Por un lado, incorporando múltiples estrategias de razonamiento, para que ante la solución de un problema, el experto siempre aplique varias estrategias de razonamiento que le permitan resolver el problema de forma más eficiente. Por otro lado, proporcionando las ventajas de los sistemas distribuidos, facilitando la reutilización y gestionando la complejidad; en nuestro caso, mediante el modelo PRISMA [9], mejorando su desarrollo y mantenimiento.

Con este enfoque se ha construido un modelo arquitectónico de un sistema experto multirazonamiento de diagnóstico, para lo cual, una vez identificados los aspectos, se ha procedido a la especificación de los distintos elementos arquitectónicos del sistema.

Como resultado del proceso de análisis aplicado al diagnóstico, se han detectado diversos tipos de elementos arquitectónicos, permitiendo especificar la topología del modelo propuesto. Con ello, dicho modelo arquitectónico, podrá ser instanciado, con el fin de obtener una topología específica para cada sistema que se quiera ejecutar

Cabello, M. E., Ramos, I., Costa, C. Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico. Actas del XIX Congreso Nacional y V Congreso Internacional de Informática y Computación del ANIEI-CIICC´07. 2006.

El artículo está estructurado en 5 apartados: en el apartado 2 se presenta brevemente el estado del arte relacionado con los sistemas expertos multirazonamiento. En el apartado 3, se introduce el modelo PRISMA y se realiza una descripción del modelo arquitectónico propuesto. En el apartado 4 se comentan los resultados obtenidos. Por último, en el apartado 5, se presentan las conclusiones y trabajos futuros.

2 Estado del arte

Existen una gran variedad de trabajos relacionados con el desarrollo de los sistemas expertos. En cuanto a las metodologías y aplicaciones de estos sistemas, existe una amplia categoría de productos de investigación, ofreciendo sugerencias y soluciones a dichos sistemas en dominios específicos. Dichas metodologías y aplicaciones están diversificadas según los antecedentes de los autores, su experiencia, sus intereses de investigación, sus habilidades en la metodología utilizada y el dominio del problema.

Un estudio realizado por Liao [7] examina y contempla las metodologías de los sistemas expertos, clasificándolos en once categorías. Dos de estas categorías corresponden a los sistemas basados en reglas y los sistemas basados en el conocimiento, las cuales han sido contempladas en nuestra propuesta, al utilizar conocimiento representado en forma de reglas (cláusulas de Horn) y hechos (variables observables).

Por otro lado, como lo mencionan varios autores de la temática de los sistemas expertos, entre ellos Giarratano y Riley [6], las arquitecturas de dichos sistemas se ha realizado con una estructura basada en componentes. Sin embargo, la arquitectura de nuestro sistema se ha realizado con una estructura basada en componentes considerando adicionalmente la orientación a aspectos.

Asimismo Liao [7] comenta que el desarrollo de los sistemas expertos ha sido caracterizado por la separación del conocimiento y el procedimiento como componentes independientes. En nuestro modelo se definen elementos arquitectónicos en el nivel de tipos, contemplando esta separación al considerar un componente que contiene el conocimiento del dominio y otro componente que realiza el proceso de inferencia del diagnóstico.

En cuanto a la codificación de los sistemas expertos, la literatura muestra que han sido implementados en diversos paradigmas de programación como lo es la estructurada, la lógica, la orientada a objetos, etc., existiendo una tendencia en su desarrollo hacia lenguajes de cuarta generación y métodos de programación visual para dar un ambiente y una estructura amigable de comunicación con el usuario. PRISMA [9] proporciona un Lenguaje de Descripción de Arquitecturas (LDA) para definir modelos arquitectónicos, que siguiendo el enfoque de MDA [12], permite la generación automática de código.

En cuanto a los sistemas expertos multirazonamiento, las investigaciones realizadas por Cabello [4] muestran que éstos pueden ofrecer diferentes características. Se han construido sistemas con una o varias bases de conocimiento, en uno o varios nodos, con uno o varios sistemas de control, con una o varias estrategias de razonamiento, etc. Otra característica importante es cómo se realiza el proceso de razonamiento; éste puede tomar una tarea y dividirla en subtareas, distribuyéndolas entre los diferentes elementos arquitectónicos. Nuestro modelo contempla los razonamientos tanto de

forma centralizada como distribuida, dado que esta característica se obtiene con el aspecto de distribución.

3 Metodología

El modelo arquitectónico propuesto en este trabajo, está diseñado en el marco del modelo PRISMA [9], el cual integra dos aproximaciones: el Desarrollo de Software Basado en Componentes (DSBC) [10] y el Desarrollo de Software Orientado a Aspectos (DSOA) [11]. Esta integración se consigue definiendo los elementos arquitectónicos mediante aspectos: coordinación, funcional, persistencia, distribución, presentación, etc. De esta forma, el modelo PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las propiedades necesarias de cada uno de ellos.

Se realiza la detección de los elementos arquitectónicos mediante el criterio de descomposición funcional, que ha sido aplicado a las arquitecturas software [1]. Dicha detección ha dado como resultado elementos arquitectónicos en diferentes niveles de abstracción, que van de un gránulo funcional más simple al más complejo.

PRISMA especifica la arquitectura en dos niveles: el de definición de tipos y el de configuración. En el nivel de tipos se definen los tipos de la arquitectura: interfaces, aspectos, componentes, conectores y sistemas, los cuales son guardados en una librería para su reutilización. A nivel de configuración se especifica la topología del modelo.

El sistema utiliza tres estrategias de razonamiento: el deductivo, el inductivo y el diferencial, que serán proporcionadas a través de tres COTS (Commercial Off-The-Shelf), provenientes de diferentes plataformas de desarrollo, las cuales se integrarán en un componente PRISMA [3]. El sistema, a través de un conector, realiza la decisión de qué estrategia de razonamiento será aplicada en el proceso del diagnóstico. Dicho conector es independiente de la implementación de cada COTS. Debido a que cada una de las COTS exporta distintas interfaces, es conveniente utilizar un elemento arquitectónico que sirva de intermediario, este es el conector denominado Inferencia. Este conector aplica funciones de transformación, tal que: $I_i' = f(I_i)$, con i=1,2,3, donde Ii es la interfaz que define los servicios de comunicación que envía y recibe el conector Inferencia y que es independiente de plataforma, I_i' es la interfaz que define los servicios de comunicación que envía y recibe el componente Deductivo, Inductivo y Diferencial, respectivamente, resultante de aplicar la función f de transformación dependiente de la implementación del COTS.

3.1 Aspectos.- Los diferentes tipos de aspectos necesarios para la definición de cada uno de los elementos arquitectónicos de la arquitectura software diseñada son:
Coordinación (que define las sincronizaciones entre los elementos arquitectónicos que coordina), **Funcional** (que captura la semántica del sistema, mediante la definición de su comportamiento), **Presentación** (que suministra la metáfora visual que define la interfaz con la que trabaja el usuario), **Distribución** (que indica la ubicación de las instancias del elemento arquitectónico, así como los servicios de movilidad para desplazarse a otra máquina, ya que se puede gestionar el sistema desde distintas máquinas), **Replicación** (que permite el acceso simultáneo de diferentes usuarios, ya que es posible que haya más de un usuario utilizando el sistema) e **Integración** (que encapsula a una COTS).

3.2 Elementos arquitectónicos.- Los elementos arquitectónicos que conforman el sistema, de un nivel de abstracción más simple (nivel 1) al más complejo (nivel 4), se describen a continuación:

3.2.1 Nivel de abstracción 1:

3.2.1.1 Componente Deductivo, Componente Inductivo y Componente Diferencial.- realizan el proceso de inferencia aplicando la estrategia de razonamiento deductiva, inductiva y diferencial, respectivamente. Estos componentes establecen su comunicación recibiendo y enviando información, a través de su único puerto. Cada uno de los componentes importan dos aspectos: integración y distribución.

3.2.1.2 Conector Deductivo, Conector Inductivo y Conector Diferencial.- estos conectores, que cuentan con dos puertos para unir al conector Inferencia con los componentes Deductivo, Inductivo y Diferencial, respectivamente, hacen la función de adaptador elementos arquitectónicos que une, reenviando los servicios solicitados, aplicando una función de transformación. Cada conector importa dos aspectos: coordinación y distribución.

3.2.1.3 Conector Inferencia.- establece la principal comunicación del Motor de Inferencia con lo elementos arquitectónicos que une, al sincronizar peticiones, reenviando los servicios solicitados, a través de sus cinco puertos. Este conector importa tres aspectos: persistencia, coordinación y distribución.

Este elemento arquitectónico es quien realiza el proceso de toma de decisión para aplicar la estrategia de razonamiento. Ante la solución de un problema, se aplicarán aquellas estrategias de razonamiento que le permitan resolver el problema de la forma más eficiente, considerando que la estrategia de razonamiento que resulta más adecuada para solucionar una parte del problema, no tiene porque ser la más adecuada para resolver otra parte del mismo problema. En cada ciclo, son ejecutados de uno a tres comportamientos. Para determinar el orden en el cual se ejecutarán los razonamientos, se tiene en cuenta el siguiente proceso: primero se lleva a cabo el razonamiento deductivo con el fin de llegar a un diagnóstico; si se llegase a dos o más posibilidades diagnósticas, se realizará el razonamiento diferencial, invocando al razonamiento inductivo para reformular preguntas al usuario o buscar datos en la memoria de trabajo que permitan determinar cuál de las posibilidades diagnósticas es la más cercana al problema analizado.

3.2.2 Nivel de abstracción 2:

3.2.2.1 Componente Motor de Inferencia.- este componente complejo o sistema es un elemento arquitectónico de mayor granularidad que encapsula el control del sistema (ver figura 1). Dado que es independiente del conocimiento, un mismo Motor de Inferencia puede ser la base de varios sistemas para diferentes casos de uso; esto significa que una variación en la Base de Conocimientos puede significar una variación en el resultado pero no variará el control y por tanto éste permanecerá libre de errores debidos a cambios en la misma. Este componente cuenta con dos puertos: un puerto por el que establece comunicación con el componente Base de Conocimientos para adquirir la información del dominio, y un segundo puerto para comunicarse con el componente Usuario con el fin de obtener los valores de los datos observables. El componente Motor de Inferencia no contiene aspectos emergentes.

3.2.2.2 Componente Base de Conocimientos.- contiene el conocimiento del dominio del caso de estudio, mediante reglas de inferencia (cláusulas de Horn) y hechos (información que es invariable de una a otra resolución). Este elemento arquitectónico

contempla un almacén temporal de información dinámica; en ésta es almacenada, en forma de hechos, toda la información brindada por el usuario al sistema (respuestas a preguntas formuladas), así como las conclusiones de las reglas disparadas en el transcurso del proceso de inferencia. Cuando el proceso de solución de un problema particular ha concluido, el contenido de la memoria de trabajo es removido o eliminado, de tal forma que esta memoria queda limpia antes de iniciar la solución de otro problema. Este componente requiere y oferta sus servicios a través de un solo puerto, y define su comportamiento a través de los aspectos funcional y de distribución.

3.2.2.3 Conector Diagnóstico.- se encarga de coordinar las comunicaciones, a través de sus cuatro puertos, entre los componentes Motor de Inferencia, Base de Conocimientos y Usuario. En el comportamiento de este conector intervienen dos aspectos: coordinación y distribución.

3.2.3 Nivel de abstracción 3:

3.2.3.1 Componente Shell.- es un sistema o componente complejo que encapsula los elementos arquitectónicos del nivel 2, el cual establecerá comunicación con el entorno. Este componente, cuenta solamente con un puerto por el que se publican los servicios (ver figura 2). Este elemento arquitectónico no contiene aspectos emergentes.

3.2.3.2 Componente Usuario.- establece la interacción humano-máquina al permitir la comunicación entre el usuario y el sistema. A través de él, el usuario ofrece datos iniciales al sistema o responde a preguntas formuladas por éste. Este componente tiene un sólo puerto, e importa cuatro aspectos: funcional, presentación, distribución y replicación.

3.2.3.3 Conector Sistema.- realiza tareas de coordinación, entre los componentes Usuario y Shell, a través de sus dos puertos. Para este conector se definen dos aspectos: coordinación y distribución.

3.2.4 Nivel de abstracción 4: Sistema de Diagnóstico.- ya identificados los elementos arquitectónicos del modelo, se especifica el modelo arquitectónico final (ver figura 3), que define la topología de dicho modelo.

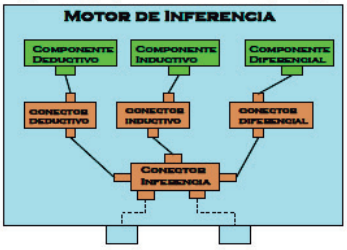


Fig. 1 Motor de Inferencia

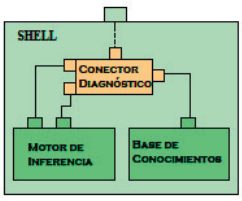


Fig. 2 Shell

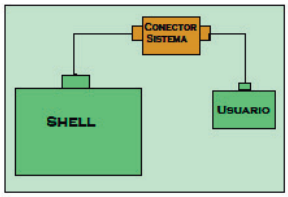


Fig. 3 Modelo arquitectónico

4 Resultados

Se ha realizado la especificación de dicho modelo utilizando el LDA de PRISMA en sus dos niveles: el de tipos para definir los elementos arquitectónicos del modelo, y el nivel de configuración para especificar la topología del modelo. El modelo arquitectónico, posteriormente podrá ser instanciado, obteniendo una topología específica para cada uno de los sistemas que se deseen ejecutar.

Para obtener la validación del modelo presentado en este trabajo, utilizando el compilador de modelos PRISMA, se generó automáticamente tanto su especificación

(en el LDA de PRISMA) como su código (en C#), lo que permite ejecutar dicho modelo sobre el middleware PRISMANET [5].

Cabe señalar que el modelo PRISMA ha sido aplicado a otros dominios, como los sistemas de tele-operación para la limpieza de buques por medio de un brazo-robot [3], así como en sistemas de manejo de cuentas bancarias [8], mostrando el mejoramiento del desarrollo y mantenimiento de dichos sistemas.

5 Conclusiones y trabajos futuros

La investigación ha propuesto un modelo arquitectónico basado en componentes y orientado a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico. Se han descrito los elementos arquitectónicos que forman el modelo, en cuatro niveles de granularidad (del más simple al más complejo), y se han detectado diversos aspectos: integración, coordinación, funcional, distribución, replicación y presentación. El sistema propuesto exhibe tres comportamientos o estrategias de razonamiento (deductivo, inductivo y diferencial), aplicando la estrategia más conveniente según lo requieran las características del problema.

La arquitectura de este sistema conserva las características y el estilo de funcionamiento similares al de otros sistemas expertos conocidos. Sin embargo, la aproximación presentada en este trabajo, frente a la gran variedad de sistemas expertos existentes para resolver tareas de diagnóstico, aporta las siguientes ventajas:

- puede aplicarse para resolver problemas de diagnóstico en diversos campos del conocimiento (generalización de la aproximación),
- utiliza varias estrategias de razonamiento (i.e. es multirazonamiento),
- se puede utilizar en sistemas centralizados o distribuidos,
- la complejidad del sistema se gestiona mediante diferentes niveles de granularidad a través de la separación de las unidades funcionales,
- está formado por elementos arquitectónicos independientes, de forma tal que se pueden añadir o intercambiar componentes y/o conectores, sin afectar el funcionamiento general del sistema ni el resto de sus elementos que lo conforman,
- contempla la integración de componentes de proveedores externos (COTS),
- los diferentes intereses (*crosscuting concerns*) del sistema se han separado en aspectos, favoreciendo el mantenimiento,
- tanto los aspectos como los componentes son reutilizables, ya sea dentro de la misma arquitectura en dominios diferentes o bien en arquitecturas diferentes,
- es independiente de plataforma y su código es generado automáticamente a través de una herramienta (compilador PRISMA), y
- es abordado desde la perspectiva del problema y no de la solución.

Con base en la arquitectura propuesta, puede mejorarse este sistema, añadiendo al modelo varias Bases de Conocimientos (conocimiento distribuido), o cambiando el Motor de Inferencia por otro que contemple otras estrategias de razonamiento adecuadas para el proceso de diagnóstico. Asimismo, y como trabajo futuro, incorporar al modelo PRISMA un nuevo tipo de aspecto, para dotarlo de capacidades de decisión.

Referencias

[1] Bachman F., Bass L., Chastek G., Donohoe P. and Peruzzi F., "The Architecture Based Design Method", *Technical Report CMU/SEI-2000-TR-001*, Carnegie Mellon University, Pittsburgh, USA, 2000, 61 p.
[2] Cabedo R., Pérez J., Carsí J.A. y Ramos I., "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", *en Actas del IV Workshop DYNAMICA*, Archena, Murcia, España, 2005, pp. 79-86.
[3] Cabedo R, Pérez J. y Ramos I., "Aplicación del lenguaje de descripción de arquitecturas PRISMA a un sistema robótico industrial", *Informe técnico DSIC-II/12/05*, Universidad Politécnica de Valencia, Valencia, España, 2005, 201 p.
[4] Cabello M.E., "Propuesta de un sistema experto multirazonamiento para tareas de diagnóstico", ó*Tesis para obtener el grado de Maestro en Ciencias de la Computación*, Fundación Arturo Rosenblueth, México, 2000, 155 p.
[5] Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I., "PRISMANET: Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", *en Actas de las X Jornadas de Ingeniería del Software y Bases de Datos*, Granada, España, 2005, pp. 27-34.
[6] Giarratano J. and Riley G., *"Expert Systems: Design and Development",* Course Technology, 4a. ed., 2004, 856 p.
[7] Liao S.-H., "Expert Systems Methodologies and Applications- a Decade Review from 1995-2004", *in Expert Systems with Applications*, Vol. 28, Issue 1, 2005, pp. 93-103.
[8] Pérez J., Ali N., Carsí J.A., Ramos I and Navarro E., "Designing Software Architectures with an Aspect-Oriented Language", *in Advice: Journal of Aspect Orientation*, Vol. 1, No. 1, 2004, pp. 1-18.
[9] Pérez J., Ali N., Carsí J.A. and Ramos I., "Dynamic Evolution in Aspect-Oriented Architectural Models", *Second European Workshop on Software Architecture, in Lecture Notes in Computer Science*, Springer Verlag, Vol. 3527, Pisa, Italy, 2005.
[10] Szyperski C., *"Component software: beyond object-oriented programming",* ACM Press and Addison Wesley, New York, USA, 1998.
[11] Aspect-Oriented Software Development, in: http://aosd net.
[12] Model Driven Architecture, in: http://www.omg.org/mda.

Developing Mobile Ambients using an Aspect-Oriented Software Architectural Model

Nour Ali, Carlos Millán, Isidro Ramos

Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n
E-46022 Valencia, Spain
{nourali, cmillan, iramos }@dsic.upv.es

Abstract. Nowadays, distributed and mobile systems are acquiring greater importance and becoming more widely used to support ubiquitous computing. However, developing systems of this kind is a difficult task. Instead of concentrating on how problems should be solved developers must worry about implementation details. Ambient Calculus is a formalism that provides primitives to describe mobile systems in an abstract way. Aspect-oriented software development and software architectures promise to achieve reusability, maintenance and adaptability, which are all essential for the development of distributed systems. In this paper, we present how a platform-independent model called Ambient-PRISMA combines both Ambient Calculus and Aspect-Oriented Software Architecture for the specification of distributed and mobile systems. A platform-specific model in .Net for supporting Ambient-PRISMA code generation is also presented.

1.Introduction

Computer networks have become pervasive and ubiquitous. They allow users to communicate and to collaborate independently of their locations. They also involve new devices that form part of our daily life such as PDAs or smartphones. Nowadays, software systems have to cope with these advances in order to exploit their distributed nature. Distributed software systems have to take into account particularties; for example complex structures, new non-functional requirements such as fault tolerance and security, and dynamic adaptation. As a result, software development processes must be able to support the distributed, mobile and ubiquitous nature of software systems.

The development of these software systems is a difficult task. Currently, decisions about these characteristics are usually postponed to late stages of the software life cycle (design and implementation). As a result, there is a loss of traceability, and the system is subject to a specific technological platform. Developers of distributed systems also have to spend more time programming than solving and considering distribution problems. The development of systems of this kind introduces important challenges such as: how to specify distribution and mobility features in a technology-

Ali, N., Millan, C., Ramos, C. Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model. Proc. of the 8th International Symposium on Distributed Objects and Applications (DOA). 2006.

independent way; how to consistently manage a distributed state; and how to support non-functional requirements such as security or fault tolerance.

Model Driven Engineering (MDE) [1] is a software development approach based on the transformation among models. Model Driven Architecture (MDA) [2] is the MDE approach proposed by the Object Management Group (OMG). The MDA proposes the development of software by using Platform Independent Models (PIMs), describe a system in a technology-independent way. Then, PIMs are transformed to a Platform Specific Model (PSM), which describes a system based on a specific technology. An application can be generated from the PSMs.

Component Based Software Development (CBSD) [3] and Aspect-Oriented Software Development (AOSD) [4] are two techniques that promise to achieve reusability, flexibility, and maintainability through the software development process. CBSD proposes to construct the system by connecting entities that provide and require services. AOSD allows the separation of concerns by modularizing crosscutting concerns in a separate entity called an aspect.

In this paper, we present a metamodel called Ambient-PRISMA that allows the definition of distribution and mobility characteristics in a technology-independent way. Ambient-PRISMA extends the PRISMA metamodel [5] with Ambient Calculus (AC) [6] concepts. PRISMA integrates CBSD and AOSD to specify software architectures. AC is a formalism that describes distribution and mobility properties in an abstract way.

In addition to the metamodel, Ambient-PRISMA uses the Aspect-Oriented Architecture Description Language (AOADL) [7] to allow the description of the distribution and mobile characteristics of software architectures at a PIM level. This work shows how a mobile agent case study can be described in a platform-independent way using the AOADL. A PSM is also proposed in order to support the generation of PIMs that are based on the Ambient-PRISMA metamodel to .Net technology.

The paper is structured as follows: Section 2 presents the state of art of distribution and mobility characteristics supported in Architecture Description Languages (ADLs). Section 3 gives an overview of AC. Section 4 presents the Ambient-PRISMA metamodel and describes how the distribution and mobility characteristics of a PIM can be specified using the AOADL. In addition, a PSM of Ambient-PRISMA in .Net technology is introduced. Finally, Section 5 presents conclusions and future works.

2 The State of Art of Distribution in Architecture Description Languages

One of the reasons why software architectures emerged was to simplify the construction of dynamic distributed systems. However, at the present time, few ADLs support the specification of distributed system properties. The first research that provided significant results in distributed software architectures was carried out in the Darwin ADL [8]. Darwin uses π-calculus [9] to define the semantics of distributed message-passing. It builds architectures by defining composite components that are

given locations and bound at instantiation time. Darwin also has been used in the CORBA environment to specify the overall architecture of component-based applications [10]. However, in the literature, we have not found new advances to Darwin in constructing software architectures with mobile components. As Darwin is based on π-calculus only, mobility can only be simulated by the movement of channels. It lacks primitives to express the movement of components that cross boundaries.

The work in [11] introduces features such as composition, reusability, and configuration that an ADL should consider in order to specify dynamic and distributed software architectures. It presents a configuration language that describes a method for a reconfiguration model at run-time. However, the reconfiguration model is not formal. Moreover, it neglects a distribution model that specifies distributed message-passing among components and connectors.

The works of Mascolo and Ciancarini [12] [13] introduce MobiS, which is a specification language that is based on a tuple-space model that specifies coordination by multiset rewriting. MobiS can also be used to specify architectures containing mobile components. However, it does not specify the mobility concern separately from the rest of the functionalities of the software architectures, reducing reusability and adaptability to changes.

The ADL C2Sadel has adapted a style to support both distribution and mobility. The style [14] provides software connectors that are able to move components. It also has an implementation infrastructure to support this architectural style. However, this approach has the drawback that there is no separation between coordination and distribution. Therefore, the components are the only architectural elements that are mobile and the connectors are static.

The work of Lopes in [15] describes the semantics of externalizing a distribution dimension to support distribution and mobility for software architectures. This distribution dimension is very similar to a connector, but instead of containing the business logic, it controls the rules for mobility and location. In this way, a separation between computation, coordination and distribution is achieved. A difference between our work and Lopes's work is that our work defines the semantics of distribution and mobility by using Ambient Calculus. This allows our approach to have an explicit primitive to represent a location with boundaries, which allows the specification of security and authentication.

3 Ambient Calculus

Ambient Calculus [3] (AC) is a process algebra that extends π-calculus [11] to introduce the concept of ambient. An ambient is a bounded place where computation occurs. Thus, an ambient can be anything with a boundary such as a laptop, a web page, a folder, etc. Each ambient has a set of running computations that can control it. These are responsible for moving an ambient. In addition, an ambient can contain other subambients that have running computations.

Thus, mobility is performed at an ambient level, i.e. ambients are mobile. Mobility is also performed by crossing ambient boundaries. AC provides mobility and local

communication primitives. These primitives can be expressed in a textual syntax and in a graphical syntax which is called Folder Calculus [3] (see Figure 1). Folder Calculus is a graphical metaphor for AC where ambients are visually represented as folders. For example in Figure 2, the folders n and m represent ambients. In our work, we have chosen a visual notation similar to the one of Folder Calculus as it will be noticed in the paper.

AC uses some of the constructs inherited from π-calculus such as naming, restriction, parallel processes, inactive processes and replication. However, the names in AC are names of ambients instead of names of channels as in π-calculus.

Textual Syntax	*Visual Syntax*	*Comments*	*Textual Syntax*	*Visual Syntax*	*Comments*
$(\nu n)P$		New name n in a scope P.	0		Inactive process (often omitted).
			$!P$		Replication of P.
$n[P]$		Folder (ambient) of name n and contents P.	$\langle M \rangle$		Output M.
$M.P$		Action M followed by P.	$(n).P$		Input n followed by P.
$P \mid Q$	P Q	Two processes in parallel. (Visually: contiguously placed in 2D.)	(P)		Grouping

Fig. 1. The Textual and Visual Syntax of Ambient Calculus constructs taken from [3]

Some of the primitives that AC provides are called capabilities. Capabilities are actions that can be performed on ambients. There are three main types of capabilities: enter, exit, and open capabilities. The enter capability orders an ambient to enter another ambient on its same hierarchy level (see Figure 2). The exit capability orders an ambient to exit its parent ambient. The open capability dissolves an ambient leaving the processes that were in it.



Fig. 2. Applying the enter capability to the ambient n

Cardelli proposed an algorithm for implementing ambient calculus in Java [16]. The implementation describes a tree structure where each node of the tree is an ambient. The tree structure is modified through operations that implement the enter and exit capabilities. However, this implementation has a limited application as it does not consider distributed nodes and has to be extended for distributed applications.

To date, we have only found a few implementations that support distributed software systems for ambient calculus. In [17], a scheme is proposed in C# for supporting ambient calculus primitives in distributed applications through an ambient

runtime that sits above the .Net framework. Nevertheless, there has been no futher work to implement or apply the ambient runtime.

4 Supporting the Development of Mobile Ambients through an Aspect-Oriented Software Architecture Model

In this section, we are going to present the Ambient-PRISMA metamodel that combines ambient calculus primitives and Aspect-Oriented Software Architecture in order to allow the description of the distribution and mobility characteristics in a Platform-Independent way. Then, the AOADL will be used to describe the mobile and distribution characteristics in a case study of the mobile agent field. A PSM in C# will also be presented to support the generation of code from PIMs. The concepts of the AMBIENT-PRISMA are mapped to .NET.

4.1 Ambient-PRISMA Metamodel

A metamodel is needed to allow the definition of PIMs. A metamodel defines the ontology used to define a PIM. This section presents the metamodel of Ambient-PRISMA that allows the definition of PIMs of distributed and mobile software systems. The Ambient-PRISMA metamodel extends the PRISMA metamodel [5] by introducing concepts of the ambient calculus. The result of the Ambient-PRISMA metamodel is to allow the description of distributed and mobile system software architectures by using both the aspect-oriented and the component-based software development techniques.

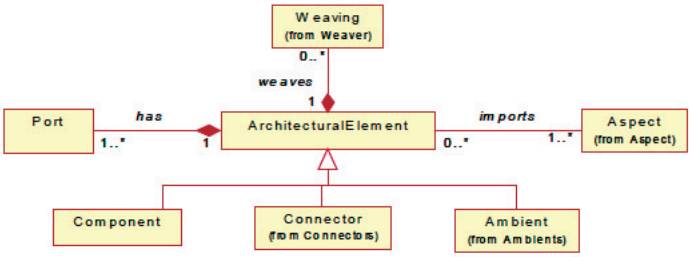


Fig. 3. Ambient as an aspect-oriented, component-based architectural element

The Ambient-PRISMA metamodel introduces the ambient concept as an architectural element (see Figure 3), which is responsible for providing mobility features to distributed architectural elements. As a result, an ambient inherits the CBSD and AOSD characteristics of PRISMA (as do other architectural elements of PRISMA such as components and connectors). The CBSD characteristics describe an ambient as a black box where it communicates with others by using ports that send and receive invocations of services. To enable the communication among

architectural elements, channels called attachments are defined. Each attachment is defined by attaching two ports of different architectural elements.

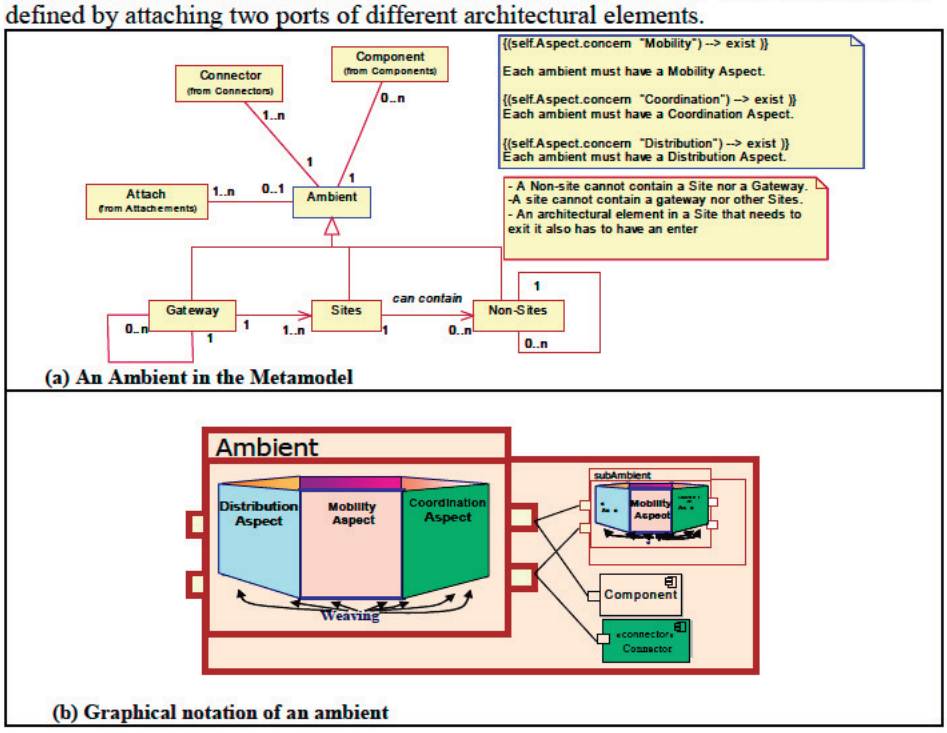


Fig. 4. The Ambient package in the Metamodel

In Ambient-PRISMA, ambients are complex architectural elements that represent the places where components, connectors and other ambients are located. Therefore, in the metamodel, the ambient is associated with the other architectural elements (see Figure 4(a)). For example, in Figure 4(b) the *Component* and *Connector* are located in the same ambient. Also, an ambient can have other subambients. This allows the hierarchy of distributed and mobile systems to be modelled in Ambient-PRISMA. An ambient offers two type of services to architectural elements located in it: mobility services and distributed architectural element services. Architectural elements that need these services are connected to the ambient through attachments (lines in Figure 4(b)).

The AOSD view describes the PRISMA ambient with a set of aspects that can be weaved. Weavings indicate that the execution of an aspect service can trigger the execution of services in other aspects. Weavings are the glue of the aspects of an architectural element. This glue is defined using temporal operations called weaving operators. For example, if a weaving with the *after* operator is specified between service s1 of aspect A1 and service s2 of aspect A2, this means that s2 of A2 is executed after s1 of A1.

The ambient uses different aspects to specify the services it offers and requests. As Figure 4(a) shows each ambient must have a Mobility, a Coordination, and a

Distribution Aspect. These constraints are specified in the Object Constraint Language (OCL).

As ambients are responsible for the mobility concern, all ambients must have the Mobility Aspect to provide mobility services to their local architectural elements. These services are the ones that provide the ambient calculus capabilities. The following are the Mobility Aspect functionalities:

- It allows an ambient to offer the exit service to the subambients that need to exit from it. (The specification of the AC exit capability).
- It allows an ambient to offer the enter service to the subambients that need to enter other subambients. (The specification of the AC enter capability).
- It allows an ambient to create subambients. (The specification of the AC restriction).
- It allows an ambient to accept a new ambient from external ambients.
- It allows an ambient to execute the open service. The open service allows a subambient to be destroyed without destroying its local architectural elements. As a result, the architectural elements of the destroyed subambient become part of the including ambient. (The specification of the AC open capability)

The separation of the Mobility Aspect concern from the rest of the concerns provides better maintainability of these functionalities because they are not scattered through the ambient specification. In addition, this is a generic aspect that must be reused by all PRISMA ambients. As a result, ambients are defined by importing the generic mobility aspect and adapting it to the needs of the software system through weavings. For example, a LAN ambient may need some security policies that are different from a PC ambient inside of the LAN. Therefore, both the LAN and PC ambient import the same Mobility Aspect, but the Mobility Aspect is weaved with different security aspects.

The coordination aspect of an ambient is needed to coordinate its architectural elements with the exterior. This coordination aspect receives calls from external architectural elements of the ambient (distributed calls) and redirects them to corresponding architectural elements of the ambient. It also receives calls from its architectural elements and redirects them to the exterior. This aspect, like the Mobility Aspect, is also a generic one that all ambients must reuse.

The Distribution Aspect of an ambient stores the name of its parent ambient. This aspect also specifies whether or not an ambient is mobile. A mobile ambient is an ambient that requests mobility services from its parent ambient. When an ambient is mobile, its parent ambient can be changed.

In Ambient-PRISMA, we have defined three types of ambients: Sites, Non-Sites, and Gateways (see Figure 4(a)). This separation is necessary since each one of them has its own semantics and constraints. Sites are ambients that represent devices; that is, they have a physical address. A Non-Site is an ambient that represents a collection of software elements. A Non-Site can only have subambients of type Non-Site. This restriction is important because Sites cannot be located in Non-Sites. Also, a Non-Site can be located either in a Site or in another Non-Site. An example of one use of a Non-Site ambient is to move a group of architectural elements together. Another example of a use of a Non-site is to separate certain architectural elements in a Non-site since they share some security protocols. A Gateway is a virtual ambient that

groups either a collection of ambient Sites or other gateways. A Gateway ambient that consists of a collection of Sites represents the borders of a network (e.g. a Local Area Network). The modelling of a Gateway hierarchy can represent the hierarchy of LANs to form a Wide Area Network (WAN). Different gateway hierarchy levels can be represented in the software architecture depending on the software architect and on the distributed requirements. In Ambient-PRISMA there is always a default Gateway ambient that represents the Root.

4.2 Case Study: Mobile Agents in an Auction Site

Mobile Agents [18] are components that can move from one site to another in order to perform tasks in an autonomous way. Mobile agents are helpful as they solve problems that are encountered in distributed systems. Network bandwidth is reduced due to the fact that the agent is moved to a new site where it performs tasks locally instead of passing information through the network. Also, unreliable network connections are not a problem because once an agent has been sent to a site, the operations are locally performed and the results can be sent when a connection is re-established.

Mobile Agents can be applied in different applications. In this paper we consider an electronic commerce application: an auction site where products are auctioned. A customer of this auction site is interested in buying a specific product with certain specifications at a maximum price. To keep track of the new auctions being performed on the auction site the customer designs two mobile agents to take charge of the purchase. One of these agents is responsible for collecting information and the other agent is in charge of managing the purchase. The two agents have to collaborate with each other. The customer sends the agents to the auction site. The agents act for the customer. When the purchase is performed the agents return to their original location.

4.3 The PIM of the Case Study Specified in the AOADL of Ambient-PRISMA

To allow the definition of PIMs, a language has to be used. The language can be a graphical language such as UML or a textual language. A graphical language, which is user friendly, is an easy way to allow users to develop their PIMs. However, in order to formalize the semantics of a PIM, it is necessary to have a formal language. In this section we focus on our formal AOADL to describe the distribution and mobility characteristics of the PIM of the case study presented in section 4.2.

In the case study, the customer designs the agents: the *purchaser* and the *collector*. As the *purchaser* and the *Collector* need to collaborate with each other, a connector called *AgentCnctr* coordinates them. The customer locates these architectural elements in a Non-Site ambient called *AgentsNonSite*. The *Customer* software component is connected with the *AgentsNonSite* by a connector called *AgentCustCnct*. Initially, the *Customer* and the *AgentsNonSite* are located in the *ClientSite*. Figure 5 shows the configuration of the software architecture before the *AgentsNonSite* ambient is moved to the *AuctionSite*. In the *AuctionSite*, there is a component that represents the *Auction*. The *AuctionCnct* is a connector that coordinates the *Auction* with other architectural elements. As, the *Auction* component

can communicate with distributed architectural elements, the *AuctionCnct* has an attachment with *AuctionSite* ambient.

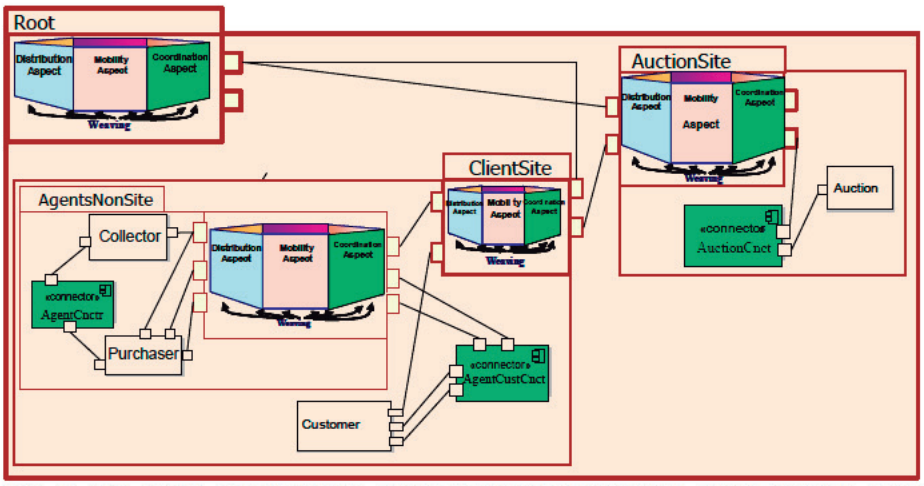


Fig. 5. The Initial Configuration of the Software architecture of the Mobile Agent Case Study

```
Distribution Aspect ANSDistribution using IMobility, ICapability
 Attributes
  location : loc;
 Services
   … …
   changeLocation(Name: String, NewLocation:loc)
        Valuations
          [changeLocation()] location:=NewLocation;
   getLocation( output Location:loc)
        Valuations
          [getLocation()] Location:=location;

   out exit (Name: String, Parent:loc );
   out enter (self.Name: String, NewAmbient: loc);
   … …
  Transactions move(NewAmbient: loc)
      move::= out exit(self.Name, location)→MOVE1;
      MOVE1::= {exitOK} out enter(self.Name, NewAmbient)→MOVE2;
 Preconditions
  in  changeLocation(Name, NewLocation)
                          if {self.Name==Name};
      … … …
End_Distribution Aspect ANSDistribution
```

(a)

```
Ambient_NonSite type AgentNonSite
 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ANSDistribution;
 Ports
     ECapabilitiesPort: ICapability;
     EDistServicesPort: ICall;
     EMobilityPort: IMobility;
     InServicesPort: ICall;
     InCapabilitiesPort: ICapability;
     InMobilityPort: IMobility;
 End_Ports
… …
End Ambient_NonSite type AgentNonSite;
```

(b)

Fig. 6. Specification of the AgentNonSite ambient and its distribution aspect

The mobility of the *AgentsNonSite* is initiated when the *Customer* component requests it to move to the *AuctionSite*. Figure 6(b) shows the specification of the *AgentNonSite* in the AOADL. The *AgentNonsite* receives the mobility request through its port, called *EMobitityPort*. This causes the execution of *move(NewAmbient:loc)* of the *ANSDistribution* Distribution Aspect that the ambient imports. Figure 6(a) shows the specification of the *ANSDistribution* aspect. The

move(NewAmbient:loc) specifies that the *AgentNonSite* has to *exit* its parent ambient and then *enter* another ambient. The *enter* and *exit* services are served by the parent ambient of the *AgentNonSite*.

```
Distribution Aspect SiteDistribution
 Attributes
  location : loc;
 Services
   … …
   getLocation( output Location:loc)
        Valuations
          [getLocation()] Location:=location;
…
End_Distribution Aspect SiteDistribution
```

(a)

```
Ambient_Site type ClientSite
 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect SiteDistribution;
 Import Security Aspect ClientSecurity;
 Ports
     ECapabilitiesPort: ICapability;
     EDistServicesPort: ICall;
     InServicesPort: ICall;
     InCapabilitiesPort: ICapability;
 End_Ports
 Weavings
     SiteDistribution.getLocation(Location) before
                                  Mobile.getParent(Parent);
   ……
End Ambient_Site type ClientSite;
```

(b)

Fig. 7. Specification of the ClientSite ambient and its distribution aspect in the AOADL

The *AgentNonSite* then requests the *exit(AgentNonSite, ClientSite)* from the *ClientSite* through the port *ECapabilitiesPort* (see Figure 6(b)). The *ClientSite* receives this petition through its port *InCapabilitiesPort* (see Figure 7(b)). As a consequence, the *Mobile* Mobility Aspect is executed.

```
Mobility Aspect Mobile using ICapability
   …. ….
 TRANSACTIONS in exit(Requested: String, Ambient: loc):
  EXIT := out isChild(input Requested, output isChildOK)→
          EXIT1;
  EXIT1::= {isChildOK==true & self.Name==Ambient}
           getParent(output Parent)→EXIT2;
  EXIT2::= moving (Requested,Parent);

TRANSACTIONS in enter(Requested: String, NewAmbient: loc):
  ENTER := out areChildren(input Requested, input Ambient,
           output areChildrenOK)→ENTER1;
  ENTER1::= {areChildrenOK==true} moving(Requested, NewAmbient);

TRANSACTIONS in moving(Requested: String, NewAmbient: loc):
  ……..

TRANSACTIONS in accept(input NewAmbient:loc, input Caller: loc,
                input Child:String, input Type:String,
                input AttachmentsList[]:Attachments,
                output AcceptanceOK:boolean):
  ACCEPT::= out addChild(input Type, input Child,
                input AttachmentsList[], output AddedOK)
                {Acceptance==AddedOK}→ ACCEPT1;
  ACCEPT1::=out changeLocation(Child, self.Name);

Preconditions
       in accept(input NewAmbient, input Caller,
                 input Child, input Type,
                 input AttachmentsList[],
                 output AcceptanceOK) if
                                    {self.Name==NewAmbient};
 … …
End_Mobility Aspect Mobile
```

Fig. 8. A segment of an Ambient Mobility Aspect specified in the AOADL

Figure 8 shows a segment of the Mobility Aspect specified in the AOADL, which all ambients must reuse. It shows that the *exit* is a ***transaction*** that consists of a set of services. First, the *exit* checks if the requested element (*AgentNonSite*) is one of its children (is one of its local elements). Since the *AgentNonSite* is its child, the *ClientSite* executes the *getParent* service in order to know who its parent ambient is. This service, as shown in Figure 7(b), has a ***Weaving*** with the *SiteDistribution* aspect. The service *getLocation* of Figure 7(a) is executed to provide the value of *location*. Figure 5 shows the *ClientSite* has the *Root* ambient as its parent. This is a default parent ambient for all sites and is of type *Gateway* as it is a virtual ambient. When the parent value is achieved, the *moving* transaction in Figure 8 is executed. Briefly, the *moving* transaction, is in charge of removing the mobile architectural element (*AgentNonSite*) from the *ClientSite,* modifying its local attachments and ensuring that the receptor (the *Root*) receives and adds it as its child. The result of the configuration after the execution of the exit is shown in Figure 9.

The *AgentNonSite* ambient then requests the *enter(AgentNonSite, AuctionSite)* to the *Root* ambient. The *Root* ambient also has the *Mobile* Mobility Aspect shown in Figure 8. The Root ambient checks if the *AgentNonSite* and the *AuctionSite* are both its children. If they are, then it executes the *moving* transaction. When the moving transaction is finalized, the *AgentNonSite* ambient becomes a child of *AuctionSite* as shown in Figure 10.

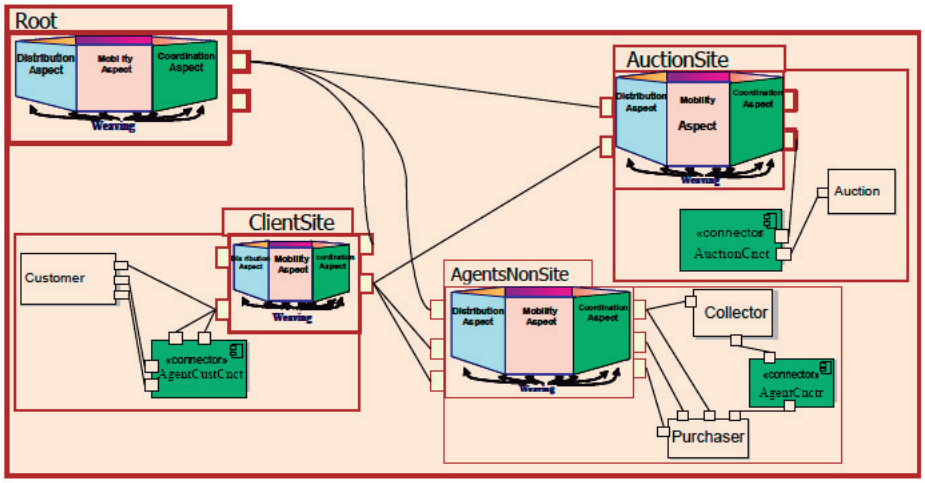


Fig. 9. The Software Architecture when the AgentNonSite executes the exit

It could happen that *AgentNonSite* does not request an *enter* from the *Root*. As explained in Section 4.1, that a Gateway ambient can only have Site ambients or other gateways. Therefore, if the *AgentNonSite* ambient does not request an enter to another Site, the *Root* would send it back to the *ClientSite*.

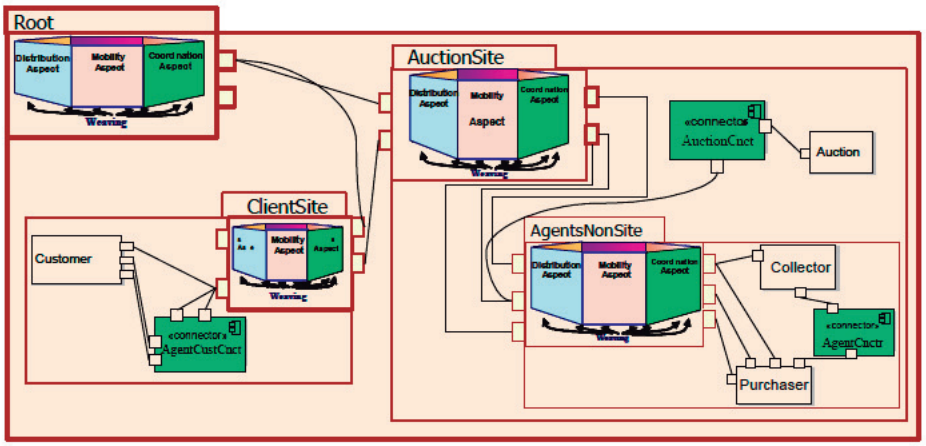


Fig. 10. The AgentNonSite becomes a child of AuctionSite

As Figure 10 shows, the *Customer* component can still send commands to the agents of the *AgentNonSite* in the *AuctionSite*. This is achieved thanks to the fact that all the attachments between the *AgentCustCnct* and the *AgentNonsite* (see Figure 5) have been replaced by attachments between these elements and their ambients (see Figure 10). Thus, distributed communication is performed through ambients.

When the *purchase* agent in Figure 10 performs the purchase that the customer desires, it invokes the *move* of the *AgentNonSite* through the *InMobitityPort* (see Figure 6(b)). The entire process is repeated until the *AgentNonSite* returns to *ClientSite* (the architecture configuration returns to the configuration shown in Figure 5).

From the above example, we have seen how the distribution and mobile characteristics of a PIM, in this case of the Mobile Agent field, have been specified using an AOADL based on a defined metamodel.

4.3 Platform-Specific Model in .Net

The PRISMA metamodel is supported in .Net by the PRISMANET middleware [19]. The PRISMANET middleware provides an execution platform for PRISMA aspect-oriented software architecture configurations. For example, it contains classes that map the concepts of aspects, ports and attachments. Each PRISMANET middleware of a host can collaborate with PRISMANET middlewares on other hosts in order to provide a distributed runtime environment (see Figure 11). The runtime environment has a distributed Domain Name Server (DNS) where all middlewares on hosts can access. PRISMANET must be extended in order to provide the mobility and distributed communication of architectural elements that Ambient-PRISMA provides. Therefore, in this section a PSM is proposed for supporting Ambient-PRISMA basing on PRISMANET.

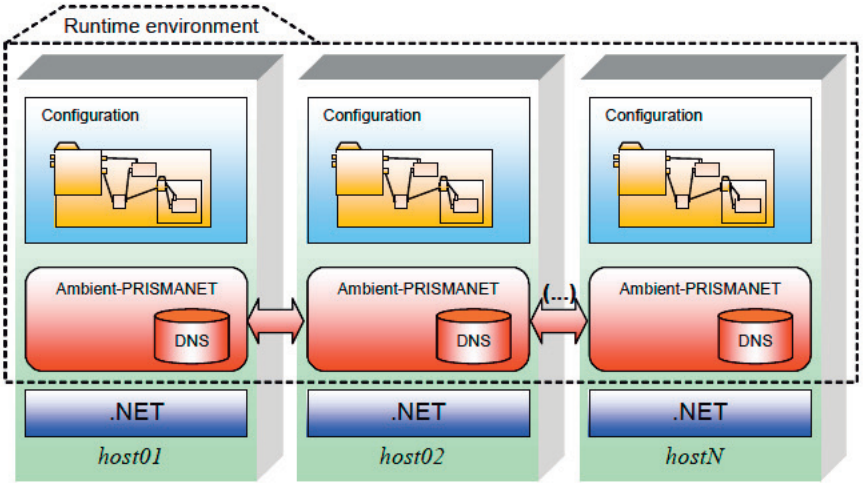


Fig. 11. Distributed Runtime Environment of PRISMANET

Ambient-PRISMANET extends the PRISMANET middleware in order to include the ambient architectural elements. Figure 12 shows the class *ArchitecturalElement* that includes the common functionality of all architectural elements. All architectural elements are composed of aspects, weavings, and ports. The classes *Component*, *Connector* and *Ambient* inherit from *ArchitecturalElement*. Using the PIM of the case study defined in section 4.2 the Collector, the Purchaser, the Customer, and the Auction are represented in C# as classes that inherit from the class *Component*. The AgentCustomerCnct, the AgentCnctr and the AuctionCnctr are classes that inherit from the class *Connector*.

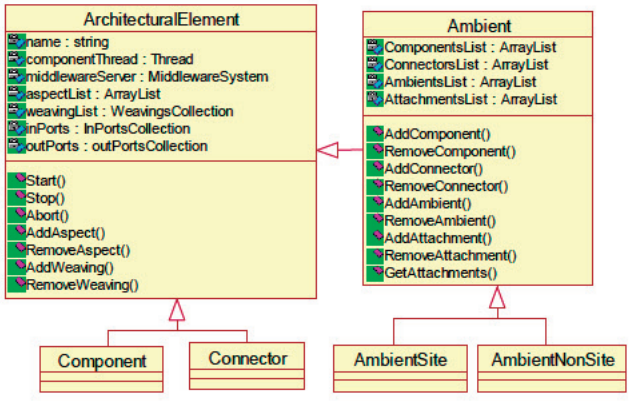


Fig. 12. The Ambient inherits from ArhitecturalElement class of PRISMANET

The class *Ambient* provides services for managing the information related to the architectural elements of an ambient. These services can change the state of an

ambient by adding or removing either architectural elements or attachments. For example in the case study, when the AgentNonSite ambient requests to exit from the ClientSite ambient, the ClientSite has to replace the attachments between the AgentNonsite and the AgentCustCnctr with attachments between the AgentCustCnctr and the port of the ClientSite that provides distributed communication. Therefore, an ambient needs services that facilitate the dynamic generation of attachments depending on its state.

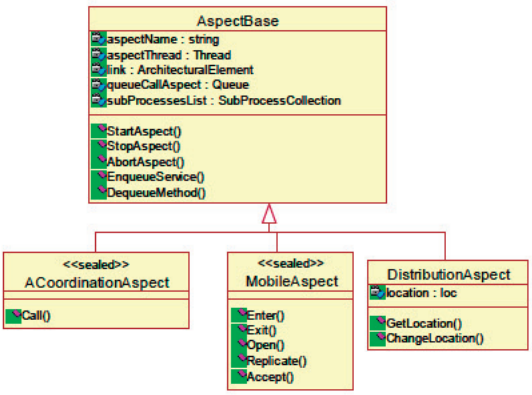


Fig. 13. The Aspects of an ambient

In addition, all ambients must have a Mobility Aspect, a Coordination Aspect and a Distribution Aspect. The Mobility and Coordination Aspects of an ambient are predefined ones that all ambients must reuse and that designers do not modify. As these aspects have a predetermined definition, they are included as specific types of aspects that inherit from the class *AspectBase* (the class that represents a PRISMA aspect). Figure 13 shows the class *MobileAspect* that maps the Mobile Aspect and *ACoordinationAspect* that maps the ACoordination Aspect. They are marked with the C# sealed modifier in order to represent that they cannot be extended by the user. The class *MobileAspect* publishes its interface through its port in order to offer the mobility services to its architectural elements. The interface of the predefined Mobility Aspect is shown in Figure 14.

```
  public interface ICapability
  {
  AsyncResult Enter(string AmbientName);
  AsyncResult Exit(string AmbientName);
  AsyncResult Open(string Ambient);
  AsyncResult Replicate(string Ambient);
  AsyncResult Accept(string grandparent, string parent, string type,
string children, ArrayList attachmentsList, out bool acceptanceOK);
  }
```

Fig. 14. The interface ICapability of the Mobility Aspect in C#

On the other hand, the distribution aspect of an ambient can be extended by the user. As a result, a class *DistributionAspect* that also inherits from *AspectBase* has

been created. The class *DistributionAspect* implements the minimum functionality of a distribution aspect. That is, it stores the reference of the ambient location and the services needed for managing it. For example, the *ANSDistribution* aspect of Figure 6(a) inherits from class *DistributionAspect* extending it with the mobility requests.

Moreover, the classes *AmbientSite* and *AmbientNonSite* have been introduced by inheriting from class Ambient (see Figure 12) in order to represent Ambient-PRISMA Site and Non-Site ambients repectively. Each of these classes expresses the restrictions of the type of ambient they represent. For example, AgentsNonSite ambient must inherit from the class AmbientSite to be implemented. In the same way, ClientSite and AuctionSite ambients inherit from AmbientNonSite to be implemented.

Another Ambient-PRISMA concept that has to be treated is the virtual Gateway Ambient. At implementation this ambient has been decided to be the set of middlewares that collaborate to form the borders of the distributed runtime environment. In this way, the Ambient-PRISMANET middleware has a layer called DistributionLayer that is in charge of the distributed communication among hosts and the mobility of architectural elements between hosts.

In the case study, when the ClientSite needs to execute the exit requested by the AgentsNonSite ambient, the ClientSite has to call the Root ambient in order to request it to accept the AgentsNonSite. This call is mapped to a request that the ClientSite performs to the DistributionLayer of the middleware at its host. As a result, the DistributionLayer stores the state of the AgentsNonSite ambient and its associated attachments. When, the AgentsNonSite exits the ClientSite, at implementation the AgentsNonSite is neither part of the ClientSite nor part of the AuctionSite. When the AgentsNonSite executes the enter to the AuctionSite, the DistributionLayer fetches in the distributed DNS to find out where the AuctionSite exists in order to create a proxy to the DistributionLayer of the middleware where the AuctionSite exists (the receptor). Then, the DistributionLayer of the middleware where the ClientSite exists (the sender) starts the serialization of the AgentsNonSite ambient and its contents. When the DistributionLayer of the receptor receives the serialized ambient, it calls the accept of the AuctionSite ambient. In this way, the DistributionLayer of the receptor acts as the local view of the parent of the AuctionSite (the Root Gateway ambient).

The middleware establishes a timeout from the moment that the AgentsNonSite exits the ClientSite ambient. If the timeout expires, the mobility process is considered to be failed. This failure is due to the fact that a Non-Site ambient can only stay in this intermediate state for requesting an enter to a Site ambient. It is important to mention, that all the operations related to mobility are transactional. This implies that once a failure occurs at any point of the transaction, the operations performed must be undone in order to return to the state previous to the mobility process, i.e. returns the AgentsNonSite to the ClientSite.

5 Conclusions and Further Work

In this paper, we have shown how ambient calculus can be used in software engineering to develop distributed and mobile applications. These applications,

which are based on ambient calculus, can be developed using AOSD and CBSD techniques. The concepts of ambient calculus have been introduced into an aspect-oriented and component based metamodel (PRISMA) in order to achieve Ambient-PRISMA. Ambient-PRISMA provides the construction of distributed and mobile software systems by reusing and adapting specifications.

Ambient-PRISMA has been able to specify the distributed and mobile characteristics of the software architecture of a mobile case study. This has been done in a platform-independent way thanks to MDE. In order to execute distributed and mobile applications based on Ambient-PRISMA, a PSM has been proposed. This PSM supports the implementation of ambient calculus in C#, with the characteristic that it preserves the traceability of the concepts of Ambient-PRISMA.

In the near future, we are going to take advantage of the ambient calculus primitives concerned with security. These primitives will be specified using our AOADL so that the case study presented in this paper could be extended to incorporate security properties. In addition, we are going to introduce the concepts presented in this paper into the PRISMA tool to be able to model and execute mobile distributed software architectures in an intuitive way. This will be done in three stages: first, a complete implementation of the concepts of Ambient-PRISMA will be introduced in the PRISMANET middleware [19] to execute these concepts; second, ambient graphical metaphor and code templates will be introduced in the modelling framework in order to automatically generate code from the graphical metaphor; third, the implementation will be validated by modelling and executing a complex, distributed, and mobile case study.

References

[1] Kent, S., 2002. Model Driven Engineering. *Proceedings of IFM2002*, LNCS 2335, Springer, 2002.

[2] OMG, 2003. MDA Guide, http://www.omg.org/docs/omg/03-06-01.pdf

[3] Szyperski, C., *Component Software: Beyond Object Oriented programming*, ACM Press and Addison Wesley, New York, USA, 2002.

[4] Aspect-Oriented Software Development, http://aosd net

[5] Perez, J., Ali, N., Carsí, J.A., Ramos, I. "Dynamic Evolution in Aspect-Oriented Architectural Models", European Workshop on Software Architecture, Pisa, June 2005 © Springer LNCS vol n.3527.

[6] Cardelli, L. "Abstractions for Mobile Computation." In Vitek, J. and (Eds.), C. J., editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects, volume 1603 of LNCS, Springer Verlag, pp. 51-94.

[7] Pérez, J., Ali, N., Carsí, J.A., Ramos, I. "Designing Software Architectures with an Aspect-Oriented Architecture Description Language", 9th International Symposium on Component-Based Software Engineering (CBSE),Västerås , Stockholm, Sweden, LNCS, June 29th -1st July 2006 (accepted to appear)

[8] Magee, J., Dulay, N., Eisenbach, S., Krammer, J. "Specifying Distributed Software Architectures". 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, 1995, pp 137-153.

[9] Milner, R., Parrow, J., Walker, D. "A calculus of mobile processes", Parts 1-2. Information and Computation, 100(1), 1992, pp. 1-77.

[10] Magee, J., Tseng, A, Kramer, J. "Composing Distributed Objects in CORBA", Third International Symposium on Autonomous Decentralized Systems, Berlin Germany, 1997, pp 257-263.

[11] Virginia C. de Paula, G.R., Justo, Cunha, Ribeiro, P. R. F. "Specifying Dynamic Distributed Software Architectures", XII Brazilian Symposium on Software Engineering, BCS Press, October, 1998.

[12] Ciancarini, P., Mascolo, C. "Software Architecture and Mobility", 3rd Int. Software Architecture Workshop (ISAW-3), November, 1998.

[13] Mascolo, C. "MobiS: A Specification Language for Mobile Systems". 3rd International Conference on Coordination Models and Languages, 1999.

[14]Medvidovic, N., Rakic, M. "Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility". Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.

[15] Lopes, A. Fiadeiro, J.L., Wermelinger, M. "Architectural Primitives for Distribution and Mobility", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, pp. 41-50.

[16] Cardelli, L. " Mobile Ambient Synchronization" SRC Technical Note Report 1997-013, Digital SRC, 1997.

[17] Vasudeva, E., Joshi, Rushikesh K. "A Scheme for Implementing Ambient Calculus over an ARC Framework", Software Design and Architecture Workshop, Dec 2004, IIIT Bangalore (position paper).

[18] Chess, D., Harrison, C., Kershenbaum, A. "Mobile Agents: Are They a Good Idea?" IBM Research Report RC.

[19]Perez, J., Ali, N., Costa, C., Carsí, J.A., Ramos, I. "Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology",3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005 , 2005

Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos Utilizando Ambientes

N. Ali, J. Pérez, C. Costa, I. Ramos y J. A. Carsí

Resumen **Con la expansión de los sistemas distribuidos, la replicación se ha convertido en un requisito fundamental para evitar la sobrecarga en la red y proporcionar tolerancia a fallos. El objetivo de este trabajo es mostrar cómo dar soporte a este requisito de forma abstracta e independiente de la tecnología. Para ello, el trabajo combina PRISMA con el Cálculo de Ambientes (CA). PRISMA es un enfoque de desarrollo para arquitecturas software orientado a aspectos, y el CA es un formalismo que proporciona primitivas para especificar características de distribución, movilidad y replicación. Con ello, esta propuesta tiene las ventajas proporcionadas por PRISMA y el CA.**

Palabras Clave **Computación distribuida, Modelaje, Algebra de procesos, Ingenieria de software.**

I. Introducción

En las últimas décadas, la aparición de nuevas tecnologías y el trabajo en grupo a través de Internet ha generado la necesidad de que los sistemas de desarrollo actuales den soporte a estructuras software complejas, nuevos requisitos no funcionales, adaptación dinámica del software, nuevas tecnologías, nuevas formas de trabajo y dispositivos de diferente índole (PCs, PDAs, teléfonos móviles, etc.). Todos estos nuevos requisitos imponen la necesidad de dar soporte a la distribución, ubicuidad y naturaleza móvil de los sistemas software durante el proceso de desarrollo software.

La mayoría de enfoques posponen el tratamiento de dichos requisitos a la etapa de implementación, no ofreciendo soporte durante las etapas de requisitos, análisis o diseño. Esta forma de abordar los requisitos no funcionales tiene graves consecuencias en el producto software, ya que no se preserva la trazabilidad entre las distintas etapas del ciclo de vida software y se crea una gran dependencia por parte de estos requisitos con la plataforma de desarrollo. Por lo tanto, el mantenimiento y reutilización de código del producto software resulta muy dificultoso.

Estos problemas generan un gran desafío: el definir un enfoque de desarrollo que permita la especificación de propiedades de distribución y movilidad de forma abstracta, evitando así la dependencia de la plataforma tecnológica. Además, dicho enfoque debe tener la capacidad de especificar otras propiedades estrechamente relacionadas con distribución y movilidad, como son replicación, seguridad, tolerancia a fallos, etc. Finalmente, también hay que destacar que el enfoque debe mantener de forma consistente el estado distribuido de las aplicaciones.

Las arquitecturas software se consideran el puente entre las etapas de requisitos e implementación del ciclo de vida software. Éstas describen la estructura de los sistemas software utilizando Lenguajes de Descripción de Arquitecturas (LDAs). Sin embargo, los LDAs no proporcionan constructores para definir requisitos de distribución o movilidad, así como propiedades relacionadas con estos requisitos.

Una de las características fundamentales a las que se le ha de dar soporte en un sistema distribuido es la replicación. La replicación evita la sobrecarga de la red en sistemas distribuidos cuando un componente software está recibiendo más peticiones de las que puede atender, o simplemente, cuando el canal de comunicaciones está dañado. En estos casos, el componente se replica bien en el mismo lugar o en otro distinto tantas veces como haga falta, con el objetivo de solucionar un problema de comunicación y que ésta sea más eficiente.

El objetivo de este trabajo es el dar soporte a esta característica de forma abstracta a nivel arquitectónico. Para ello, se ha realizado la combinación de un enfoque de desarrollo de arquitecturas orientado a aspectos, llamado PRISMA [1], y de un formalismo que proporciona mecanismos para especificar características de distribución y movilidad, conocido como Cálculo de Ambientes (CA) [2]. Esta propuesta pretende no sólo ser capaz de especificar replicación en arquitecturas software de una forma independiente de la tecnología, sino también realizarlo aportando las ventajas añadidas que proporcionan PRISMA y el CA.

El CA mediante su constructor ambiente y las operaciones que ofrece permite definir fronteras de computación, localización de componentes, jerarquías de localizaciones, políticas de seguridad y replicación de componentes de forma abstracta.

PRISMA es un enfoque de desarrollo que combina el

Este artículo ha sido financiado por el Proyecto Nacional de Investigación, Desarrollo e Innovación DYNAMICA TIC 2003 07804 C05 01

N. Ali, J. Pérez, C. Costa , I. Ramos y J. A. Carsí pertenecen al Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia, Camí de Vera s/n, 46022, Valencia (España)
(e mails: {nourali | jeperez | ccosta | iramos | pcarsi}@dsic.upv.es)

Desarrollo de Software Basado en Componentes (DSBC) [3] y el Desarrollo de Software Orientado a Aspectos (DSOA) [4] para la descripción de arquitecturas software. PRISMA está compuesto por un meta-modelo [1], un Lenguaje de Descripción de Arquitecturas Orientado a Aspectos [5], una metodología y una herramienta de desarrollo. Un elemento arquitectónico PRISMA (componente o conector) está formado simétricamente [6] por un conjunto de aspectos de distinto tipo, las relaciones de sincronización entre estos aspectos (*weavings*) y uno o más puertos. El *weaving* define que la ejecución de un servicio de un aspecto puede provocar la invocación de un servicio de otro aspecto. De este modo, PRISMA da soporte al modelado de requisitos no funcionales, no mezcla los distintos concerns del sistema, y mejora la reutilización, mantenimiento y evolución del producto software.

La estructura del trabajo es la siguiente: El apartado 2 realiza una revisión del estado del arte de los LDAs que soportan distribución. El apartado 3 presenta el Cálculo de Ambientes brevemente. El apartado 4 explica como se ha introducido el concepto de ambiente en PRISMA (AMBIENT-PRISMA). El apartado 5 presenta cómo la replicación distribuida es modelada utilizando AMBIENT-PRISMA en un caso de estudio. Finalmente, en el apartado 6 se presentan las conclusiones y trabajos futuros.

II. TRABAJOS RELACIONADOS

La simplificación del desarrollo de sistemas distribuidos y dinámicos es una de las razones que provocaron la aparición de las arquitecturas software. Sin embargo, pocos LDAs proveen mecanismos para especificar características de distribución. Darwin fue el primer trabajo significativo que proporcionó resultados para la especificación de arquitecturas software distribuidas [7]. Dicho LDA utiliza el π-cálculo [8] para definir la semántica del envío y recepción de mensajes de forma distribuida, construye componentes compuestos y asigna localizaciones a las instancias. Sin embargo, al estar basado en π-cálculo, Darwin sólo permite definir la movilidad mediante el movimiento de canales, y carece de primitivas para expresar la replicación y la movilidad atravesando fronteras.

Los trabajos de Mascolo y Ciancarini [9] presentan MobiS, un lenguaje de especificación basado en un modelo de espacio de tuplas que especifica la coordinación mediante la reescritura de multiconjuntos. MobiS también puede ser utilizado para especificar arquitecturas que contienen componentes móviles. Sin embargo, no especifica la movilidad separada del resto de asuntos (*concerns)* de las arquitecturas software, obteniendo una especificación enmarañada que dificulta la reutilización y adaptabilidad del software.

El LDA C2Sadel ha adaptado un estilo arquitectónico para definir características de distribución y movilidad, dándoles soporte mediante una infraestructura. Dicho estilo [10] proporciona conectores capaces de mover componentes. Al igual que MobiS, uno de los inconvenientes de este enfoque es que no realiza una separación entre la coordinación, la movilidad y la distribución.

El trabajo de Lopes [11] describe la semántica de una dimensión para separar la distribución en arquitectura software. Esta dimensión se especifica de forma similar a como se define la coordinación en un conector, pero en lugar de especificar la lógica del negocio, se definen las reglas de movilidad y localización. De esta manera, se consigue separar la computación, coordinación y distribución en elementos software de diferente tipo. Sin embargo, este trabajo no tiene una primitiva explícita para representar una localización delimitada, tal y como ofrece el cálculo de ambientes, y tampoco da soporte a la replicación de componentes.

En [6] se presenta un análisis comparativo de los diferentes LDAs que introducen orientación a aspectos. Sin embargo, ninguno de estos LDAs orientados a aspectos provee primitivas para describir las propiedades de distribución, movilidad y replicación de sus elementos arquitectónicos.

III. CALCULO DE AMBIENTES

El CA [2] es un álgebra de procesos que extiende el π-cálculo [8] para introducir el concepto de ambiente. Un ambiente es un lugar delimitado donde se ejecutan una serie de procesos. Por lo tanto, un ambiente puede ser cualquier cosa con una frontera delimitada como un ordenador portátil, una página web, una carpeta del sistema de ficheros, etc. Además, un ambiente puede contener a otros subambientes. Cada ambiente tiene un conjunto de primitivas que permiten gestionarlo. Dichas primitivas son las que permiten la movilidad de ambientes.

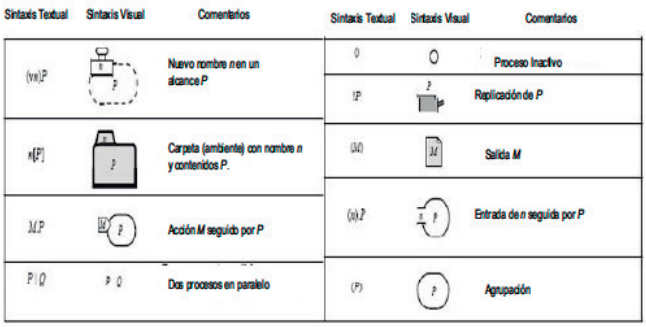


Fig. 1. La sintaxis textual y grafica del cálculo de ambientes [2]

En CA, los ambientes son las entidades móviles. Dicha movilidad se realiza cruzando las fronteras de los ambientes. El CA no sólo proporciona primitivas de movilidad, también ofrece primitivas para la comunicación local. Estas primitivas se pueden expresar con una sintaxis textual o gráfica, llamada Cálculo de Carpetas (*Folder Calculus)* [2] (ver Figura 1). El cálculo de carpetas es una metáfora gráfica asociada al CA donde los ambientes se representan como carpetas.

Algunas construcciones que utiliza el CA son heredadas del π-cálculo tales como el nombramiento, la restricción, la composición paralela, el proceso inactivo y la replicación. Por ejemplo, en la Figura 1 se muestra que la replicación de un proceso en el cálculo de carpetas se representa mediante una

fotocopiadora con el proceso que replica encima. Sin embargo, a diferencia del π- cálculo, los nombres del CA son nombres de ambientes en lugar de nombres de canales. Por lo tanto, para describir que un ambiente con el nombre n contiene un proceso P, se especifica como n[P].

Algunas de las primitivas que proporciona el CA son conocidas como capacidades (*capabilities*). Las capacidades son acciones que solicitan la movilidad a los ambientes. Existen tres capacidades principales: *enter*, *exit* y *open*. La capacidad *enter* solicita al ambiente que entre en otro ambiente de su mismo nivel jerárquico. La capacidad *exit* solicita al ambiente que salga de su ambiente padre. La capacidad *open* disuelve un ambiente dejando libres los procesos que estaban dentro de él.

IV. AMBIENT-PRISMA: INTRODUCCIÓN DEL CONCEPTO DE AMBIENTE EN PRISMA

En esta sección se describe cómo los conceptos de CA han sido integrados en la aproximación PRISMA para describir sistemas distribuidos y móviles. Con el objetivo de que los elementos arquitectónicos PRISMA puedan hacer uso del concepto de ambiente de CA, dicho concepto ha sido introducido en el metamodelo de PRISMA. Para ello, se han identificado las correspondencias entre el metamodelo de CA y el metamodelo de PRISMA y el concepto de ambiente se ha añadido como ciudadano de primer orden del metamodelo PRISMA.

En [2] se comenta que un ambiente puede ser visto como un componente software que proporciona movilidad y que posee una identidad propia en tiempo de ejecución, por lo que puede ser mantenible. En nuestro modelo, esto concuerda con la definición de un elemento arquitectónico PRISMA. Dichos elementos arquitectónicos pueden ser de dos tipos: componentes (elementos que ejecutan procesos computacionales) o conectores (elementos que coordinan dichos procesos). Sin embargo, un ambiente no puede definirse como un componente, conector, o una extensión de éstos, ya que su semántica es distinta. Los ambientes definen los límites donde se ubican los procesos computacionales (componentes), pero no son procesos computacionales, ni se encargan de coordinar su ejecución. Por estos motivos, el concepto de ambiente ha sido introducido en el metamodelo PRISMA como un nuevo tipo de elemento arquitectónico (ver figura 2) (homogéneo con los anteriores) que proporciona las características de movilidad, comunicación y replicación a los elementos arquitectónicos distribuidos, de acuerdo con CA.

Un ambiente puede estar compuesto por una colección de agentes locales (procesos computacionales) y de otros subambientes [2]. En PRISMA, los agentes locales corresponden a componentes coordinados a través de conectores, de forma que aquellos elementos arquitectónicos que tengan la misma localización estarán ubicados dentro del mismo ambiente. Además, la incorporación del concepto de subambiente en PRISMA permite modelar la jerarquía de sistemas distribuidos y móviles. De este modo, se definen los ambientes en PRISMA como elementos arquitectónicos complejos que permiten representar adecuadamente la ubicación de componentes y conectores.

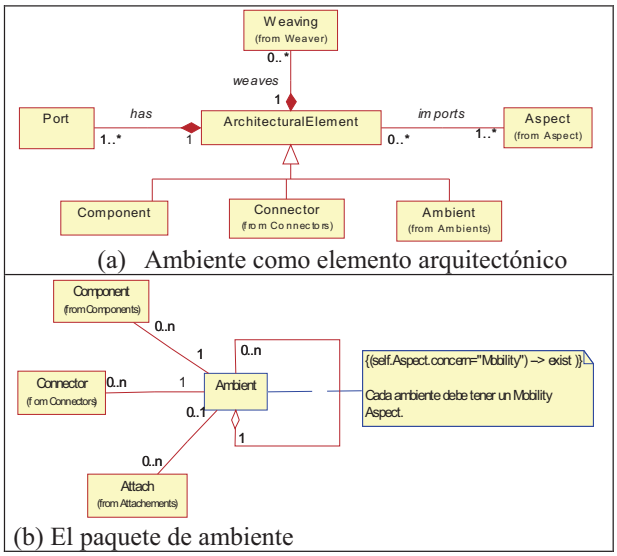


Fig. 2. Parte del Metamodelo de PRISMA con el concepto de Ambiente incluido

Un ambiente hereda todas las características de un elemento arquitectónico (su vista DSBC y su vista DSOA). La vista DSBC de un ambiente lo describe como una caja negra la cual se comunica con los otros ambientes mediante el uso de puertos que envían y reciben solicitudes de servicios.

Por otro lado, la vista DSOA de un ambiente define el conjunto de aspectos que, sincronizados mediante los weavings, definen su semántica. El ambiente usa diferentes aspectos para especificar los servicios que ofrece y requiere. Debido a que los ambientes permiten la movilidad, todos ellos importan un aspecto de movilidad que proporciona los servicios de movilidad a sus elementos arquitectónicos. De acuerdo con el CA, dicho aspecto proporciona las siguientes funcionalidades:

- Creación de nuevos subambientes: definición de los límites computacionales de un conjunto de elementos arquitectónicos.
- Servicio *exit*: salida de un subambiente de su ambiente padre.
- Servicio *enter*: entrada de un ambiente en otro ambiente del mismo nivel jerárquico
- Servicio *open*: eliminación de un subambiente sin destruir los elementos arquitectónicos que lo constituyen. Como resultado, los elementos arquitectónicos del subambiente destruido pasan a formar parte del ambiente padre.
- Servicio *photocopy*: replicación de elementos arquitectónicos.

La separación de la movilidad del resto de *concerns* a través de un aspecto mejora su mantenibilidad y reutilización, ya que no se encuentra entremezclada con el resto de *concerns* del ambiente. Además, el aspecto de movilidad es un aspecto

genérico que es reutilizado por todos los ambientes PRISMA. Por lo tanto, los ambientes son definidos importando dicho aspecto de movilidad y otros aspectos que definen otros *concerns*. Por ejemplo, un ambiente puede necesitar definir movilidad y políticas de seguridad para los procesos que contiene. Por tanto, el ambiente importaría el aspecto de movilidad genérico y un aspecto de seguridad.

En la figura 2(b) se muestra el paquete de ambientes introducido en el metamodelo PRISMA, que define las relaciones y restricciones respecto a otros conceptos del metamodelo. Por ejemplo, una de ellas es que todo ambiente debe tener un aspecto de movilidad.

V. CASO DE ESTUDIO: REPLICACIÓN DISTRIBUIDA EN UNA ARQUITECTURA DE LOS SISTEMAS DE TELE-OPERACIÓN

En esta sección se va a utilizar un caso de estudio de sistemas de tele-operación para ilustrar cómo se realiza la replicación distribuida de elementos arquitectónicos mediante AMBIENT-PRISMA.

Los sistemas de tele-operación son sistemas de control que dependen del software para realizar sus tareas. Generalmente, estos sistemas son robots que realizan actividades de alto riesgo y deben ser controlados por operarios desde zonas seguras. De ahí que en este tipo de sistemas software surja la necesidad de tener componentes distribuidos y de comunicar los componentes distribuidos con los operarios. Concretamente, el caso de estudio que se va a desarrollar en este artículo consiste en el soporte de la replicación del componente software *Operario*, para que otro operario pueda mandar órdenes al robot desde una PDA, a pesar de que inicialmente sólo había uno que lo realizaba desde una sala de control. Esta necesidad surge cuando las actividades son complejas y se necesita que más de un operario controle el robot desde diferentes lugares.

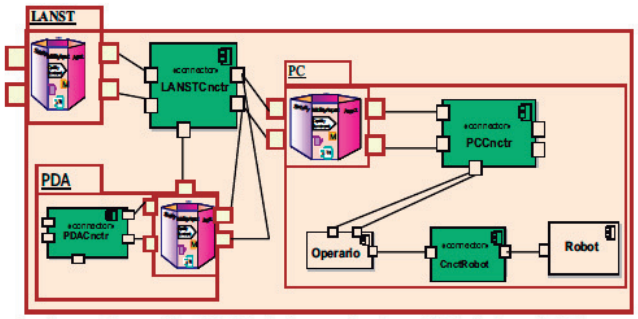


Fig. 3. La configuración inicial de la arquitectura del Robot modelada con ambientes

```
Component_type Operario
   Port
     MobilityPort: IMobility;
     ReplicatePort: IRep;
     FunctPort: IRobotCommands,;
   End_Port
   Import Distribution Aspect OpDistribution;
   Import Replication Aspect OpReplication;
   Import Functional Aspect OpFunct;
   Weavings
   OpDistribution.move(NewAmbient) afterif(ok=true)
                      OpReplication.replication(MyName, output ok);
   OpReplication.calculateRequestRate() after OpFunct.sendRobotCommands();
   End_Waevings;
End Component_type Operario;
```

Fig. 4. La especificación del componente Operario en LDA PRISMA

La figura 3 muestra la estructura inicial de la arquitectura del sistema de tele-operación. El ambiente *LANST* (la LAN del Sistema de Tele-operación) está compuesto por un ambiente *PDA* y un ambiente *PC*. Los componentes *Operario* y *Robot,* y su conector *CnctrRobot* están en el ambiente *PC*.

```
Replication Aspect OpReplication using IRep
 Attributes
  requestRate : nat(0);
     RequestRateLimit: nat NOT NULL;
 Services
   calculateRequestRate()
        Valuations
          [calculateRequestRate()] requestRate:=
                                          requestRate +1;
      initializeRequestRate()
        Valuations
          [initializeRequestRate()] requestRate:=0;
  out replicate (self.Name: String, output ok:bool);
 TRANSACTIONS replication(self.Name: String, output ok:bool)
   replication::= out replicate( self.Name, output ok)→
                  REPLICATION1;
   REPLICATION1::=[ok=true]initializeRequestRate();
 triggers
   replication(self.Name, output ok)
          when
           {requestRate >=requestRateLimit};
   … … …
End_Replication Aspect OpReplication
```

(a)

```
Distribution Aspect OpDistribution using IMobility
 Attributes
  location : loc;
 Services
   … …
   changeLocation(NewLocation:loc)
        Valuations
          [changeLocation()] location:=NewLocation;

   out exit (self.Name: String);
   out enter (self.Name: String, NewAmbient: loc);
   … …
  Transactions move(NewAmbient: loc)
      move::= out exit(self.Name)→MOVE1;
      MOVE1::= out enter(self.Name, NewAmbient)→MOVE2;
      MOVE2::= changeLocation(NewAmbient);
   … … …
End_Distribution Aspect OpDistribution
```

(b)

Fig. 5. El Aspecto de Replicación y el Aspecto de Distribución del Operario en el LDA PRISMA

El Operario es un componente (ver figura 4) que necesita ser replicado, y cuya réplica debe existir en una localización distinta a la del PC. Por este motivo, el componente Operario necesita tener definido un aspecto de replicación (OpReplication) (ver figura 5 (a)) y un aspecto de distribución (OpDistribution) (ver figura 5 (b)). Definiendo estos dos aspectos por separado se consiguen una serie de ventajas: el mantenimiento de las estrategias de replicación se mejora y la reutilización del aspecto de replicación se incrementa, ya que se puede utilizar siempre que se quiera replicar un componente de forma local y/o de forma distribuida; el carácter distribuido lo aporta el aspecto de distribución. Para que el Operario realice la replicación distribuida la sincronización (*weaving*) entre los aspectos OpReplication y OpDistribution es necesaria. En este caso, el *Weaving* indica que el Operario tiene que moverse (invocar al servicio *move*

del aspecto de distribución) después de que su réplica haya sido realizada (servicio *replication* del aspecto de replicación) (ver figura 4).

El *Operario* se replica cuando hay una sobrecarga de peticiones al robot. Por este motivo, el aspecto de replicación necesita saber el número de servicios que se le están mandando al robot a través del aspecto funcional. Para ello, se ha definido un weaving entre el aspecto funcional y el de replicación (ver figura 4). De esta forma, cuando la tasa llega a cierto límite se dispara la transacción *replication* (ver figura 5(a)). Esta transacción consiste en solicitarle al ambiente padre (*PC*) la necesidad de replicar el *Operario*.

```
Mobility Aspect Mobile using ICapability, IAccept
   …. ….
 Transactions in replicate(Requested: String, output ok):
  replicate::= out replInf(input Requested, output Instance,
               output AttachmentList[])→REPLICATE1;
  REPLICATE1::= photocopy(Instance)→ REPLICATE2;
  REPLICATE2::= photocopy(AttachmentList[]);
 Transactions in exit(Requested: String, NewAmbient: loc):
  EXIT := out isSon(input Requested, output isSonOK)→
          EXIT1;
  EXIT1::= {isSonOK==true} out checkTypeAmbient( input
          Requested, output  isTypeAmbient) →
          EXIT2;
  EXIT2::=  if(isTypeAmbient==false)then
          createAmbientFor ( input
          Requested, output RequestedAmbient) →
          EXIT3 else EXIT3 ;
  EXIT3::= out movingInf(input RequestedAmbient, output Type,
          output MobileInstance, output AttachmentList[])→
          EXIT4;
  EXIT4::= out accept(input Type, input MobileIntstance, input
         AttachmentsList, output Acceptance)→ EXIT5;
  EXIT5::= {Acceptance==true} out modifyAttachment(Requested)→
          EXIT6;
  EXIT6::= out destroy(RequestedAmbient)→EXIT7;
  EXIT7::= out removeAttachments(requestedAmbient);
   … …
End_Mobility Aspect Mobile
```

(a)

```
Ambient_type PC
 Import Mobility Aspect Mobile;
 Import Security Aspect Sec;
 Weavings
  Sec.CheckSecurity() before
          Mobile.exit(Requested, NewAmbient);
 Ports
    AccceptancePort: IAccept;
    DistServicesPort: ICall;
    ServicesPort: ICall;
    CapabilitiesPort: ICapability
 End_Ports

End Ambient_type PC;
```

(b)

Fig. 6. La especificación del aspecto de movilidad y el ambiente PC en el LDA PRISMA

La solicitud *replicate* se solicita al ambiente *PC* a través del conector *PCCnctr* que manda esta petición al *PC* (ver figura 3). Como consecuencia, el ambiente *PC* (ver figura 6(b)) ejecuta su transacción ***in replicate*** definida en el aspecto de movilidad *Mobile* (ver figura 6(a)). Esta transacción averigua el estado del *Operario* y las conexiones que tiene con los demás elementos para que haga una réplica del Operario (una instancia con el mismo estado) y además con sus mismas conexiones. La figura 7 muestra el resultado de la solicitud de replicación local del componente *Operario* a su ambiente *PC*.

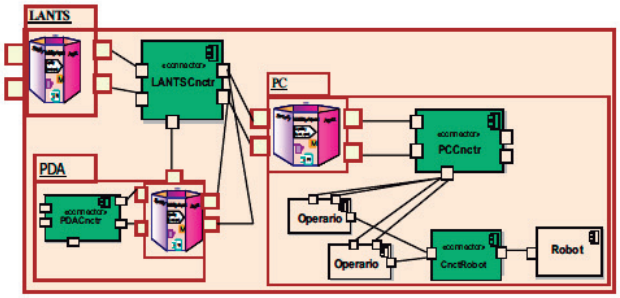
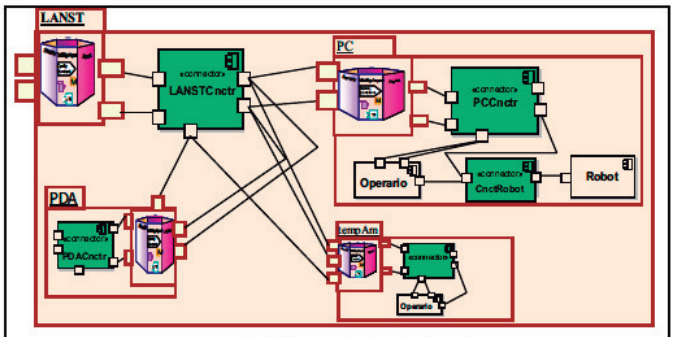
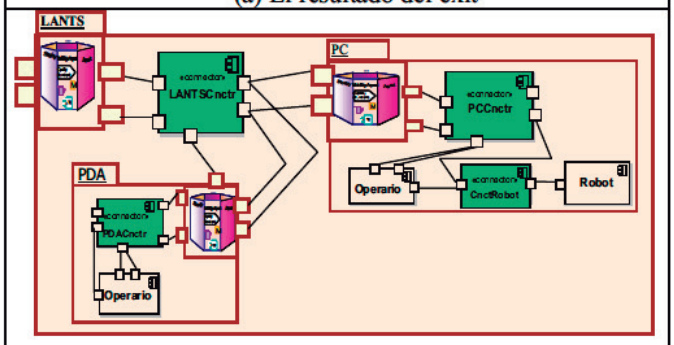


Fig. 7. La replicación del componente *Operario* de la Figura 6

Como la replicación es distribuida, el weaving asociado a la transacción *replication* del *Operario* original dispara el servicio *move* del aspecto *OpDistribution* (ver figura 4). El *move* solicita la capacidad *exit* (ver figura 5(b)) para que el *Operario* original pueda salir del ambiente *PC* y la capacidad *enter* para que después pueda entrar en el ambiente *PDA*.



(a) El resultado del exit



(b) La configuración final después de la replicación y movilidad del operario

Fig. 8. La movilidad del Operario

El conjunto de servicios que componen la transacción *exit* en el aspecto de movilidad del *PC* (ver figura 6(a)) se ejecutan en el siguiente orden. Primero, se comprueba si el elemento que ha solicitado el *exit* (*Operario*) es uno de los hijos de *PC*. Si lo es, entonces se comprueba si el solicitante es un ambiente y en caso negativo se crea un ambiente que encapsula el elemento para que éste pueda moverse (*tempAmb*). Una vez encapsulado el *Operario* en *tempAmb*, se recoge la información necesaria para la salida del ambiente *PC*: el estado del operario y de sus conexiones. Después, se le pregunta al ambiente padre (*LANST*) si puede aceptar al ambiente *tempAmb*. El resultado de la ejecución del *exit* tras la

aceptación de *tempAmb* en *LANST* se muestra en la figura 8(a).

Finalmente, la figura 8 (b) muestra el componente *Operario* en el *PDA*. Éste es el resultado después de que, a través de *tempAmb,* el *Operario* solicite a *LANST* entrar (*enter*) al ambiente *PDA*. Entonces, el ambiente *LANST* comprueba si *PDA* es local a él y le pregunta si puede aceptar la solicitud de *tempAm*. *PDA* acepta a *tempAm* y lo abre (capacidad *open*), dejando la réplica del componente *Operario* en *PDA*.

La descripción de este caso de estudio muestra cómo las características distribuidas, replicables y móviles de un sistema de teleoperación pueden ser descritas en AMBIENT-PRISMA. La especificación anterior se beneficia de los conceptos del CA para definir formalmente y de forma precisa las características de movilidad y replicación del operario del sistema. Dichas primitivas del CA se han especificado de forma completa mediante el LDA de PRISMA preservando su independencia de la tecnología. Además, la especificación anterior potencia la reutilización y mejora el mantenimiento de los concerns de distribución, movilidad y replicación utilizando el concepto de aspecto que ofrece el LDA de PRISMA.

VI. Conclusiones y Trabajos Futuros

En este trabajo se ha presentado cómo AMBIENT-PRISMA es capaz de dar soporte a la replicación distribuida y local de sistemas complejos, distribuidos y móviles desde las primeras etapas del ciclo de vida software. Dicho enfoque combina el enfoque de desarrollo PRISMA con el CA con el objetivo de obtener las ventajas de ambos enfoques: la especificación de arquitecturas software orientadas a aspectos y la precisión de un formalismo como el CA que ofrece la posibilidad de expresar la topología de una red, sus fronteras de computación y sus políticas de movilidad.

La replicación distribuida que ofrece AMBIENT-PRISMA es posible porque el concepto de ambiente se ha introducido en el metamodelo PRISMA para poder mover elementos arquitectónicos. El ambiente en PRISMA es un nuevo tipo de elemento arquitectónico que se caracteriza por contener otros elementos arquitectónicos y subambientes, así como de importar un aspecto de movilidad genérico que implementa y ofrece las capacidades del CA. Por otro lado, se han definido los tipos de aspecto de replicación y distribución para especificar la replicación independientemente de si es local o distribuida y las necesidades de movilidad de los componentes y conectores, respectivamente. De este modo, la replicación distribuida se simula mediante la sincronización de estos dos tipos de aspectos.

El trabajo futuro consiste en la introducción de estos conceptos dentro del entorno de desarrollo PRISMA para modelar gráficamente arquitecturas software orientadas a aspectos, móviles, distribuidas y replicables, generar su código automáticamente y ejecutarlo.

VII. Referencias

[1] J. Perez, N. Ali, J.A. Carsí, I. Ramos, "Dynamic Evolution in Aspect Oriented Architectural Models", *in Proc. Of 2nd European Workshop on Software Architecture*, Pisa, June 2005 © Springer LNCS.3527, pp. 59 76.

[2] Cardelli, L. "Abstractions for Mobile Computation." In Vitek, J. and (Eds.), C. J., Secure Internet Programming: Security Issues for Distributed and Mobile Objects, LNCS 1603, Springer, pp. 51 94.

[3] Szyperski, C., *Component Software: Beyond Object Oriented programming*, ACM Press and Addison Wesley, New York, USA, 2002.

[4] Aspect Oriented Software Development, http://aosd.net

[5] J. Pérez, N. Ali, J. A. Carsí., I. Ramos, "Designing Software Architectures with an Aspect Oriented Architecture Description Language", *in Proc 9th International Symposium on Component-Based Software Engineering (CBSE)*, Västerås , Stockholm, Sweden, June 29th 1st July 2006, pp. 123 135.

[6] C. E. Cuesta, P. Romay, P. De La Fuente, M. Barrio Solórzano, "Architectural Aspect of Architectural Aspects", *in Proc. Of 2nd European Workshop on Software Architecture*, Pisa, June 2005 © Springer LNCS.3527, pp. 247 262.

[7] J. Magee, N. Dulay, S. Eisenbach, J. Krammer, "Specifying Distributed Software Architectures" *in Proc 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, 1995, pp 137 153.

[8] R. Milner, J. Parrow, D. Walker. "A calculus of mobile processes", Parts 1 2. Information and Computation, 100(1), 1992, pp. 1 77.

[9] P. Ciancarini., C. Mascolo "Software Architecture and Mobility", *in Proc. 3rd Int. Software Architecture Workshop (ISAW-3*), November, 1998.

[10] N. Medvidovic, M. Rakic, "Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility", *in Proc. Software Engineering and Mobility Workshop*, Toronto, Canada, May 2001.

[11] A. Lopes, J.L. Fiadeiro, M. Wermelinger, "Architectural Primitives for Distribution and Mobility", *in Proc. 10th Symposium on Foundations of Software Engineering*, SIGSOFT FSE 2002, pp. 41 50.

VIII. Biografía



Nour Ali Irshaid es licenciada en Ciencias de la Computación de la Universidad de Bir Zeit (Palestina) en el año 2002. Actualmente, es estudiante de doctorado en el Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia. Sus áreas de interés son los sistemas distribuidos y móviles, la Generación Automática de Código, Arquitecturas Software, Desarrollo de Software Orientado a Aspectos. Ha participado en los proyectos de investigación PRISMA y en el proyecto PRISMA MODEL COMPILER, financiado por Microsoft Research Cambridge.



Jennifer Perez Benedi es Ingeniera en Informática por la Universidad Politécnica de Valencia. Obtuvo el titulo de Doctor en el año 2006 por la misma universidad. . Ha participado en los proyectos de investigación PRISMA y en el proyecto PRISMA MODEL COMPILER, financiado por Microsoft Research Cambridge. Sus áreas de interés son la Evolución Dinámica del Software, la Generación Automática de Código, Arquitecturas Software, Desarrollo de Software Orientado a Aspectos.



Cristóbal Costa Soria es estudiante de doctorado en el Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia. En el año 2005 obtuvo el título de Ingeniero en Informática por la Universidad Politécnica de Valencia, fue becario Socrates/Erasmus en Finlandia y realizó prácticas de empresa en el centro de investigación europeo "European Synchrotron Radiation Facility" (ESRF, Grenoble, Francia). Actualmente, sus tareas de investigación se realizan en el seno del grupo de

investigación "Ingeniería del Software y Sistemas de Información" (ISSI) como becario de Formación de Personal Investigador (FPI) de la Generalitat Valenciana. Sus áreas de interés son la Evolución Dinámica del Software, la Generación Automática de Código, Arquitecturas Software, Desarrollo de Software Orientado a Aspectos y Tecnologías Middleware. Ha participado en los proyectos de investigación PRISMA y en el proyecto PRISMA MODEL COMPILER, financiado por Microsoft Research Cambridge.



Isidro Ramos Salavert es Licenciado en Ciencias Físicas por la Universidad de Valencia (España) en 1966, Licenciado en Informática por la Universidad Politécnica de Madrid (España) en 1972. Doctor en Ciencias Físicas por la Universidad Complutense de Madrid (España) en 1971. En la actualidad es profesor Catedrático de Universidad en la Universidad Politécnica de Valencia. Sus areas de interés son el desarrollo de software basado en componentes y orientado a aspectos, las arquitecturas software y la gestión de modelos.



Jose Angel Carsí Cubel es Licenciado en Informática por la Universidad Politécnica de Valencia (España) en 1996. Doctor en Informática por la UPV en 1999. Profesor Titular de Universidad del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia. Sus áreas de interés son la evolución del software, la gestión de modelos, los modelos de componentes, y los lenguajes de descripción de arquitecturas.

Mobile Ambients in Aspect-Oriented Software Architectures

Nour Ali, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose A. Carsí

Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n
E-46022 Valencia, Spain
{ nourali, jeperez, ccosta, iramos, pcarsi }@dsic.upv.es

Abstract. Nowadays, distributed and mobile systems are acquiring importance and becoming widely extended for supporting ubiquitous computing. In order to develop such systems in a technology-independent way, it is important to have a formalism that describes distribution and mobility at a high abstraction level. Ambient Calculus is a formalism that allows the representation of boundaries where computation occurs. Also, distributed and mobile systems are usually difficult to develop as they need to take into account functional and non-functional requirements and reusability and adaptability mechanisms. In order to achieve these needs it is necessary to separate the distribution and mobility concerns from the rest of the concerns. PRISMA is an approach that integrates the advantages of Component-Based Software Development and Aspect-Oriented Software Development for specifying software architectures. In this paper, we describe how our work combines Ambient Calculus with PRISMA to develop distributed and mobile systems gaining their advantages.

1. Introduction

In the last few decades, the information society has undergone important changes. New technologies have become part of our daily life and the Internet has been established as a framework for global knowledge. For these reasons, two important ideas have been arisen: the world is considered as a whole unit with no boundaries, and people work in a collaborative way without meeting physically. These ideas have created the need for current software development processes to deal with complex structures, new non-functional requirements, dynamic adaptation, and new technologies. In addition, most software systems require the capability to work with different devices (PCs, laptops, PDAs, smart phones, etc) through communication networks in a distributed and secure way. As a result, software development processes must also take into account the distributed, ubiquitous and mobile nature of software systems.

The development of distributed, ubiquitous and mobile software systems is a difficult task, especially if these characteristics are to be considered from the beginning of the software life cycle. Currently, decisions about these characteristics

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Mobile Ambients in Aspect-Oriented Software Architectures. Proc. of the IFIP Working Conference on Software Engineering Techniques - SET 2006. 2006.

are usually postponed to late stages of the software life cycle (design and implementation). As a result, there is a loss of traceability, and the system is subject to a specific technological platform (such as CORBA [1] or .NET Remoting [2]). As a result, the development of systems of this kind introduces important challenges such as: how to specify distribution and mobility features in a technology-independent way, how to consistently manage a distributed state, how to support non-functional requirements such as security or fault tolerance.

Software Architecture is considered to be the bridge between the requirements and implementation phases of the software life cycle. Software Architectures describe the structure of software systems in terms of computational (components) and coordination (connectors) units of software. Architecture Description Languages (ADLs) specify the functional and coordination properties of these software units in a formal way. However, current ADLs do not provide constructs for describing distribution or mobility features in an abstract way.

A formalism that provides mechanisms to describe distribution and mobility properties is Ambient Calculus (AC) [3]. AC introduces the concept of ambient, which represents boundaries where computation occurs. Ambients can model the location hierarchy encountered in distributed systems and model the mobility as the crossing of the locations boundaries.

PRISMA [4] is an approach that integrates the advantages of Component-Based Software Development (CBSD) [5] and Aspect-Oriented Software Development (AOSD) [6] to specify software architectures. This approach has a meta-model [4], formal Aspect-Oriented Architecture Description Language (AOADL) [7], and a framework [8].

In this paper, we combine the PRISMA approach and the AC in order to deal with the specification of distributed and mobile features from the beginning of the software life cycle in a technology-independent way. In this work, ambients are specified as architectural elements that use separation of concerns (aspects) to describe their functionalities.

The paper is structured as follows: Section 2 presents related works performed in the area of distribution at an architectural level. Section 3 presents the PRISMA approach and the motivation for the work presented in this paper. Section 4, gives an overview of AC. Section 5, introduces how the PRISMA approach combines ambients. Finally, Section 6 presents conclusions and further works.

2. Related Work

One of the reasons why software architectures emerged was to simplify the construction of dynamic distributed systems. However, at the present time, few ADLs support the specification of distributed systems properties. The first research that provided significant results in distributed software architectures was carried out in the Darwin ADL [9]. Darwin uses π-calculus [10] to define the semantics of distributed message-passing. It builds architectures by defining composite components that are bound and given locations at instantiation time. Darwin has also been used in the CORBA environment to specify the overall architecture of component-based

applications [11]. However, in the literature, we have not found new advances to Darwin in constructing software architectures with mobile and replicable components. As Darwin is based on π-calculus only, mobility can only be simulated by the movement of channels. It lacks primitives to express the movement of crossing boundaries.

The work in [12] states that an ADL should consider features such as composition, reusability, and configuration in order to specify dynamic distributed software architectures. It presents a configuration language that describes a method for a reconfiguration model at run-time. However, the reconfiguration model is not formal. Moreover, it neglects a distribution model for specifying distributed message-passing among components and connectors.

The works of Mascolo and Ciancarini [13] [14] introduce MobiS, which is a specification language that is based on a tuple-space model that specifies coordination by multiset rewriting. MobiS can also be used to specify architectures containing mobile components. However, it does not specify the mobility concern separately from the rest of the functionalities of the software architectures, reducing reusability and adaptability to changes.

The ADL C2Sadel has adapted a style to support both distribution and mobility. The style [15] provides software connectors that are able to move components. It also has an implementation infrastructure to support this architectural style. However, this approach has the drawback that there is no separation between coordination and distribution. Therefore, the components are the only architectural elements that are mobile while the connectors are static.

The work of Lopes in [16] describes the semantics of externalizing a distribution dimension in order to support distribution and mobility for software architectures. This distribution dimension is very similar to a connector, but instead of containing the business logic, it controls the rules for mobility and location. In this way, a separation between computation, coordination and distribution is achieved. A difference between our work and Lopes's work is that our work defines the semantics of distribution and mobility by using Ambient Calculus. This allows our approach to have an explicit primitive to represent a location with boundaries allowing the specification of security and authentication.

3. PRISMA Distribution and Mobile Model

The PRISMA model [4] allows the definition of software architectures of complex software systems by integrating the AOSD and the CBSD. PRISMA uses the AOSD to separate the crosscutting concerns (distribution, security, context-aware, coordination, etc.) of architectures in aspects. The PRISMA architectural elements are specified using aspects that define their behaviour. As a result, an architectural element (components and connectors) can be viewed as a prism where each side of the prism is an aspect (*white box view*). In addition, an architectural element encapsulates its functionality and publishes a set of services that it offers to the rest of the architectural elements (*black box view*) (see Fig. 1).

There are two kinds of architectural elements: components and connectors. A component is an architectural element that captures the functionality of software systems and a connector is an architectural element that acts as a coordinator among other architectural elements. Components and connectors are formed by a set of aspects, the weaving relationships among these aspects, and the ports that offer and request services.

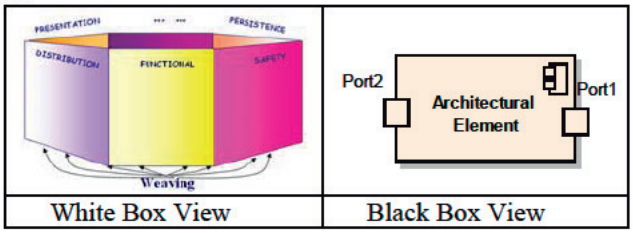


Fig. 1. Views of an Architectural Element

Weavings indicate that the execution of an aspect service can trigger the execution of services in other aspects. Weavings are the glue of the aspects of an architectural element. This glue is defined using temporal operations called weaving operators. Initially, the weaving operators that PRISMA provides are *after*, *before*, *around*, *afterif, beforeif*, *and insteadif*. For example, if a weaving with the *after* operator is specified between service s1 of aspect A1 and service s2 of aspect A2, this means that s2 of A2 is executed after s1 of A1.

It is important to emphasize that connectors do not have the references of the components that they connect and vice versa. Thus, architectural elements are reusable and unaware of each other. This is possible due to the fact that the channels (*attachments)* defined between components and connectors have their references, instead of architectural elements. Attachments are the channels that enable the communication between components and connectors. Each attachment is defined by attaching a component port with a connector port (represented as lines in Fig. 2).

However, when we applied PRISMA to a real case study such as the tele-operated *TeachMover* robot local communication was a limitation. Tele-operation systems are control systems that depend on software to perform their operations. They are usually robots that perform high-risk activities. For this reason, they must be controlled by operators from safe areas. As a result, the need to locate components in different places (nodes) as well as to communicate the distributed software components of the operator and the robot emerged.

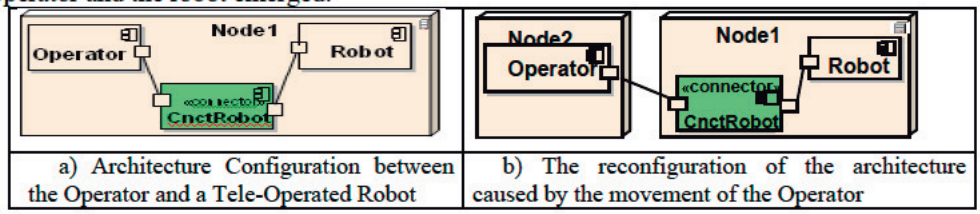


Fig. 2. Mobility of the Operator in a Tele-Operated System

Mobility is also a characteristic that is fundamental in distributed and dynamic systems, where the topology of the architecture can change at run-time. For example,

in the tele-operated system, the mobility requirement emerges to be able to move the operator to different places (nodes). This mobility is necessary to allow the operator send commands to the robot from different places, maintaining the information of the operator component consistent (see Fig. 2).

As Fig. 2 illustrates, mobility is the process of transferring a component instance from one node to another one. Moreover, the transfer process must ensure that the transferred component instance continues its execution at the target node, conserving its state and maintaining the same execution point.

PRISMA has been adapted to support distribution and mobility properties [17] in order to be applied to real case studies. Distribution is supported in PRISMA by introducing the following properties into the model:

1) The use of attachments: Attachments store, not only the references of the architectural elements that they are connecting, but also the locations of these architectural elements (nodes). In this way, the reusability of architectural elements is preserved, and distributed communication is enabled. As a result, architectural elements are unaware of the distributed or local nature of the others.

2) The use of a Distribution Aspect: The distribution aspect specifies the features and strategies that are related to the distributed behaviour of a PRISMA architectural element. It specifies the site where the architectural element is located and indicates when an element needs to be moved.

This distribution model was initially implemented in the PRISMANET middleware [8] and has been validated using case studies where distribution properties are required. However, a model that includes an explicit primitive for supporting locations as boundaries, describes the location hierarchies and supports the mobility of elements by the crossing of boundaries is richer. Therefore, we have combined the PRISMA model and AC.

4. Ambient Calculus

Ambient Calculus [3] (AC) is a process algebra that extends π-calculus [11] in order to introduce the concept of ambient. An ambient is a bounded place where computation occurs. Thus, an ambient can be anything with a boundary such as a laptop, a web page, a folder, etc. Each ambient has a set of running computations that can control it. These are responsible for moving an ambient. In addition, an ambient can contain other subambients that have running computations.

Thus, mobility is performed at an ambient level, i.e. ambients are mobile. Also, mobility is performed by crossing boundaries of ambients. AC provides mobility and local communication primitives. These primitives can be expressed in a textual syntax and in a graphical syntax which is called Folder Calculus [20] (see Fig. 3). Folder Calculus is a graphical metaphor for AC where ambients are visually represented as folders.

AC uses some of the constructs inherited from π-calculus such as naming, restriction, parallel processes, inactive process and replication. However, the names in AC are names of ambients instead of names of channels as in π-calculus. Therefore,

in order to syntactically write that an ambient with name *n* has process *P*, it is written as *n[P]*.

Textual Syntax	Visual Syntax	Comments	Textual Syntax	Visual Syntax	Comments
(νn)P		New name *n* in a scope *P*.	0		Inactive process (often omitted).
			!P		Replication of *P*.
n[P]		Folder (ambient) of name *n* and contents *P*.	⟨M⟩		Output *M*.
M.P		Action *M* followed by *P*.	(n).P		Input *n* followed by *P*.
P | Q	P Q	Two processes in parallel. (Visually: contiguously placed in 2D.)	(P)		Grouping

Fig. 3. The Textual and Visual Syntax of Ambient Calculus constructs

Some of the primitives that AC provides are called capabilities. Capabilities are actions that can be performed on ambients. There are three main types of capabilities: enter, exit and open capabilities. The enter capability orders an ambient to enter another ambient on its same hierarchy level (see Fig. 4). The exit capability orders an ambient to exit its parent ambient. The open capability dissolves an ambient leaving the processes that were in it.



Fig. 4. Applying the enter capability to the ambient n

4. AMBIENT-PRISMA: Combining Ambient Calculus and the PRISMA Approach

This section presents how the AC concepts are integrated to the PRISMA approach in order to describe distributed and mobile systems. To allow PRISMA architectural elements to make use of the ambient concept of AC, the ambient construct must be included in the PRISMA meta-model. Therefore, some mappings between the AC meta-model and the PRISMA meta-model have been identified.

In [18], it is discussed that an ambient can be seen as a software component that offers mobility and that it has a proper identity at run-time so that it can be maintainable. In our model, this corresponds to a PRISMA architectural element. Since PRISMA architectural elements are components and connectors, an ambient cannot be a PRISMA Component because a PRISMA component performs the computations. Nor can the ambient be a PRISMA Connector because a PRISMA Connector coordinates computations. Therefore, in the PRISMA meta-model an ambient is introduced as a new type of architectural element (see Fig. 5) that is responsible for providing mobility sevices to distributed architectural elements. As a

result, an ambient inherits all the characteristics of a PRISMA architectural element (its CBSD view and its AOSD view) and provides its proper semantics. Fig. 6 shows the graphical representation of a PRISMA ambient. The graphical representation preserves the folder calculus representation of an ambient. The Ambient CBSD view describes it as a black box where it communicates with others by using ports that send and receive invocations of services.

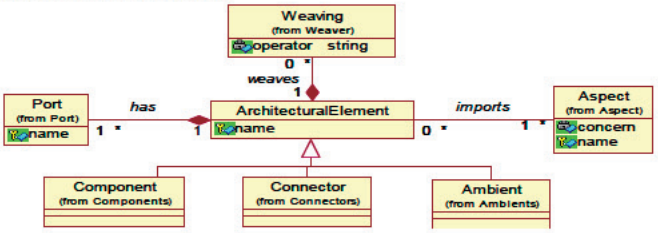


Fig. 5. Including the Ambient as another architectural element

An ambient has a collection of local agents and can also have other subambients [18]. In PRISMA, the local agents correspond to components that are coordinated using connectors. Ambients in PRISMA are complex architectural elements that represent the places where components, connectors and other ambients are located. In addition, by allowing an ambient to have other ambients inside it, the hierarchy of distributed and mobile systems can be modelled in PRISMA.

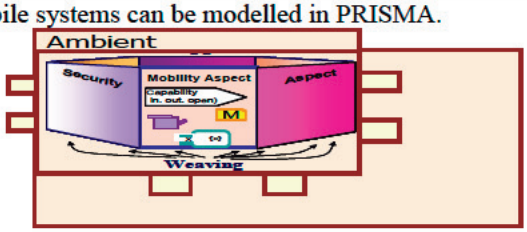


Fig. 6 A PRISMA Ambient with CBSD and AOSD views

The AOSD view describes the PRISMA ambient with a set of aspects that can be weaved. The ambient uses different aspects to specify the services it offers and requests. As ambients are responsible for the mobility concern, all ambients must have the Mobility Aspect to provide mobility services to their local architectural elements.

The Mobility Aspect specifies the following ambient functionalities:

- It allows an ambient to offer the exit service to its subambients that need to exit from it. (The specification of the AC exit capability).
- It allows an ambient to offer the enter service to its subambients that need to enter other subambients. (The specification of the AC enter capability).
- It allows an ambient to create subambients. (The specification of the AC restriction).
- It allows an ambient to accept a new ambient in it from external ambients.
- It allows an ambient to execute the open service. The open service allows a subambient to be destroyed a local ambient without destroying its local architectural elements. As a result, the architectural elements of the destroyed

subambient become to form part of the ambient. (The specification of the AC open capability)

The separation of the Mobility Aspect concern from the rest of concerns provides a better maintainability of these functionalities because they are not scattered through the ambient specification. In addition, this is a generic aspect that must be reused by all PRISMA ambients. As a result, ambients are defined by importing the generic mobility aspect and adapting it to the software system needs through weavings. For example, a LAN ambient may need some security policies that are different from a PC ambient inside of the LAN. Therefore, both the LAN and PC ambient import the same Mobility Aspect, but the Mobility Aspect is weaved with different security aspects.

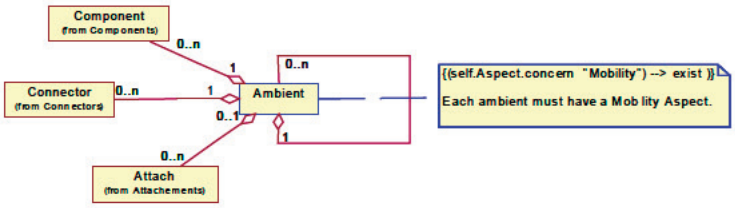


Fig. 7. The Ambient Package in the PRISMA meta-model

In order to introduce the concepts that describe a PRISMA ambient, an Ambient Package has been defined in the PRISMA meta-model(see Fig. 7). This package contains the relationships and constraints that an ambient has with other meta-model concepts. An Ambient can contain Components, Connectors, Attachments, and other ambients. A constraint is specified in the Object Constraint Language in order to indicate that an Ambient must have a Mobility Aspect.

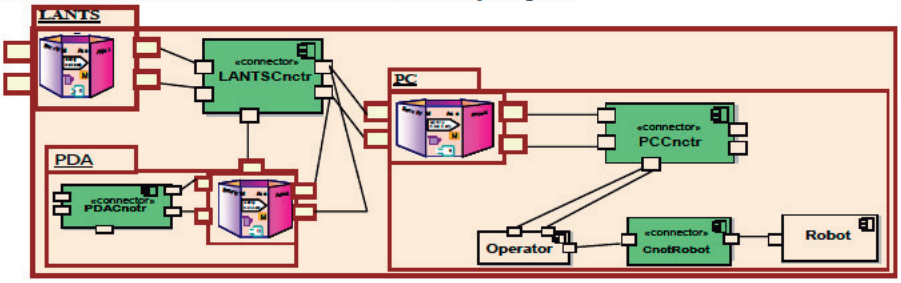


Fig. 8. The Initial Configuration of the Robot software architecture modeled with Ambients.

We are going to use the tele-operation system example to illustrate how a distributed and mobile system is specified in PRISMA after introducing the ambient architectural element. In the example, the operator is a mobile component that can move from a PC to a PDA to be closer to the robot. Fig. 8 shows the distributed hierarchy of the tele-operation software architecture. It shows that the *LANTS* ambient (the LAN of the Tele-operation System) consists of a *PDA* ambient and a *PC* ambient. The *Operator* component, *Robot* component and their connector (*CnctrRobot*) are initially located in the *PC* ambient. Using the ambient calculus syntax, this is written as *LANTS[PDA[]| PC[Operator[out PC, in PDA] , CnctRobot, Robot]*.

Every ambient has services that are offered to its local architectural elements and services that are offered to the exterior. As a result, some of the ambients ports in Fig. 8 are only internally connected through attachments to a connector (e.g. the *PCCnctr* of the *PC*) that synchronizes the ambient with its local architectural elements.

Fig. 9 (b) shows how the *Operator* Component is specified in the PRISMA ADL. As the *Operator* is a distributed component, it imports a predefined Distribution Aspect *OpDistribution*, specified in Fig. 9(a), which defines a distributed behaviour. The *Operator* has three ports: *ExitingPort* to request an *exit*, *EnteringPort* to request an *enter*, and *FunctPort* to send commands to the robot. The *OpDistribution* specifies the *move*. *Move* indicates that the movement of the element that imports this aspect needs to *exit* from its parent ambient and then *enter* to another ambient. For this reason, requests for *enter* and *exit* are made to other architectural elements (***out*** = client behaviour). For example in Fig. 8, to move the *Operator* from the *PC* to the PDA, the Operator makes a request to *exit* the *PC* from the *PC*. Fig. 8 also shows that the ports *ExitigPort* and *EnteringPort* are connected to the *PCCnctr* in order to be synchronized with the *PC* ambient. *FunctPort* is connected to *CnctRobot* to be synchronized with the *Robot.*

```
Distribution Aspect OpDistribution
                  using IExiting, IEntering
  Services
    … …
    out exit (MyName: String);
    out enter (MyName: String, NewAmbient: loc);
    … …
  Transactions move(NewAmbient: loc)
      Exiting= out exit(MyName).Entering;
      Entering= out enter(MyName, NewAmbient);
   … … …
End Distribution Aspect OpDistribution
```

(a)

```
Component_type Operator
  Import Distribution Aspect OpDistribution;
  Import Functional Aspect OpFunct;
  Port
     ExitingPort: IExiting;
     EnteringPort: IEntering;
     FunctPort: IRobotCommands,;
   End_Port
End Component_type Operator;
```

(b)

Fig. 9. The Operator Distribution Aspect and Component specified in the ADL

```
Mobility Aspect Mobile
                using IExiting, IEntering, IAccepting
   …. ….
 Transactions in exit(Requested: String, NewAmbient: loc):
  EXIT ::= out isSon(input Requested, output isSonOK)→
           EXIT1;
  EXIT1::= {isSonOK==true} out checkTypeAmbient( input
           Requested, output  isTypeAmbient) →
           EXIT2;
  EXIT2::=  if(isTypeAmbient==false)then
           createAmbientFor ( input
           Requested, output RequestedAmbient →
           EXIT3 else EXIT3 ;
  EXIT3::= out movingInf(input RequestedAmbient, output Type,
           output MobileInstance, output AttachmentList[])→
           EXIT4;
  EXIT4::= out accept(input Type, input MobileIntstance, input
          AttachmentsList, output Acceptance)→ EXIT5;
  EXIT5::= {Acceptance==true} out modifyAttachment(Requested)→
           EXIT6;
  EXIT6::= out destroy(RequestedAmbient)→EXIT7;
  EXIT7::= out removeAttachments(requestedAmbient);
   … …
End_Mobility Aspect Mobile
```

(a)

```
Ambient_type PC
 Import Mobility Aspect Mobile;
 Import Security Aspect Sec;
 Weavings
  Sec.CheckSecurity() before
           Mobile.exit(Requested,
NewAmbient);
 End_Weavings

  Ports
     AccceptancePort: IAccept;
     DistServicesPort: ICall;
     ServicesPort: ICall;
     CapabilitiesPort: ICapability
  End_Ports

End Ambient_type PC ;
```

(b)

Fig. 10. The Mobility Aspect and the PC Ambient specified in the ADL

Fig. 10(a) shows a fragment of the Mobility Aspect *Mobile* that all ambients import. It shows how the *exit* capability is mapped in PRISMA. Fig. 10(b) shows the

specification of the *PC* ambient. It shows that the *PC* imports the behaviour that the *Mobile* aspect defines. It also shows that it imports a Security Aspect *Sec*. In the **Weavings** section, a weaving is specified to indicate that a security rule must be checked in the *Sec* aspect *before* the exit is executed in the *Mobile* aspect.

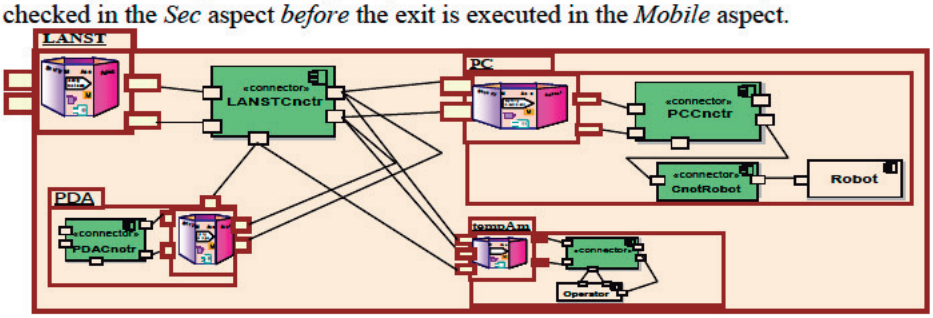


Fig. 11. The new configuration of Fig. 8 after the execution of the *exit*

The *exit* in Fig. 10(a) is specified as a transaction in the ***Transactions*** section. The *exit* has a server behaviour in the ambient that imports the aspect, that is, other architectural elements are going to request it (***in***=server behaviour). Using the example in Fig. 8, the *Operator* would be the element that makes an *exit* request to the ambient. Then the *exit* ***Transaction*** consists of a set of services. First, it checks if the requested element (*Operator*) is one of the PCs children (is one of its local elements). If it is, then the *exit* checks if the requester is an ambient or not. If it is not an ambient, then an ambient is created to encapsulate the element. The creation of an ambient is necessary due to the fact that ambients are the only architectural elements that can be mobile. Then the information needed for the exit of the *Operator* is collected: the state of the *Operator* and its attachments. The *exit* transaction then asks the parent ambient (*LANTS*) if it can accept the *Operator's* ambient (*tempAm*) and sends the needed information. If *LANTS* accepts the *tempAm*, for each attachment between the *Operator* and other architectural elements, a new attachment is created between the *PCCnctr* and those local architectural elements that are connected to *Operator*. Fig. 11 shows the new attachment that is created between *CnctRobot* and *PCCnctr* in place of the attachment between *CnctRobot* and *Operator*. Then the *PC* ambient deletes the *Operator* and all its attachments. Fig. 11 shows the result of the software architecture configuration after executing the *exit* transaction.

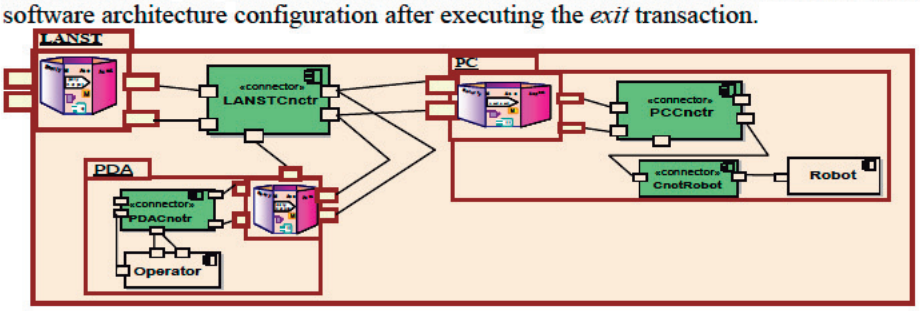


Fig. 12. The configuration of the architecture when the *Operator* reaches *PDA*.

Finally, Fig. 12 shows the *Operator* component in the *PDA*. This is possible after the *Operator* in *tempAm* in Fig. 11 requests the *LANTS* to enter PDA. Then, the *LANTS* checks if the PDA is local to it and requests the PDA to accept the *tempAm*. The *PDA* accepts *tempAm* and opens it, leaving the *Operator* in *PDA*.

The tele-operation system specification shows how its distributed and mobile properties can be described. The previous specification benefits from the concepts introduced in AC; thus the mobility of the *Operator* is specified in a formal way thanks to the AC capabilities. Also, the AC primitives can be completely specified by the PRISMA ADL in a technology-independent way. In this way, the ambient functionalities can benefit from the reusability and maintainability that the AOSD and CBSD provide.

5. Conclusions and Further Work

In this work, we have presented an approach to represent complex, distributed and mobile systems in a technology-independent. Our model combines the PRISMA approach with the AC formalism, which provides the following advantages: 1) It can describe a complex system in terms of computational, coordination and distribution and mobility units on different levels of abstraction. In this way, a system is built by reusing and adapting these separated units achieving a higher level of maintainability. 2) It can also describe the specific issues of current distributed systems such as the network topology and security.

We have introduced the ambient concept in the PRISMA meta-model as a new architectural element that can contain several computation and coordination processes (components and connectors) or other subambients. The capabilities provided by an AC ambient are offered in PRISMA by an ambient-specific aspect called the Mobility Aspect. Another aspect, the Distribution Aspect, manages the location of an architectural element and defines how and when ambient capabilities can be executed. A Security Aspect can be added to an ambient in order to provide security mechanisms. Mobility, distribution and security concerns are specified separately from other functional and non-functional requirements, thereby increasing reusability and adaptability to changes.

In the near future, we are going to introduce these concepts into the PRISMA tool to be able to model and execute mobile distributed software architectures. This will be done in three stages: first, ambients will be introduced in the PRISMANET middleware [8] to execute these concepts; second, ambient graphical metaphor and code templates will be introduced in the modelling framework; third, the implementation will be validated by modelling and executing a complex, distributed, and mobile case study.

References

[1] CORBA Official Web Site of the OMG Group, http://www.corba.org/

[2] Microsoft .Net Remoting: A Technical Overview, http://msdn microsoft.com/library/default.asp?url=/library/enus/dndotnet/html/hawkremoting.asp

[3] Cardelli, L., Gordon, A. D. "Mobile Ambients", Foundations of Software Science and Computational Structures: First International Conference, FOSSACS '98, LNCS 1378, Springer, 1998, pp. 140-155.

[4] Perez, J., Ali, N., Carsí, J.A., Ramos, I. "Dynamic Evolution in Aspect-Oriented Architectural Models", European Workshop on Software Architecture, Pisa, June 2005 © Springer LNCS vol n.3527.

[5] Szyperski, C., *Component Software: Beyond Object Oriented programming*, ACM Press and Addison Wesley, New York, USA, 2002.

[6] Aspect-Oriented Software Development, http://aosd net

[7] Pérez, J., Ali, N., Carsí, J.A., Ramos, I. "Designing Software Architectures with an Aspect-Oriented Architecture Description Language", 9th International Symposium on Component-Based Software Engineering (CBSE 2006), Mälardalen University, Västerås near Stockholm, Sweden, June 29th -1st July 2006 (accepted to appear)

[8] Perez, J., Ali, N., Costa, C., Carsí, J.A., Ramos, I. "Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology",3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005 , 2005

[9] Magee, J., Dulay, N., Eisenbach, S., Krammer, J. "Specifying Distributed Software Architectures". 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, 1995, pp 137-153.

[10] Milner, R., Parrow, J., Walker, D. "A calculus of mobile processes"**,** Parts 1-2. Information and Computation, 100(1), 1992, pp. 1-77.

[11] Magee, J., Tseng, A, Kramer, J. "Composing Distributed Objects in CORBA", Third International Symposium on Autonomous Decentralized Systems, Berlin Germany, 1997, pp 257-263.

[12] Virginia C. de Paula, G.R., Justo, Cunha, Ribeiro, P. R. F. "Specifying Dynamic Distributed Software Architectures", XII Brazilian Symposium on Software Engineering, BCS Press, October, 1998.

[13] Ciancarini, P., Mascolo, C. "Software Architecture and Mobility", 3rd Int. Software Architecture Workshop (ISAW-3), November, 1998.

[14] Mascolo, C. "MobiS: A Specification Language for Mobile Systems". 3rd International Conference on Coordination Models and Languages, 1999.

[15]Medvidovic, N., Rakic, M. "Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility". Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.

[16] Lopes, A. Fiadeiro, J.L., Wermelinger, M. "Architectural Primitives for Distribution and Mobility", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, pp. 41-50.

[17] Ali, N., Ramos, I., Carsi, J.A. "A Conceptual Model for Distributed Aspect Oriented Software Architectures", International Conference on Information Technology (ITCC 2005), IEEE Computer Society, ISBN 0-7695-2315-3, April 2005, pp 422-427.

[18] Cardelli, L. "Abstractions for Mobile Computation." In Vitek, J. and (Eds.), C. J., editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects, volume 1603 of LNCS, Springer Verlag, pp. 51-94.

Configurable Satisfiability Propagation for Goal Models Using Dynamic Compilation Techniques[1]

Elena Navarro[*] Patricio Letelier[**], David Reolid[*] and Isidro Ramos[**]

[*]Computing Systems Department, UCLM, EPSA, Campus Universitario s/n, Albacete, 02071, Spain, [enavarro|dreolid] @info-ab.uclm.es
[**]Department of Information Systems and Computation, UPV, Camino de Vera s/n, 46022, Valencia, Spain, [letelier|iramos]@dsic.upv.es

1 Introduction

It is frequently the case that at early stages of the requirements engineering process, critical decisions about what the system should provide are taken. Stakeholders and developers must evaluate alternatives and conflicts among the system requirements. In addition, a great deal of work must be done through focused brainstorming, validation, negotiation, and decision-making associated to vague or not completely defined requirements. In this context, Goal-Oriented modeling techniques emerge as a suitable way of defining and analyzing requirements, but also as an effective way to provide the necessary traceability towards other derived software artifacts.

This work aims at introducing a framework for exploiting Goal Models that allows the analyst to customize the analysis mechanisms according to the project needs. Our approach is based on the propagation algorithm proposed by (Giorgini et al. 2003), which establishes the essential computation of propagation. Using metamodeling techniques we provide the analyst with extensibility and customization mechanisms to modify the computation according to particular Goal Model elements, application domain, business rules, etc. These facilities are supplied by MORPHEUS, a tool we have developed for supporting our proposal.

[1] This work has been funded by the Spanish CICYT project DYNAMICA TIC2003-07776-C02-02.

Navarro, E., Letelier, P., Reolid, D., Ramos, I. Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques. Proc. of the 15th International Conference on Information System Development. 2006.

The remainder of this work is structured as follows. In section 2 a brief introduction to Goal Models and their analysis capabilities for requirements is presented. Section 3 describes our proposal presenting the integration of metamodel elements in rule description that are used in a propagation algorithm. Section 4 describes MORPHEUS, a tool developed to give support to our framework and especially an add-in incorporated to exploit the Goal Model by means of satisfaction/denegation propagation. Eventually, in section 5 conclusions and future work are presented.

2 Background

A Goal Model is built as a directed graph by means of a refinement from the systems *goals* (or concerns). This refinement lasts until goals have enough granularity and detail so as to describe an *operationalization*, i.e., a solution that provides the target system to meet users' needs and expectations. This refinement process is performed by using *AND/OR* refinement relationships. An AND (OR) relationship between a goal $Goal_X$ and a set of sub-goals $G_1 \ldots G_N$ is established if the whole set of (at least one) sub-goals has to be satisfied in order to satisfy $Goal_X$. In addition, operationalizations are associated to the requirements (leaf goals) by means of *contribution* relationships that denote how they collaborate to achieve a goal.

Once a Goal Model is defined mechanisms can be used to analyse its satisfiability. The satisfaction (denegation) of a goal means that it will (will not) be provided by the system-to-be, i.e., user's needs and expectations will (will not) be met. The propagation to carry out this reasoning about goal satisfaction is addressed by means of two approaches:

- *Qualitative approach*. The idea is to establish positive or negative influence (for instance, by means of ++, +, #, -, -- symbols) of contributions from operationalizations to goals in the Goal Model. These operationalizations can be designs, agents, events on the market, etc., depending on the specific Goal Model that is being used. In this sense, the degree of satisfaction does not have a precise interpretation, i.e., it is not based on domain or system properties but on the analyst criteria. (Chung et al. 2000) and (Giorgini et al. 2003) are examples of this approach.
- *Quantitative approach*. In this case, weights are set to contribution relationships describing the satisfaction degrees that goals have among them. The propagation is performed in a similar way to the previous case, but now a specific value of satisfiability is achieved. Those weights can be assigned according to quite different criteria:

  1. *Subjective assignment* where only the analyst criteria is used to decide, as for instance (Giorgini et al. 2003)'s proposal.
  2. *Objective assignment*, which is based on domain properties. Some proposals are that presented by (Letier and Lamsweerde 2004) for reasoning about par-

tial satisfiability of requirement; or (Hansen et al. 1998) to analyze safety constraints of system-to-be with fault-trees.

Figure 1 shows a summary example of a Goal Model which has been defined within the European project Environmental Friendly and cost-effective Technology for Coating Removal (EFTCOR, 2003). Its aim is to design a family of robots capable of performing maintenance operations for ship hulls. On the left of the image, it can be observed how goals are refined from a high level goal (suitability) towards requirements (operationalizable goals) and operationalizations. For instance, it can be observed that to achieve a suitable system-to-be both "*ApproachRobot*" and "*CatchObject*" have to be satisfied; or how "*MoveUsingMUC*" positively contributes to "*MoveStepArms*". In addition, on the right side of the image the result of the propagation for that selected operationalizations is shown. The explanation of the used rules is presented in the next section.

Currently, there is no standard notation for goal-oriented specification but several proposals have appeared that address different activities and perspectives in the Requirements Engineering Process[2]. In this sense, Goal Models are mainly exploited for evaluating alternative designs such as those described by (Chung et al. 2000) (Letier and Lamsweerde 2004), business goals (Giorgini et al. 2003), etc. However, there is no consensus on which the most appropriate mechanism or proposal should be. For this reason, it is the analyst who finally has to make the final decision about which should be used for a specific project.

Similarly, the propagation of satisfaction/denegation through the Goal Model depends on the application domain and the expressiveness of the Goal Model (provided by its elements such as kinds of refinements, artefacts and associated attributes). Furthermore, the propagation rules could also be specific for the project or could even be modified in the same project in order to reflect some additional consideration during the analysis. This required flexibility is missing in the current

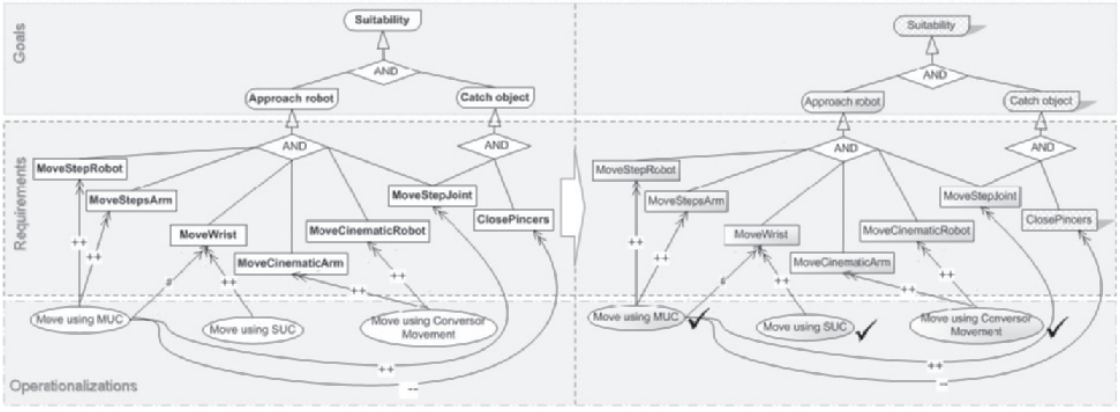


Fig. 1. Propagation of Satisfiability/Denegability(Selected Operationalization ✓, Satisfied Goal or Requirement Satisfied, Denied Goal or Requirement Denied)

[2] (Kavakli and Loucopoulos 2004) offer a detailed comparative framework about these proposals and their role in Requirements Engineering

proposals and constitutes an important obstacle while applying a specific Goal Model.

3 Our proposal

Our proposal took shape in the context of the ATRIUM methodology, presented by (Navarro et al. 2003), that guides the analyst through an iterative process, from a set of user/system needs to the instantiation of a Software Architecture. This proposal employs a Goal Model which is based on the (Dardenne et al. 1993) and (Chung et al. 2000) proposals. This Goal Model was extended by (Navarro et al. 2004) integrating the Aspect-Oriented approach, in order to achieve both the efficient management of crosscutting and the correct organization of the SRS (Software Requirement Specification). A set of techniques have been also incorporated to identify and specify *variability*, from the requirements stage, so that product lines and dynamic architectures can be dealt with. In this context, mechanisms for exploiting the ATRIUM Goal Model had to be defined and developed that do not only deal with specific expressiveness of the model but customize these techniques according to other specific needs.

In order to facilitate this customization a proposal for requirements metamodeling was developed and introduced in (Navarro et al. 2006). In this work, the most widely known notation for requirements specification (use cases, goal models, etc) were studied so as to identify the essential terms and concepts of each of them. Taking into account this study a metamodel for the essential concepts was defined that allows one to deal with generic expressiveness. The core concepts and their relationships are shown in Fig. 2. The first topic to consider was to define the metamodel is the description of artefacts. It allows one to describe any element to be included in the SRS. In this sense, any needed artefact can be described by inheriting from *Artefact* metaclass.

In addition to the artefact concept, it is also necessary to establish the artefact relationship with other artefacts of the SRS. Therefore, two kinds of relationships were identified. An artefact can be refined through other artefacts, forming a hierarchical structure. The basic kind of refinement included is *Refinement* that allows the analyst to define hierarchies of alternative specializations from the same parent and to relate one child to more than one parent by multiple inheritance. In addition

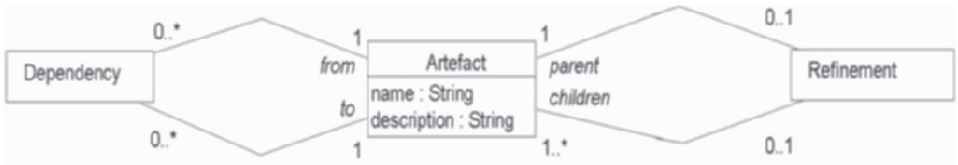


Fig. 2. Core Metamodel for Requirements Engineering

the *dependency* relationship has also been included in the metamodel. Perhaps this is the most conflictive relationship for consensus. For this reason, it is represented in its most generic form, i.e., by means of *Dependency* metaclass which is applicable to artefacts in the core. This metamodel has to be tailored according to the specific needs of expressiveness. With this purpose in mind, several steps were suggested to adapt and/or extend the metamodel. They need not be applied sequentially but in accordance with the analyst's preferences to describe new kinds of artefacts, relationships, attributes, constraints, etc, extending that described in the core.

In order to allow the analyst to customize the rules to be used during the propagation process an extension to the algorithm proposed by (Giorgini et al. 2003) has been developed. In this sense, the customization allows the analyst to include any kind of relationship and artefact along with their attributes to describe the propagation rules.

	$(G_2, G_3) \xrightarrow{and} G_1$	$G_2 \xrightarrow{+S} G_1$	$G_2 \xrightarrow{S} G_1$	$G_2 \xrightarrow{++S} G_1$	$G_2 \xrightarrow{S} G_1$
sat(G_1)	$\min \begin{Bmatrix} sat(G_2), \\ sat(G_3) \end{Bmatrix}$	$\min \begin{Bmatrix} sat(G_2), \\ P \end{Bmatrix}$	N	sat(G_2)	N
den(G_1)	$\min \begin{Bmatrix} den(G_2), \\ den(G_3) \end{Bmatrix}$	N	$\min \begin{Bmatrix} sat(G_2), \\ P \end{Bmatrix}$	N	sat(G_2)

Fig. 3. Qualitative Propagation rules described by (Giorgini et al. 2003), where $\xrightarrow{++S}$, $\xrightarrow{S}$, etc. are describing contribution relationships

(Giorgini et al. 2003) have described a set of rules to specify how the propagation has to be compute that Fig. 3 depicts. Sat(G_i) and Den(G_i) specify the satisfiability and deniability, respectively, of the goal G_i, whose value is taken from the ordered set {"++","--","+", "-", "#"}. For example, let G_2 and G_3 subgoals of G_1 refined by using an AND relationship, the satisfiability of G_1 is set to the minimum of satisfiability of its subgoals. This means that G_1 is undefined, partially satisfied or totally satisfied depending on which minimum value between G_2 and G_3 is. As can be observed in Fig. 1 "*ApproachRobot*" is fully satisfied because each one of its subgoals is satisfied. Regarding contribution relationships, Fig. 3 depicts that an asymmetric propagation is performed. For instance, $G_2 \xrightarrow{+S} G_1$ means that if G_2 is satisfied, then there is some evidence that G_1 is satisfied, but if G_2 is denied, then nothing is said about the satisfaction of G_1. On the contrary, if the relationship is $G_2 \xrightarrow{S} G_1$ then there is some evidence that G_1 is denied whether G_2 is satisfied, but if G_2 is satisfied, then nothing is said about the satisfaction of G_1. But Giorgini et al.'s have also described the rules for symmetric propagation. In that case, these rules consider the propagation of both satisfiability and deniability. An example could be the relationship $G_2 \xrightarrow{+} G_1$ that means that if G2 is satisfied (denied), then there is some evidence that G1 is satisfied (denied).

Table 1. Propagation algorithm based on (Giorgini et al. 2003)'s proposal

```
label array Label_Graph(graph (G, R); label array Initial)
   Current=Initial;
   do
      Old=Current;
      for each Gi ∈ G do Current[i] = Update_Label(i, (G, R) Old);
   until (Current==Old);
   return Current;

label Update_Label(int i; graph (G, R); label array Old)
   for each Rj ∈ R s.t. target(Rj) == Gi do
      satij = Apply Rules Sat(Gi;Rj ;Old);                    //dropped
      denij = Apply Rules Den(Gi;Rj ;Old);                    //dropped
   return (maxj(maxj(satj);Old[i].sat), max(maxj(denij);Old[i].den))
      for each Ak∈ A AND Ak∈ set_valuable_attributes(Gi)       //added
         if applicable (Ak, Gi, Rj, Old)                      //added
               valuate (Ak, Gi, Rj, Old)                      //added
    return set_valuable_attributes(Gi);                       //added

boolean applicable(Attribute Ak;Artifact Gi;Relation Rj;      //added
labelarray Old)
   // depending on the specific rule it will check if         //added
   // the valuation can be performed                          //added

valuate(Attribute Ak; Artifact Gi; Relation Rj;               //added
        label array Old)                                      //added
   // depending on the specific rule it will perform          //added
   // the valuation of the attribute Ak                       //added
```

Table 1 shows how Giorgini et al.'s algorithm has been modified for customization purposes, highlighting the added pseudo-code with shadowed text and the dropped pseudo-code with a grey text. Two main functions were initially described by Giorgini et al.'s: "*Label_Graph*" which iterates through the goal model until there is no changes in the satisfibiality and deniability; and "*Update_Label*" which apply the appropriate rule, from those described in Fig. 3 and depending on the kind of relationship, to compute the satisfibiality and deniability of each node of the graph. As can be noticed, the initial proposal only describes the valuation for two attributes (sat and den) with a fixed set of rules (Fig. 3). However, with our proposal this set and attributes can be customized according to the specific needs of the project, as will be described below.

It is shown in Table 1 that whenever a rule has to be applied two steps must be performed. First, *applicable* describes which state or situation has to be satisfied in order to apply a specific rule. Second, *valuate* specifies the propagation computation when a rule is applied. In view of this, the grammar for defining both quantitative and qualitative rules is introduced by using the Backus-Naur Form (BNF). Table 2 and Table 3 show how applicable and valuate can be described, respectively. By using this grammar the rules defined by (Giorgini et al. 2003) can be easily described as shown in formulae (1) and (3).

Table 2. BNF for describing condition grammar

```
<condition> ::  '('<condition>')'<relational op>'('<condition>')'
                  | not '('<condition> ')' | <node> <logic op> <node>
<node>   ::  <identifier>|<expression>| <function>'('<identifier> ')'
<identifier>   ::  <id kind artefact>'.'<id attribute>
         | <id  kind relation>'.'<id attribute>
<function>  ::  max | min | count | avg
<relational op>::  '&&' | '||' | '%%'
<logic op>::  '>' | '> ' | ' ' | '! ' | '< ' | '<' | '>'
<expression>   ::  ''<string>'' | <number>
```

Table 3. BNF for describing valuation grammar

```
<valuation>:: id kind artefact>'.'<id kind enum attr>' ' <val enum>
      | <id kind artefact>'.'<id  kind numb attr> ' ' <val  numb>
<val enum>  ::  <id kind artefact>'.'<id  kind enum attribute>
   |<expression enum>
   |<funct enum>'('<id kind artefact>'.'<id  kind enum attribute> ')'
<funct enum>   ::  max | min
<val  numb>:: '('<val numb>')'<op numb>'('<val numb>')'| <ident num>
               | <funct num> '(' <ident num> ')'| <expression num>
<funct num> ::   max | min | count | avg | sum | prod
<ident num> ::  <id kind numbered artefact>'.'<id attribute>
               | <id  kind relation>'.'<id attribute>
<op numb>   ::  '+' | ' ' | '*' | '/'
```

For instance, considering how the *CONTRIBUTION* relationship ($G_2 \xrightarrow{label} G_1$) is evaluated by Giorgini et al. we can appreciate that both the state of this relationship and the Goal source (G_S) are used to determine if the rule can be applied or not. In this sense, the condition could be described as: G_S*(satisfied) && label=--S*, i.e., G_S has an attribute that describes if G_S is satisfied. It is similarly applied to *label*, i.e., *CONTRIBUTION* relationship needs an attribute for specifying *--S* as its current state. In these terms, the best alternative is to represent these attributes following a syntax as described in Table 2 for *<identifier>*, i.e., by prefixing the attribute name with the name of the artefact or the relationship (see (1)).

In addition, when refinement relationships are considered, for instance an AND relationship ($(G_1 \mathrm{K} G_n) \xrightarrow{and} G_D$), some functions may be needed to determine the condition being applied to the set of artefacts G_1 to G_n. For this reason, an easy alternative is to use group functions as *<function>* describes in Table 2.

Furthermore, it is worthy of note that conditions can be combined to express other more complex ones. This will be described by using relational operations. This means that it would be possible to describe conditions such as:

$$(\text{Goal}_S.\text{satisfied} = \text{'Full'}) \text{ and } (\text{CONTRIBUTION.label}) = \text{'-S'} \quad (1)$$

$$(\text{Goal}_i.\text{type} = \text{'Peformance'}) \text{ and } (\text{Goal}_i.\text{priority} = \text{'High'}) \text{ and } \min(\text{Goal}_i.\text{satisfied} = \text{'Full'}) = 2 \quad (2)$$

In a similar maner, the syntax for the valuation is described by using the BNF (Table 3). As was stated in section 2, both quantitative and qualitative valuation should be described. For this reason, we have to distinguish between the two in order to make the peculiarities inherent in both kinds of valuations available. While describing qualitative valuations only enumerated attributes are made available to the analyst. This restriction is straightforward so that possible valuations are always constrained to a set of values. For instance, when considering the satisfiability, as described by Giorgini et al., the set *{Full, Partial, None}* is used. It also facilitates the valuation of this kind of attributes by describing functions for enumerations (<*function_enum*>). This requires that the set be defined as an ordered set in order to be able to properly apply these functions (min and max). This means that the valuation for an AND relationship (Fig. 3) could be easily described as appears in (3).

$$\text{Goal}_D.\text{satisfied} = \min(\text{Goal}_i.\text{satisfied}) \quad (3)$$

Related to the artefacts involved in a refinement relationship, aggregated functions (<*function_num*>) can be used for its treatment as described in:

$$\text{Goal}_D.\text{satisfied} = \text{sum}(\text{Goal}_i.\text{satisfied}) - \text{prod}(\text{Goal}_i.\text{satisfied}) \quad (4)$$

Some specific rules had to be introduced since ATRIUM Goal Model was defined to facilitate the analysis of variability. For instance, it has to be considered that whenever a *variation point* is described, its *multiplicity* must be defined, i.e., how many variants must exist at the same time in a product or architecture when the variability is being removed. Table 4 shows how rule OR relationship has been modified for dealing with variability.

Table 4. Defining rules for variability analysis

	Relationship	**Condition**	**Valuation**
Giorgini et al.	Satisfiability ($G_{1...}\ G_n \xrightarrow{OR} G_D$)		$\text{Goal}_D.\text{Sat} = \max(\text{Goal}_i.\text{Sat})$
Variability	Satisfiability ($G_{1...}\ G_n \xrightarrow{OR} G_D$)	(count(Goal_i.Sat=="+S") + count(Goal_i.Sat=="++S")) >= OR.multiplicity.min) && (count(Goal_i.Sat=="+S") + count(Goal_i.Sat=="++S")) <= OR.multiplicity.max)	$\text{Goal}_D.\text{Sat} = \max(\text{Goal}_i.\text{Sat})$

4 Morpheus: Using dynamic compilation techniques

With the aim of supporting this proposal a tool called MORPHEUS has been used. Due to the fact that it has to manage each model defined by ATRIUM (Requirements Model, Scenarios Model and Software Architecture Model) three different environments are provided by MORPHEUS (Fig. 4). Related to the Requirements Model, MORPHEUS is able not only to define both new kinds of artefacts and relationships but also to instantiate and exploit them, in such a way Goal Models with different expressiveness can be defined. For this reason, the Requirements Model environment has been split into two different working contexts. The first one allows analysts to establish the requirements metamodel to be used by means of the Metamodel Editor; and the second provides analysts with facilities for modelling according to the defined metamodel by using the Model Editor.

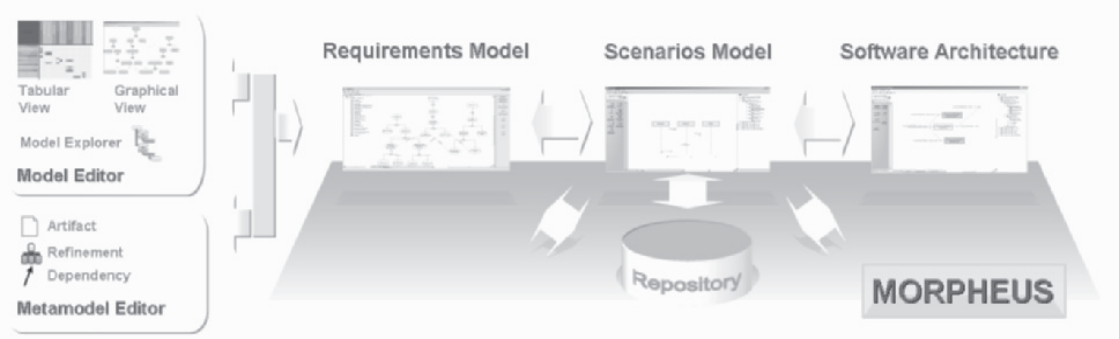


Fig. 4. Describing MORPHEUS capabilities MORPHEUS capabilities

In addition, MORPHEUS has been developed with capabilities to extend its functionality with analysis techniques. The main reason is that as new metamodels are defined, their related techniques can also be included and exploited. An example of this capability has been the development and integration of an add-in for Goal Model analysis based on satisfability propagation. Fig. 5 shows how this add-in has been designed. It has been split into three main components: a Rules Editor, a Code Compiler and a Propagation Processor.

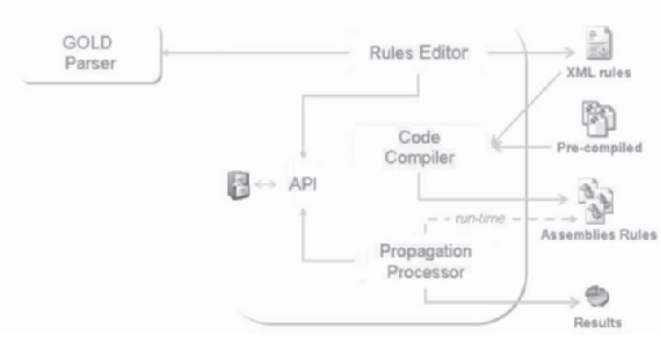


Fig. 5. A sketched view of the propagation add-in

Figure 6 shows what MORPHEUS looks like whenever the Rule Editor is loaded. For its development several alternatives were evaluated. However, the usability of the proposal was one of the main characteristics to be achieved. For this reason, a user interface (Rule Editor in Fig. 5) that allows the analyst to introduce the rules in a simple and comprehensible manner was developed. The Rule Editor is split into three main parts: a browser, a rules descriptor and an editor. The browser allows one to navigate through the relationships and the artefacts connected by them. The rules descriptor displays the applicable rules, for a selected relationship and source and destination artefacts. It can be appreciated that the *when* text box describes the condition and next to it appears the valuation. Below the rules descriptor, a visual control permits to edit the condition and the valuation. This provides the analyst with several buttons and capabilities that prevent him from knowing any detail about how his/her metamodel is described in the repository or how rules are internally implemented.

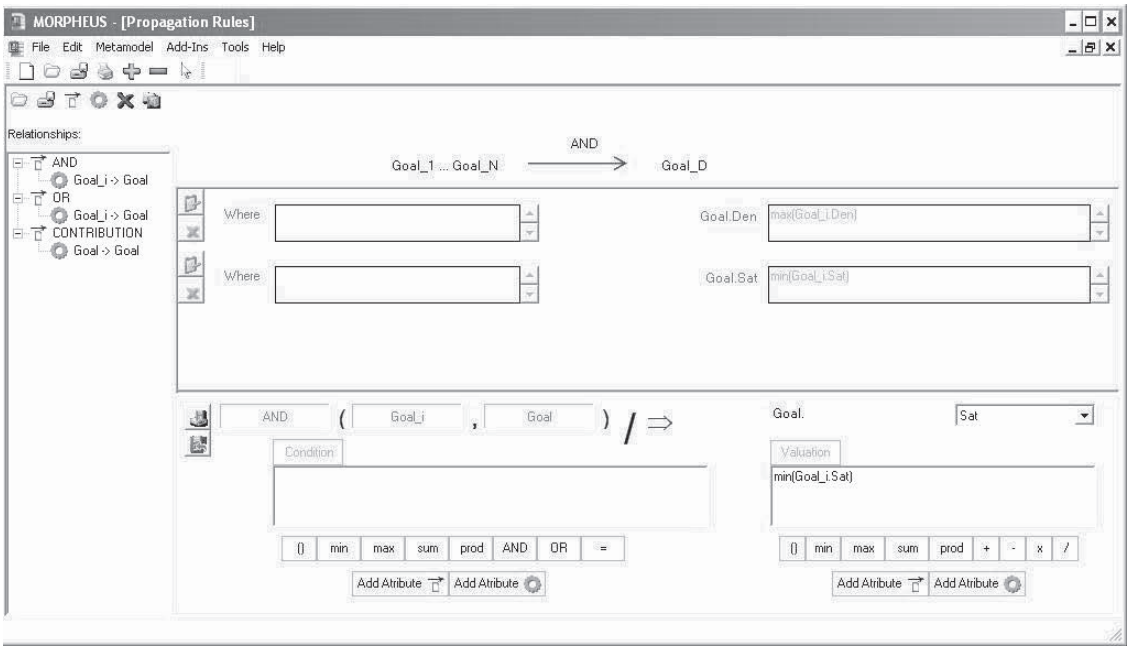


Fig. 6. MORPHEUS while loading the Rule Editor

In addition, a syntactic checking is performed when a rule is being defined thanks to the capabilities provided by (GOLD 2005). This is a free parsing system that can be used to develop one's own programming languages, scripting languages and interpreters by previously writing your grammar using BNF. The BNF, which was described for the condition (Table 2) and the valuation (Table 3), was introduced in GOLD. Then, the GOLD Parser Builder was used to analyze this grammar and create the Compiled Grammar Table file (CGT) used by a compilation engine. It uses this CGT file to generate a C# skeleton program with a custom parser class that acts as a template for parsing any source satisfying the BNF grammar. By means of this template, the specific compilation to the objective code can be described. In this case, a translation to C# code was performed to make available artifacts and relationships from the repository, and that computa-

tion which was needed for both condition and valuation. So, for each rule its valuation and condition description along with their respective compilation to C# is stored in a XML rules file to be lately used for the Code Compiler. The performance could have been seriously compromised due to the necessity of representing these rules as code for its use in run-time. For this reason, the approach of dynamic compiling, while it is more complex, provides us with a proper solution. Microsoft .NET Code Document Object Model technology, as described by (Harrison 2003), has been used for this purpose for the implementation of the Code Compiler.

By using Code Compiler (Fig. 5) a set of assemblies, containing both the rule code and other functionality, is generated at run time. For each rule, which is saved in a XML rule file, a C# class is generated which inherits from IRule. It is an abstract class with two abstracts methods to override for each inherited class: applicable function, which checks if the rule can be applied; and valuate function, which performs the propagation computation. This class also has a set of functions to perform the minimum, maximum, etc. So, while generating code, each rule C# class is going to override the abstract methods with that code stored in the XML rule file. Other classes are also used for the management of the generated classes which are previously pre-compiled to speed up this process.

Afterwards, these assemblies are accessed by the Propagation Processor to perform the propagation on a specific goal model and generate the results. In terms of integration, Propagation Processor makes use of the MORPHEUS API to access the model and will pass through the relations and artefacts retrieved from the repository.

5 Conclusions and further work

Goal Models are a very promising technique to improve requirements elicitation. Thanks to their special capabilities to analyze goals/requirements, they can be used at early stages of requirements engineering process, when alternatives are explored, conflicts are identified and, in general, the project is in the phase of requirement negotiation. However, Goal Model techniques must face a common obstacle in requirements engineering: the diversity of proposals with an evident lack of integration; and the specific needs of the project (or domain), which usually requires a customization of the requirement method and its notation.

We have presented a framework to cope with the integration and customization problem in requirements engineering techniques. In this work, we have used our framework to provide customizable support in Goal Model propagation analysis. We have illustrated how the propagation rules can be defined according to a specific metamodel. In this way, we can not only establish propagation rules for analysis but also redefine the propagation rules whenever it is necessary. This functionality was provided by means of dynamic compilation techniques using CodeDom. MORPHEUS, a tool we have developed to give support to our approach,

was extended with an add-in which allows the analyst to define propagation rules associated to the established metamodel and perform their computation.

Following this line of research, our ongoing work covers several issues. First, we intend to go more deeply into the analysis and exploitation of Goal Models using the specification of the goal/requirements itself apart from its attributes and relationships with other goals/requirements. This entails working with formal specifications of requirements (at least at some level of formalization) to perform a deeper verification and provide some automated support for the Goal Model specification. Other interest is to improve the mechanisms for analyzing the propagation, including graphical visualization (apart from tabular representation) and predefined report analysis. Finally, we are working on the establishment of precedence mechanisms for solving conflicts when more than one rule is applicable.

6 References

Chung L, Nixon B A, Yu E and Mylopoulos J (2000) Non-Functional Requirements in Software Engineering, Kluwer Academic Publishing.

Dardenne A, Lamsweerde A van, and Fickas S (1993) Goal-directed Requirements Acquisition. Science of Computer Programming, 20, pp 3-50.

Giorgini P, Nicchiarelli E, Mylopoulous J, and Sebastiani R (2003) Formal reasoning techniques for goal models. Journal of Data Semantics, 1: 1-20.

GOLD Parsing System, http://www.devincook.com/GOLParser/ index.htm, 2005.

Hansen K M, Ravn A P, Stavridou V (1998) From Safety analysis to software requirements. IEEE Tran. on Software Engineering, 24(7):573-584.

Harrison N (2003) Using the CodeDOM. O'Reilly Network, http://www.ondotnet.com/pub/a/dotnet/2003/02/03/codedom.html.

Kavakli E and Loucopoulos P (2004) Goal Driven Requirements Engineering: Analysis and Critique of Current Methods. In: Krogstie J, Halpin T and Siau K (eds) Information Modeling Methods and Methodologies, 102-124.

Lamsweerde A van (2000) Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice. In: Proc 12th IEEE International Requirements Engineering Conference, IEEE Computer Society, Los Alamitos, pp 4-7.

Letier E and Lamsweerde A van (2004) Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering. In: Taylor R N, Dwyer M B (eds) Proc of 12th ACM International Symposium on the Foundations of Software Engineering, ACM Press, New York, pp 53-62.

Navarro E, Letelier P, Mocholí, J.A. Ramos I (2006) A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems, 46(5): 67-77, Special Issue on Systems Analysis and Design.

Navarro E, Letelier P and Ramos I (2004) Goals and Quality Characteristics: Separating Concerns, Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, collocated to OOPSLA.

Navarro E, Ramos I and Pérez J (2003) Software Requirements for Architectured Systems. In Proc of 11th IEEE International Requirements Engineering Conference, IEEE Computer Society, Los Alamitos, pp 365-366 (position paper).

Requirements and Scenarios: running Aspect-Oriented Software Architectures*

Elena Navarro
Department of Computing Systems, UCLM
Avda. España S/N
Albacete, Spain
enavarro@info-ab.uclm.es

Patricio Letelier
Dept. of Information Systems and Computation, UPV, Camino de Vera s/n
Valencia, Spain
letelier@dsic.upv.es

Isidro Ramos
Dept. of Information Systems and Computation, UPV, Camino de Vera s/n
Valencia, Spain
iramos@dsic.upv.es

Abstract

The intertwining of Requirements and Software Architectures must be based on solid traceability mechanisms in order to effectively maintain the consistency between the two specifications. A new issue has come to the fore in this area: Aspect-Oriented Software Development (AOSD). Achieving the separation of concerns from the very beginning of software development along with their proper traceability throughout the lifecycle is mandatory in terms of quality and maintainability. ATRIUM is a methodology that is defined to guide the development of architectures from requirements and using AOSD techniques as its cornerstone. In this work, we present how ATRIUM uses a scenario technique that provides traceability support. ATRIUM pays special attention to non-functional requirements, their crosscutting relationships, and how they can derive aspects in the software architecture.

1. Introduction

Several definitions for Software Architecture (SA) have emerged over time. Perry and Wolf [16] state that "architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design". This statement shows that the SA is a bridge between requirements and implementation. SA permits reasoning about whether a system meets its requirements and, therefore, assess the quality shown by the system. The key factor in achieving this intertwining is the establishment of appropriate traceability mechanisms.

In addition, a new issue has emerged: Aspect-Oriented Software Development (AOSD) [6]. Its goal it to provide a proper separation of concerns both functional and non-functional (performance, safety, security, etc). A great deal of work related to this paradigm has been performed from requirements [2] to architecture [5] and implementation [7]. In this context, how to identify, specify, and trace each concern across the software life-cycle satisfying the closure property [6] is an open issue that has not yet been solved [3].

In this context, we have proposed ATRIUM (Architecture generaTed from RequIrements applying a Unified Methodology) [11] a methodology that pays special attention to the traceability between requirements and SA using the AOSD approach (Figure 1). The aim of this work is to describe how ATRIUM achieves the integration of non-functional concerns for their later transformation into a proto-architecture by means of enrichment of functional scenarios and use of aspectual techniques.

As a target architecture description language we use PRISMA [13] which combines Component-Based Software Development (CBSD) and AOSD to model

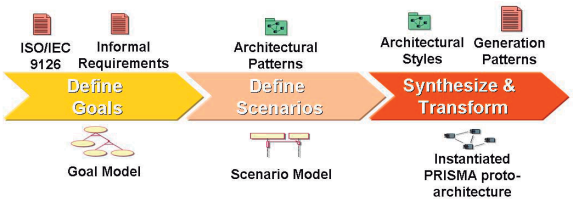


Figure 1. A sketched view of ATRIUM

* This work has been funded by the Spanish CICYT project DYNAMICA (TIC2003-07776-C02-02)

Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

© IEEE COMPUTER SOCIETY

Navarro, E., Letelier, P., Ramos, I. Requirements and Scenarios: running Aspect Oriented Software Architectures. Proc. of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). 2007.

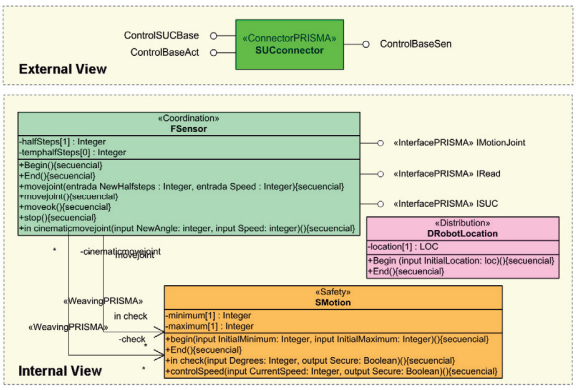


Figure 2. Internal and external view of a PRISMA connector

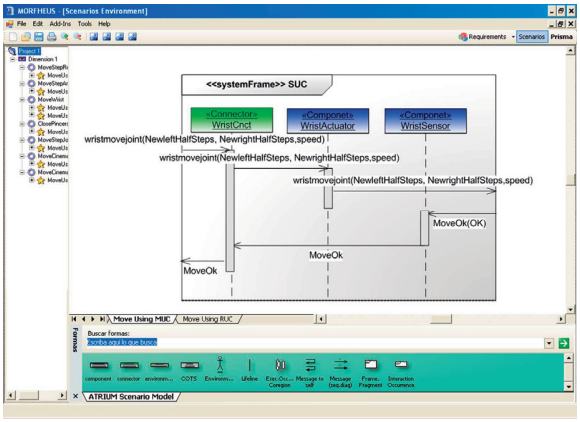


Figure 3. Describing a SUC system

software architectures. In this sense, it specifies the different concerns (safety, distribution, coordination, etc.) that crosscut a software system as aspects. These aspects are used to define PRISMA architectural elements, i.e., components, connectors and systems (Figure 2).

This work is structured as follow. Section 2 describes the defined scenario notation along with a process for its utilization. Section 3 presents MORPHEUS which is a tool that provides support to ATRIUM. Finally, related works and conclusions and futures works conclude this work.

2. ATRIUM Scenario Model

As observed in Figure 1, a Goal Model is used as an input for the construction of the Scenario Model. The elaboration of both models is performed by two intertwined processes in order to overcome some of their deficiencies and limitations when they are used in isolation. This provides the necessary maintainability and traceability

In the ATRIUM Scenario Model, a *scenario* is used to describe the system behaviour associated to a specific requirement and taking into account certain operationalization decisions. A scenario is described by the sequence of interaction messages in accordance with a specific choreography. Unlike classic scenario proposals, our scenarios specify architectural elements interaction instead of objects. They also specify the environmental elements which play a role in these scenarios.

We use Sequence Diagrams (SD) of UML 2.0 [19] as the graphical language for scenarios. SDs help to identify the coordination structure of the involved architectural and environment elements throughout their construction. These time-based interactions help us to define the choreography for the involved elements. UML 2.0 provides several notation elements that are used to describe the ATRIUM Scenario Model. These elements along with their specific use in ATRIUM are described as follow:

- *Lifelines* represent both architectural elements and environment elements that participate in the scenario being modelled. As can be observed in Figure 3, each kind of architectural or environment element is represented by means of a different UML stereotype («component», «connector», «human» and «environment»).
- *Messages* describe services that the requested architectural or environmental element provides. These services will belong to interfaces of the architectural elements.
- *Guards* describe when a condition must be met for a message to be sent to the object.
- *Frames* are a new addition to the notation in UML 2.0 diagrams to describe a graphical boundary for a diagram. Specifically, they are used to provide notation for *alternatives, optionalities, parallel composition,* and *references.* These notation elements are used to simplify the specification of scenarios.

One of the characteristic that PRISMA has is its ability to decompose the description of a Software Architecture by means of *Systems*. A PRISMA system is a complex component that is internally described by means of components, connectors and systems. Thus, systems are specified by means of frames. This frame is labelled using «systemFrame» as UML stereotype. An example of that facility is shown in Figure 3, where a system called *SUC* is identified by that initial description of shallow components and connectors.

Taking into account that PRISMA promotes the separation of concerns by using aspects to describe the internal view of components and connectors (see Figure 2), it is important to provide support to the detection and specification of aspects from

IEEE COMPUTER SOCIETY

requirements and maintain the proper traceability. A PRISMA aspect is captured as a collection of services from a set of scenarios according to a specific concern. It makes no sense to introduce aspects as another kind of interacting element in a scenario because they are described in the internal view of the architectural elements. Thus, whenever the analyst considers that a service should be included in the specification of a specific aspect, this service will be prefixed with the name of the concern it refers to. Figure 4 shows an example of how the service *secure check()* has been prefixed with *C{Safety}* to indicate that it will be part of a safety concern. To facilitate the legibility of the scenario, a code colour that can be customized according to the user preferences has also been defined. In this case, the colour is orange to highlight that it is related to a safety concern.

2.1. A process for Scenario Model Description

ATRIUM proposes a set of activities to guide the analyst in making the architectural description. The initial activity is to select the *architectural style* that will be appropriate for the system-to-be. The architectural styles have a great impact on the final description of the architecture because they cross the whole architecture. Shaw & Clements [18] offered a taxonomy where styles such as Layered architecture, Blackboard, etc., have been described. However, domain-driven architectural styles have recently been suggested because they provide analysts with more information and guidance for the architectural description. An example of this approach is ACROSET [12] which describes a style for the development of the control units of teleoperated service robots.

Table 1. ISO 9216 for selecting Architectural Styles

Characteristic	Sub- Characteristic		Layer	Pipe& Filter	…	ACROSET
functionality	suitability	teleoperated				++
		e-business				
		…				
	Interoperability		++	+		++
	Security		++			
	…					
…						
Safety						++
…						

Before describing any possible scenario, the analyst must make a decision about what architectural style should be used for the system-to-be. Since the Goal Model uses the quality model ISO 9126 as the initial framework of concerns, along with their assigned priorities, it can be employed for the selection of the architectural style. Therefore, we can evaluate the

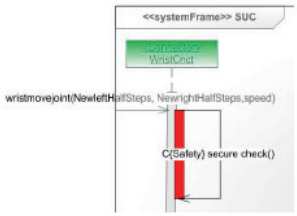


Figure 4. Describing a Safety Concern

impact of the architectural styles on the ISO characteristics/sub-characteristics that are included in the Goal Model. These architectural styles have been classified (Table 1) according to their positive and negative contributions to each quality characteristic. The sub-characteristic suitability has been decomposed into several sub-kinds for different domains.

Once the architectural style has been selected, the operationalization of the requirements can be applied iteratively in two different phases. The first phase focuses its efforts on functional requirements. The second phase only considers the non functional ones because most non-functional requirements are analyzed as constraints that are applicable to functional ones. This proposal introduces the transformation recommended by Bosch [4], i.e., integrating quality requirements into functional solutions. This extends the architecture with functionality that is not related to the problem domain but that is used to fulfil the requirements. The process entails the following activities:

- *Select requirement to operationalize*. The analyst must select from the Goal Model a requirement which has not yet been operationalized.
- *Look for architectural patterns in the catalogue*. In our proposal, we have included both domain and technological patterns by describing them in a catalogue. These patterns have been indexed according to the following: the concerns of the ISO 9126 the positively or negatively contribute to; and, the architectural styles that they are more appropriate for. Thus, the analyst can select a pattern or set of patterns that is instantiated to operationalize the requirement.
- *Specify alternative operationalizations*. Several alternatives can operationalize the same requirement, just like several alternative programs can implement the same specification. For this reason, the analyst must specify each detected operationalization taking into account the requirement that must be fulfilled. During the operationalization specification, the analyst has to perform two different tasks: to make a textual description of how the related requirement can be operationalized; and, to identify the software, hardware and environment elements which

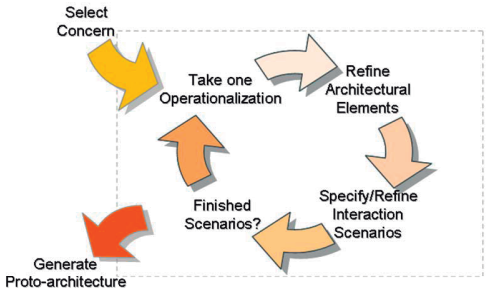


Figure 5. Process for Scenarios Description

collaborate in the operationalization. It is worthy of note that the splitting of the work between the system-to-be and its environment is delayed until the operationalization specification. This avoids taking premature decisions which would limit our ability to define different systems depending on time, costs, available resources, etc [8].

- *Specifiy contribution relationships* In order to obtain the best global solution, operationalizations have to be analysed to determine whether between them and other requirements conflicts exist. The analyst must then introduce a positive contribution relationship from each identified operationalization and its related requirement. In addition, if their inclusion in the system means that other requirements can be negatively/ positively affected then negative/possitive contributions would also be introduced.
- *Determine the end of the operationalizaiton process.* The analyst must decide if the operationalization process has finished, i.e., if every described requirement has its associated operationalizations. However, an entire specification is not necessary to continue the process since a partial description would be enough to obtain a draft generation of the architecture.

Once the description of the operationalizations has been accomplished, the analyst can proceed to analyze the architectural alternatives by using the algorithm described in [10]. It has been defined to determine a set of operationalizations with minimum conflicts.

Using the set of operationalizations, the analyst can start to describe the scenario/s associated to each operatinalization following the process described in Figure 5. Similarly to the previous process, two different phases can be distinguished. During the first phase, the analyst takes one functional requirement with the selected operationalization and describes its scenario/s. During the second phase, each scenario associated to a functional requirement which crosscut a non-functional one is enriched to fulfill the non-functional one.

3. Morpheus: providing tool support

A tool called MORPHEUS has been developed to support ATRIUM. Since it must manage each model used by ATRIUM three different environments are provided:

- *Requirements Environment* [10]. It provides user with a requirements tool to describe requirement metamodels that are customized according to project's semantic needs and to later exploit and analysis them.
- *Scenarios Environment*. It provides a tool called *Scenarios Editor* where the analyst can describe the scenarios according to the notation described in section 2 along with a *Model Explorer* that facilitates the navigation through the model (Figure 3). Finally, a *Synthesis processor* is in charge of generating the proto-architecture by implementing the specified QVT rules using XSL Transformations (XSLT). This technology has been selected because the Scenario Models are extracted from the repository as XML documents to generate a XML PRISMA specification.
- *Software Architecture Environment* [14] provides the analyst with a graphical environment for the PRISMA ADL [13] so that the proto-architecture obtained from the Scenario Model can be refined and lately used to generate code. This code is run in the PRISMANET Middleware [15].

4. Related works

Araujo et al [2] have introduced the crosscutting management during the definition of scenarios. However, their purpose was to describe scenarios that crosscut other scenarios as interaction pattern. They di not provide any notation for aspect description nor traceability through the lifecycle of the system-to-be.

GRL [9] has been defined for traceability from goals to Use Case Maps (UCM). However, these UCM are far from the architectural scenarios that are used in our work because their abstraction level is much higher. In addition, GRL make an early split of the work between the system-to-be and the environment because the responsibility for this work is assigned at the goal level.

Both GBRAM [1] and CREWS l'Escritorie [17] are also similar to our proposal because they use scenarios as operationalizations of the specified goals. However, as in the previous proposal, they are not architectural scenarios due to their higher abstraction level.

In addition, we would like to highlight that none of the above mentioned proposals, as far as we know, deal

IEEE COMPUTER SOCIETY

with an open issue: the separation of concerns and their traceability from the very beginning of the software lifecycle. This challenge is the main purpose of ATRIUM.

5.Conclusions and Future works

One of characteristics that can be highlighted in our proposal is traceability. The traceability support from system goals to architectural elements, passing through requirements, operationalizations, and scenarios has been one of our main goals. The Aspect-Oriented approach is integrated throughout the process traceability and supported by a tool called MORPHEUS

Another important characteristic of our approach is that the splitting of responsibilities to accomplish the requirements is delayed until the operationalization analysis. This means an improved flexibility for the entire development process.

With regard to the scenario descriptions, it should be emphasized that, in our proposal, they are specified in enough detail level to be used for the proto-architecture generation by synthesizing the behaviour from the set of scenarios. The scenarios include operationalization decisions which take into account functional and non-funtional requirements and their crosscuttings. On the other hand, the enrichment of scenarios when considering crosscutting among requirements provides a systematic way of dealing with early aspects and their traceability to software architecture.

As future works there are several topics to deal with. The first one is related to the application of our approach and tool to other domains to achieve a wider validation. The second one is related to analyzing conflicts. Incompatibilities between architectural styles may appear during their selection. For this reason, the analyst must be provided with techniques that facilitate the necessary trade-offs.

References

[1] A. I. Antón, "Goal-Based Requirements Analysis", Proc. 2nd ICRE, Colorado Springs, CO April 15 - 18, 1996.

[2] J. Araujo, J. Whittle, D. Kim, "Modeling and Composing Scenario-Based Requirements with Aspects". 12th IEEE RE, 6-10 September 2004, Kyoto, Japan.

[3] E. Baniassad, P. C. Clements, J. Araújo, A. Moreira, A. Rashid, B. Tekinerdogan, "Discovering Early Aspects", IEEE Software 23(1), January/February 2006. pp. 61-70.

[4] J. Bosch, Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach, Addison-Wesley, May 2000.

[5] C. Cuesta, M. P. Romay, P. de la Fuente, M. Barrio-Solórzano, "Architectural Aspects for Architectural Aspects", 2nd EWSA, Pisa, June 2005, LNCS vol 2005.

[6] T. Elrad M. Aksits, G. Kiczales, K. Lieberherr, H. Ossher, "Discussing aspects of AOP", Communications of the ACM, Vol. 44 , Issue 10, pp. 29 – 32, October, 2001.

[7] G. Kiczales , J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming". ECOOP, Finland. LNCS 1241. June 1997.

[8] S. Lauesen, "Task Descriptions as Functional Requirements". IEEE Software 20(2): 58-65, 2003.

[9] L. Liu and E. Yu, "Designing Information Systems in Social Context: A Goal and Scenario Modelling Approach", Information Systems, 29(2) (April 2004), pp. 187 – 203.

[10] E. Navarro, P. Letelier, D. Reolid, I. Ramos, "Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques", 15th ISD, Budapest, Hungary, 31 August, 2006 - 2 September, 2006.

[11] E. Navarro, I. Ramos, J. Pérez, "Software Requirements for Architectured Systems", 11th IEEE RE'03 (short paper), Monterey, USA, September 8-12, 2003.

[12] F. J. Ortiz, D. Alonso, B. Álvarez, J. A. Pastor, "A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle", Proc. 10th Ada-Europe Int. Conf. on Reliable Software Technologies, York, UK, June 20-24, 2005, pp. 13-24

[13] J. Pérez, N. Ali, J. Á. Carsí, I. Ramos, "Designing Software Architectures with an Aspect-Oriented Architecture Description Language", 9th CBSE 2006, Västerås, Sweden, June 29th -1st July 2006.

[14] J. Pérez, E. Navarro, P. Letelier and I. Ramos, A Modelling Proposal for Aspect-Oriented Software Architectures, 13th IEEE ECBS, Potsdam, Germany, March 27th-30th, 2006.

[15] J. Pérez, N. Ali, C. Costa J. Á. Carsí, I. Ramos, "Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology", Int. Conference on .NET Technologies, Plzen, Czech Republic, 2005.

[16] D.E. Perry and A.L. Wolf: "Foundations for the study of software architecture". ACM SIGSOFT Software Engineering Notes, 17(4):40 52, October 1992.

[17] C. Rolland, G. Grosz and R. Kla: "Experience with Goal-Scenario Coupling in Requirements Engineering", IEEE Int. Symposium on RE, June 7-11, 1999, Limerick.

[18] M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proc. 21st COMPSAC'97 Washington, DC, Aug. 1997.

[19] UML 2.0 Superstructure Final Adopted Specification http://www.omg.org/cgi-bin/doc?formal/05-07-04.

Analyzing Styles of the Modular Software Architecture View

Rogelio Limon Cordero and Isidro Ramos Salavert

Department of Information Systems and Computation, Technical University of Valencia,
Camino de Vera s/n E-46022 Valencia
{rlimon,iramos}@dsic.upv.es

Abstract. Software architecture views represent the basic structures of a complex software system. By means of these views, it is possible to shape the different concerns that appear in the requirements and design phases. A modular view specifies the elements that must be built in the detailed design, and the relationships that must be established among them. This paper makes an analysis of the styles present in the modular view. This work establishes how these styles can be shaped and analyzed by means of their relations.

1 Introduction

Software architecture is a key element in software development because it is the bridge between the requirements and design phases. The design of this architecture allows the user to determine the structures that will shape the software system and the way in which its elements communicate before proceeding with the detailed design. Each structure is really an architectural view, which contains architectural element relationships with each other.

The aim of this paper is to study this area by considering the architectural style as a relationship instead of a type of view. In addition, a proposal is presented to model the relationships in order to make the styles operative in the analysis phase of the modular architectural view.

2 The Modular Architectural View and Its Styles

From the definition presented in [1], the architectural view is considered as a specific perspective of each structure of the architecture.

There are several approaches about what the architectural view should be; among the more relevant, we can enumerate the following: SEI[1], Kruchten [2], and SIEMS[3]. Even though there are some differences among them about how they interpret a view, there are coincidences in the type of elements and relations that are considered in each one. The modular view is included in these approaches. The focus of this work is to analyze and deal with the relationships of elements of the modular view-type[1].

Limón, R., Ramos, I. Analyzing Styles of the Modular Software Architecture View. Proc. of the First European Conference on Software Architecture (ECSA 2007). 2007.
F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 275 – 278, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The relationships among the elements of a view-type adopt either some of the pre-established shapes known as architectural styles (or simply styles) or similar shapes. *Styles* define a set of features and constraints that typify the relationship among the architectural elements.

The basic styles in a modular view type are: *decomposition*, *uses, generalization* and *layered*. The links established among the architectural elements are relationships of dependence.

3 Establishing the Architectural Relationships

Considering that an architectural style is a specific type of a relationship that is established between at least two architectural elements, a mechanism must be established to identify, represent, and analyze such relationships in order to improve the architectural design process and to achieve a quantitative analysis.

Our approach is based on the design rules of [4], where a dependency matrix for analyzing relations among elements is proposed. In this paper, a basic binary relationship is used to represent the relationships in each one of the styles of the modular view-type.

Let's consider that a relationship between two elements (modules) named A and B can be represented by a relationship matrix as shown in Figure 1(a).

In the relationship matrix, V_x, and V_y (in a cell) represent values that indicate the relation between their corresponding row and column elements. Each value has a different interpretation to indicate the type of relationship. Using a representation of binary relational algebra, A V_x B means that A and B have a type of relationship that is represented by the value of V_x, where V_x and V_y are associated with a type of relationship (A_Relationship and B_Relationship) by means of a correspondence, i.e., types of relationships = {(0, null), (V_x, A_Relationship), (V_y, B_Relationship)}.

The range of values for V_x and V_y varies from 0 to R-1, where R is the number associated to the relationship types, and the ´-1` is due to the fact that the null relationship (0) must be excluded.

In this case, the dependence and decomposition relationships are used because the modular view only represents a static relationship; therefore, the value of R for the matrix is 2.

The decomposition and uses architectural styles are represented by means of matrices and their corresponding UML 2.0 representation, as shown in Figure 1(b).

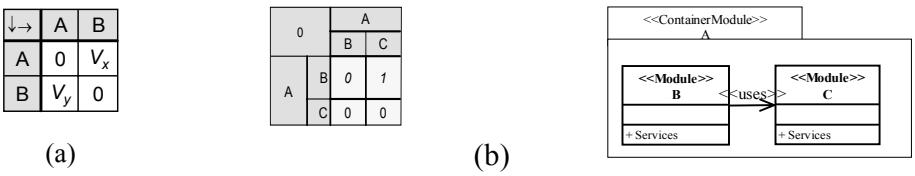


Fig. 1. (a) Relationship matrix between two elements, (b) Matrix for basic architectural styles of a modular view, and its corresponding representation in UML 2.0

4 Analysis of Architectural Styles

There are several methods for analyzing software architectures. The main goal of these methods is assess the quality attributes of the system. In a survey that was done on eight methods [5], it was shown how most of these techniques use qualitative criteria to assess the quality attributes of the system; for example ATAM [1][5], which is currently widely used.

The goal in this work is improve the current methods through the inclusion of indicators that evaluate the relationship types used in the software architecture views. These indicators use the relationship matrices as objects of analysis.

The indicators quantify the architectural styles used in each architectural view, taking into account the relationship types that their styles have.

In a modular view type, the styles link the modules through the <<uses>>, <<layered >>, and <<decomposition>> relationships. The interest in these relationships is focused on how the modules are distributed, i.e., if a *module* is overused by others, or if a *Container-Module* includes too many elements. The following indicators are identified to assess this: $UM_{i,m}$ means that the module *m* is used by the module *i*; LD_m indicates the level of dependence in a module *m*, i.e., LD is the number of modules that uses to the module *m*; DX indicates the maximum level of dependence in the view, i.e., $MAX(LD_m)$; IM_m indicates the number of modules included in the module *m;* TM indicates the total of container modules in the view; AM is the average number of modules included in the modular view; MD_m is the modular density of module *m*, where *m* is a module of the <<container>> class.

The valuations in some indicators are:

$$LD_m = \Sigma_{i=1} UM_{i,m} \quad \text{where } i \neq m$$
$$MD_m = NM_m / AM \quad ; \quad \text{where } AM = \Sigma_{i=1} NM_i / TM$$

These can be evaluated using the relationship matrix of the view to be analyzed. For example, $UM_{i,m}$ is obtained in a straightforward way accessing row *i* and column *m* from the matrix. LD_m for the element *m* is calculated by summing all the values of each row of the column where element *m* is located, and so on for the other indicators.

These indicators are useful for making comparisons between two or more views generated in the design process, or for restructuring a module when its MD > 1 or when it has a high value of LD.

5 Applying the Analysis

An analysis is applied to the software architecture for a purchase and reservation ticket system for any kind of passenger transport (bus, air plane, train, etc). This example was presented in [6].

In order to focus on the use of the architecture styles proposed here, let's suppose that the architectural modules derived from the requirements are: Query formation (QuerForm); Generating itinerary alternatives and routes (GeneItin); Making the reservation(Reservation), and Purchasing a ticket(Purchase). This architecture is shown in matrix and UML 2.0 notation in Figure 2.

For the modules shown in Figure 2(a), the dependence level (DL) for Reservation, Make-Itin and Purchase is: 1, 2 and 0, respectively. The average of modules (AM) included in this modular view is 2. The modular density (MD) for Make-Itin (the only container modular) is 1. This example case only indicates how to evaluate some indicators through the relationship matrices. However, an exhaustive quantitative analysis of a software architecture can be carried out with these matrices in order to evaluate its complexity.

0		Reservation	Make-Itin		Purchase
			QuerForm	Geneltin	
Reservation		0	1	1	*0*
Make-Itin	QuerForm	*0*	*0*	*1*	0
	Geneltin	*0*	0	0	0
Purchase		*1*	1	1	0

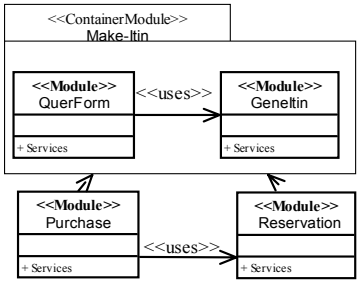


Fig. 2. The architectural modular view for a ticket purchase-reservation system, in relationship matrix and UML 2.0 representations

6 Conclusions

By modeling the architectural style as a type of relationship, we have proposed a novel way to analyze an architectural view. The identification and analysis of the basic relationships proposed here can be used in other view types.

The use of the relationship matrices allows us to: a) adapt some view types where there is dynamism among the element relationships; b) represent structural relationships as a composition of architectural views; c) make a quantitative analysis using relationship matrices.

Finally, it is shown that the application of the architectural styles improves the method of design of the software architectural views by making the analysis of relations in the architectural design process operational.

References

1. Bass, L., Clements, P., Kazman, R.: Software architecture in practice, 2[a] edn. Addison-Wesley, Reading (2003)
2. Philippe, K.: The 4+1 View Model of Architecture. Paper published in IEEE Software 12(6), 42–50 (1995)
3. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison Wesley, Boston, MA (2000)
4. Baldwin, C., Clark, K.: Design Rules: The Power of Modularity, vol. 1. The MIT Press, Cambridge, MA (2000)
5. Dobrica, L., Niemelä, E.: A survey on software architecture analysis methods. IEEE Transactions On Software Engineering 28(7), 638–653 (2002)
6. Limon, C.R., Ramos, S.I., Torres, J.J.: Designing Aspectual Architecture Views in Aspect-Oriented Software Development. In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganà, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3983, pp. 726–735. Springer, Heidelberg (2006)

Relating Software Architecture Views by Using MDA

Rogelio Limon Cordero and Isidro Ramos Salavert

Department of Information Systems and Computation, Technical University of Valencia,
Camino de Vera s/n E-46022 Valencia – Spain
{rlimon, iramos}@dsic.upv.es

Abstract. Software architecture views separate the concerns that are presented in phases previous to the detailed design phases of a software system. Each view represents a different structure of a system for its analysis and documentation. Although each view can be designed separately, they are linked because they are part of the same system. These links must be explicitly shaped and maintained in order to achieve synergy in their design and also to preserve consistency among the views when they change for evolution or maintenance reasons. This work proposes the use of the Model-Driven Architecture (MDA) approach to establish the links among the architectural views through a relation of models. This work shows how relations among software architectural views can be established by means of strategies used in MDA in a systematic way.

Keywords: Software Architecture, Modeling Driven Architecture.

1 Introduction

Nowadays, software architecture (SA) is used in the development of complex software systems to shape the fundamental structures that support the detailed design phase of these systems. Software architectural views (SAV) represent the different structures that a system can have and shape the concerns present at the design level in different ways.

Even though architectural views can be shaped separately, these views establish relations among each other that help maintain the cohesion of the system. These relations can be traced and their corresponding elements at lower levels of design can also be traced.

The relations of each architectural view must be identified and explicitly shaped in order to achieve the following objectives: i) generate a view using another view as reference, ii) make rules to assure conformance between the architectural view and the code, iii) preserve the consistency among views when a view is modified (for maintenance or evolution), and iv) document the software architecture.

The studies made to date do not fully cover all the objectives stated above. For objective (i), there are methods such as the Attribute-Driven Design (ADD) approach presented in [1], which uses a view to design another view. However, the relations between them not are explicitly specified and are not recorded. For objective (ii) there are many proposed methods; however their solutions as demonstrated are incomplete as stated in [2]. For objective (iii), a specific algebra to deal with the consistency is

O. Gervasi and M. Gavrilova (Eds.): ICCSA 2007, LNCS 4707, Part III, pp. 104 – 114, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Limón, R., Ramos, I. Relating Software Architecture Views by Using MDA. Proc. of the ICCSA 2007 International Conference on Computational Science and its Applications. 2007.

used in [3], but it only focuses on one type of view. [4] deals with relations of multiple views in a formal way, but its concept of view is too specific to apply to other types of views. For objective (iv), there are many studies for documentation, e.g. [5].

The main goal of this work is to use a Model-Driven Architecture (MDA) proposal to deal with the view relations and to achieve the objectives described above. This work presents a strategy to relate two SAVs and generate one SAV from another one, by means of relations and the transformation of models.

This paper is structured as follows: Section 2 shows the relation between SAV and MDA. It explains how to use the MDA to relate SAVs with each other by means of a transformation process among the architectural models. Section 3 identifies the relations among the meta-models of the SAVs considered. Section 4 shows how to relate SAVs at the model level. Section 5 illustrates this approach in a case study, and section 6 presents our conclusions.

2 Relating SAVs and the Model-Driven Architecture

2.1 Software Architecture Views

The SA is shaped by a set of views that represent the different structures of a software system [1]. The views can be used to separate concerns and several models can be used to represent them.

There are several approaches that describe what the architectural view should be. Among the most relevant are the following: SEI [1], IEEE-1471 [6], and Kruchten [7]. Even though there are some differences in how they interpret a view, there are coincidences in the type of elements and relations that are considered in each one. In all of these approaches, components and connectors are considered as basic building blocks, and the module element is included in different ways.

The focus in this work is to analyze and deal with the relations between at least two types of architectural views. Therefore, the component-connector(C&C) and modular view-types proposed in [1] have been chosen as the most appropriate approaches to apply to our proposal.

The main goal is to shape the correspondence rules at the meta level and perform the transformation at the model level.

Figure 1(a) shows an example of two correspondence relations at the meta-model level: in this case, R1 indicates that each module element (Module) of the *<<Modular View>>* has a correspondence with an element (component) of the *<<C-C View>>*, and R2 indicates that a *<<Uses>>* of the *<<Modular View>>* has a correspondence

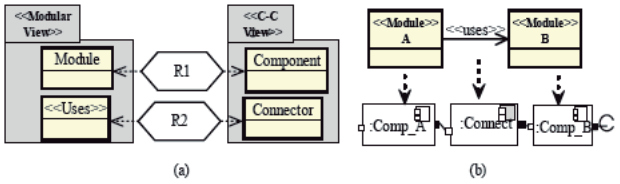


Fig. 1. Example of: (a) Relations between the two views at the Meta-model level, (b) Transformation from a view model to another view model

with a connector element of the *<<C-C View>>*. These relations are used to achieve the transformation at the model level as shown in Figure 1(b). This is an example of objective (i) of the relation of software architecture views, described in section 1.

2.2 The Model-Driven Architecture and the Software Architecture Views

The MDE is a novel paradigm that reduces the software complexity introduced by the current diversity in technology. It uses methodologies and support tools that are based on models with a high level of abstraction (meta-models) instead of those of a specific technology. MDA [8] is a proposal of OMG (Object Management Group) for MDE.

MDA creates system models with a high level of abstraction. These models can be refined using a series of transformations to obtain different models at the code level. [9]. The highest model is called PIM (Platform-Independent Model), and the model of a specific platform is called PSM (Platform-Specific Model). The PSMs are generated by a transformation from PIM to PSM.

SAVs can be shaped as PIMs, even though they represent specific systems. They have features that are applicable to a set of similar systems (families of software systems) so that the same SAV can be adapted to systems that have different platforms, i.e., to PSMs. Furthermore, SAVs are formed with elements of a high level of abstraction, which are shaped by means of languages (ADLs) or notations that are independent of the technological platform. These elements can be mapped at code level. Therefore, the MDA can be used not only to refine a SA, but also to make transformations among architecture views that are at the same level [9].

2.3 Using MDA to Build Relations Among SAVs

Three types of relations among the SAVs studied in this work have been identified: *i*) the relations that are present between the modular view and the C&C view. These generate a view at the model level using each other as reference; *ii*) the relations that are formed between the C&C view model and other models that define the interactions among the components (scenarios, maps of use cases).These other models are used as references to refine the C&C view model; and *iii*) the relations established when a view is adapted to a specific system (at different levels of abstraction).

The relations *i* and *ii* are shaped using the MDA frame as shown in Figure 2. The Meta Object Facility (MOF) [10] provides the language for expressing the models. It

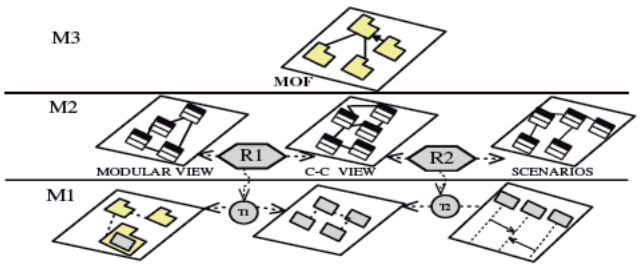


Fig. 2. Relations among meta-models and transformation among models

is used to specify itself (level M3) and to specify the meta-models of the views and their interactions, which are given by the scenarios meta-model (level M2). The relations R1, R2 and R3 are also established at level M2. R1 is related to *i*, and R2 is related to *ii*.

The next level (M1) is where the transformations T1 and T2 are carried out. A first version of the C-C view model is obtained by transformation T1, using the modular view model as source. This first version is refined by means of T2 using the scenarios model a refinement.

2.4 Transformation Process of the Architectural Views Models

In order to achieve the transformations among the view models, a process that allows the design of the C&C view using the modular view and the scenario model as inputs, is proposed.

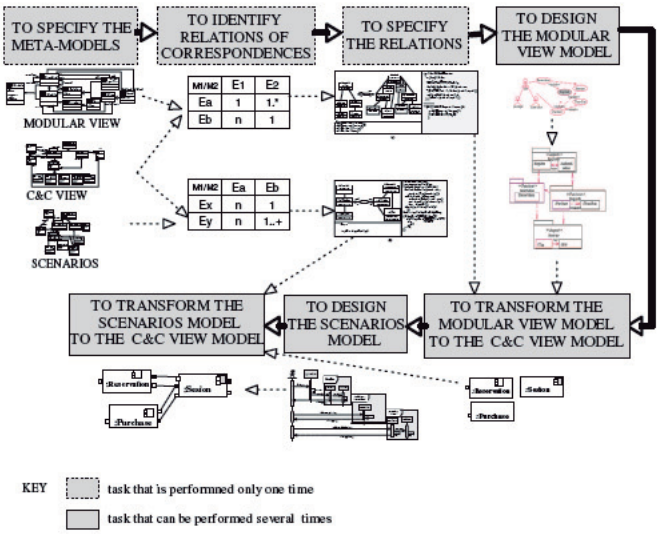


Fig. 3. The tasks and elements involved in the transformation process of the architectural view models

The tasks and elements involved in this transformation process are shown in Figure 3. The meta-model specifications of the modular view, the C&C view, and scenarios are the first elements that are required to start this process. Next the correspondence relations between the modular view and the C&C view, and between the scenarios and the C&C view are identified, and specified by mean of a language and/or notation. The QVT-Relational language and the MOF notation are used for this task. All these tasks are performed only one time since their products (elements) are used as a template for the transformation models.

The design task of the modular view model is the first task that can be performance several times as required. This task uses requirement as reference, these requirements can be expressed by use cases.

The modular view model and the relations established between the two meta-model views are used as source in the transformation task, which produces a first version of the C&C view model. Next, a second transformation is carried out to refine to the produced model. To do it the scenarios model and the first version of the C&C view model are used as source to produce a refined version of this view model.

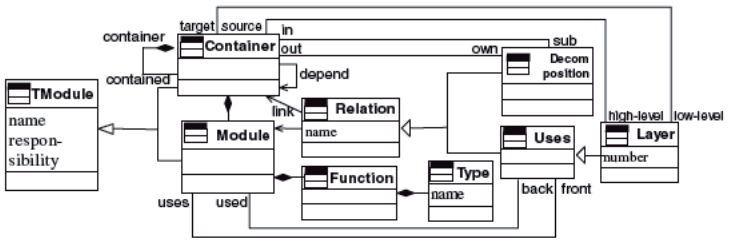
3 Specifying the Meta-models

For the conformation and specification of the meta-models of the modular and C&C views, several proposals [5][6][7] have been analyzed. The MOF graphic notation has been used for the specification of these meta-models.

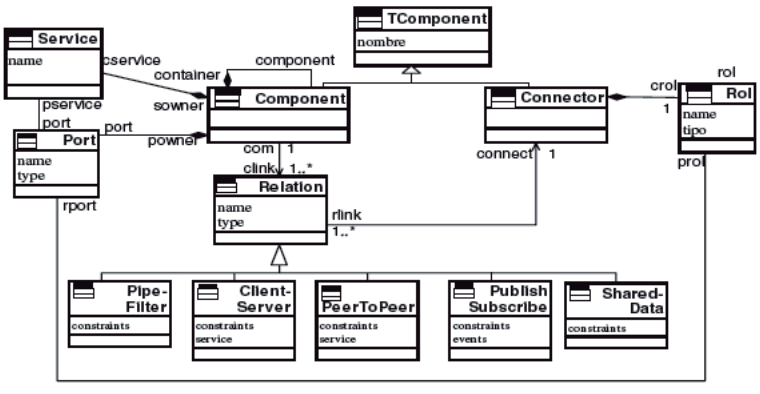
3.1 Specification of the Modular View Meta-model

The main element considered for this view is the module. We have made a distinction between a module that is used as a container for more modules and the module that only carries out a single function.

In Figure 4(a) shows the meta-model for this view. The relation class represents the styles presented in [5] (decomposition, uses and layers). These styles are treated as



(a)



(b)

Fig. 4. (a) Modular view meta-model, (b) C-C view meta-model. (MOF notation).

relations since what distinguishes one style from another is the type of relation. The labels in the links are useful for indicating how the relation is made; for example in Figure 4(a) the *'uses'* label indicates which element of the *Module* class will be used by another element of the *Module* class.

3.2 Specification of the C-C View Meta-model

The *C-C* view meta-model is depicted in Figure 4(b). This Figure shows the *Component* class and *Connector* class as the main elements. Both are derived from a more general component class (*TComponent*). Similarity to the above meta-model, the way in which these elements are related corresponds to their styles or relations (Pipe-Filter, Client-Server, PeerToPeer, Publish-Suscribe, Shared-Data). These styles are inherited from the *Relation* class. This class links the components by means of the *Connector* class.

The interactions among the components of this view are refined using the scenarios meta-model; this meta-model specifies how the components (objects) communicate (by means of events or messages). The scenario meta-model is specified by the UML 2.0 meta-model [11].

4 Establishing Relations Among the Meta-models

Before the relations are shaped, a language must be chosen to represent them correctly. For the specific case of MDA, OMG proposes the Meta Object Facility (MOF) to express the meta-models. It uses to Query/Views/Transformations (QVT) [10] to establish the relations among these meta-models. Specifically, the QVT-relational language is used to describe the relations.

4.1 Identifying Correspondence Relations

The correspondence among each element of the meta-models previously specified must be identified, taking into account the way in which their rules of relation are represented through QVT. The source and target meta-models must be identified first. In this case the source meta-model is the modular view. The target meta-model is the C&C view model, since its model is obtained from the model of the modular view. This means that the rules considered in the relations for the elements of the modular view can only be of type "**check only**", to verify the elements of the modular view, and the type "**enforce**", to create the elements of the C&C view.

Table 1. Relations identified in the modular and C&C meta-models

Relation	Type	Classes involved by view	
		Modular view	C-C view
moduleToComponet	top	Module	Component
functionToService	-	Module, Function, Type	Component, Service, Port
rUsoModToConnector	top	Uses, Module, Service	Connector, Role, Component, Port; Service
rCompoModToCom	top	Composition, module	Component

Note that not all the elements between the models considered have correspondence relations. Table 1shows some of the identified correspondence relations, their name, type (according QVT), and the elements involved (classes).

4.2 Specification of the Relations Between the Software Views

The relations shown in Table 1 shown are specified by means of a MOF diagram. The code for these relations is written in the QVT-relational, it and the diagrams are shown in Figure 5.

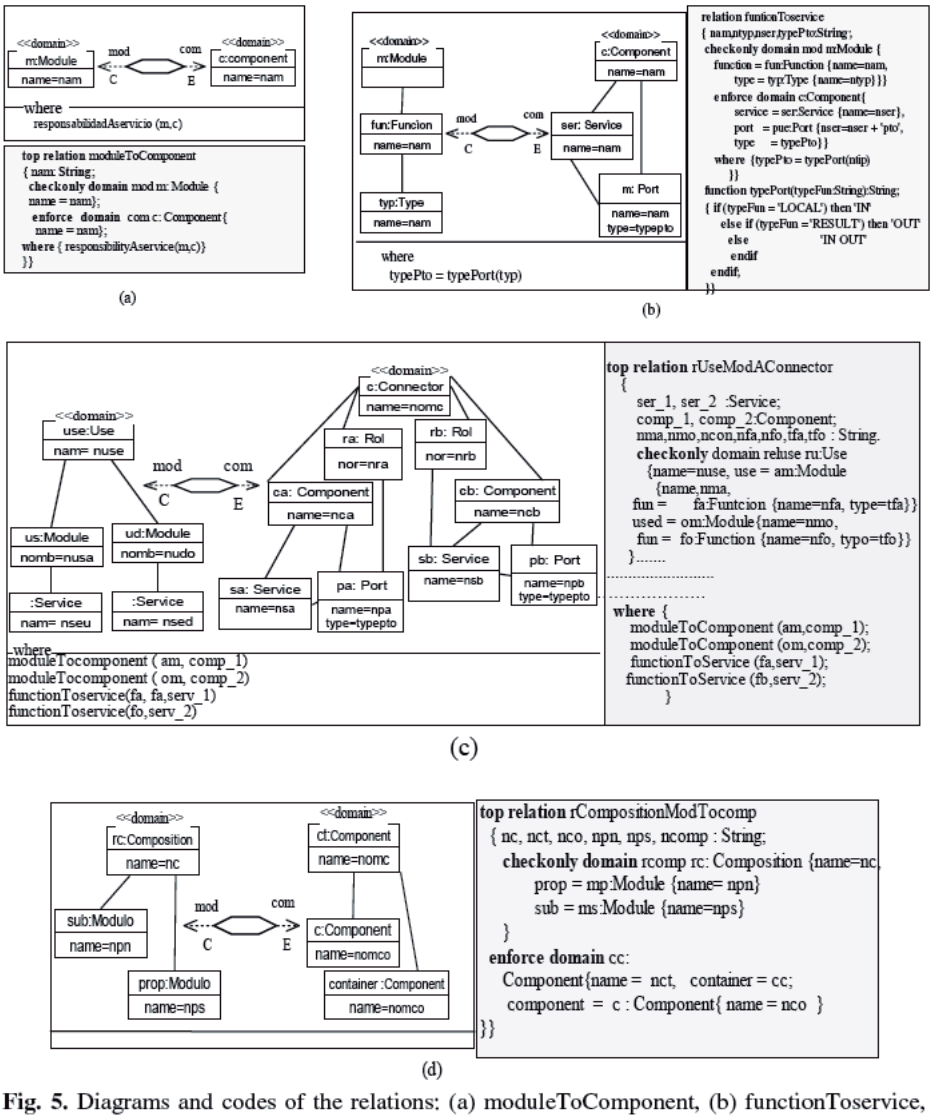


Fig. 5. Diagrams and codes of the relations: (a) moduleToComponent, (b) functionToservice, (c) rUseModToConnector, (d) rCompositionModTocomp

Relation *moduleToComponent*
This relation maps each module with a component. Figure 5(a) shown two types of relations: the *checkonly* type and the *enforce* type. The object of the *Module* domain is of the *checkonly* type. In contrast, the object of the *Component* domain is of the *enforce* type. This creates an object of the *Component* class that is related to the *Module* class. The *where* clause indicates a call to the *functionToService* relation, which relates an object of the *Module* class with an object of the *Component* class. The diagram and the corresponding code are shown in Figure 5(a).

Relation *functionToService*
This relation implies that a function of the module meta-model will generate a service of the Component class. When this occurs, the type of the Function will generate a *port*. In the *where* clause, the name of the *port* is obtained by calling the function *typePort*.

When the relation is executed, the classes that are in the source meta-model can only be verified and the classes that are in the target meta-model will be created (only the *ServiceToPort* class). The diagram and the corresponding code are shown in Figure 5(b).

Relation *rUseModToConnector*
The *uses* relation is transformed from the modular meta-class to a link between a connector and two components in the C&C meta-class. This is shown in Figure 5(c). The creation of objects for this relation is from 1 to *n* because a relation of two components is generated through a connector. Figure 5(c) shows part of the code to illustrate how the relations are invoked in the *where* clause to create the relations among module, component, and *functionToService*.

Relation rCompositionModToComp
The set of modules will also generate a set of components. However, in this case when a component is created (container) a subordinate is created (c) inside it. This is shown in Figure 5(d).

5 Applying Transformation at the Model Level

To illustrate how the transformations at the model level are applied in the transformation process (see Figure 3), the case study presented in [12] is used. This study consists of designing software architecture for a purchase and reservation ticket system for any kind of passenger transport (bus, air plane, train, etc) where the passengers can reserve and buy a ticket, or just search the Internet. After introducing the query, the system will generate several tentative itineraries as a first step. Next, the passenger can choose an itinerary and make the reservation or the purchase.

As a step previous to the transformation, the modular architectural view model is designed from the specification given above using the method proposed in [12]. The OMONDO tool developed for Eclipse [13] is used to design this model (and its corresponding meta-model) as shown in Figure 6. The modular view model is depicted in the window named *TicketPRModView* (Fig. 6, right). This model is made up of five classes (modules): *QuerForm, GeneIti, Purchase, Reservation,* and *RegisterAuthen*. These are all linked with each other through *use* relations. Next, the transformation

task for producing the C&C view model (target) is performed. To do this, the previously designed model is used as the source model. The relations established in section 5 are the rules that are applied to generate the new model (using the meta-model as templates). In this case, the *moduleToComponent* and *rUsoModToConnector* relations were applied. The transformation is carried out with the MOMENT [14] tool, which is a plug-in developed on ECLIPSE.

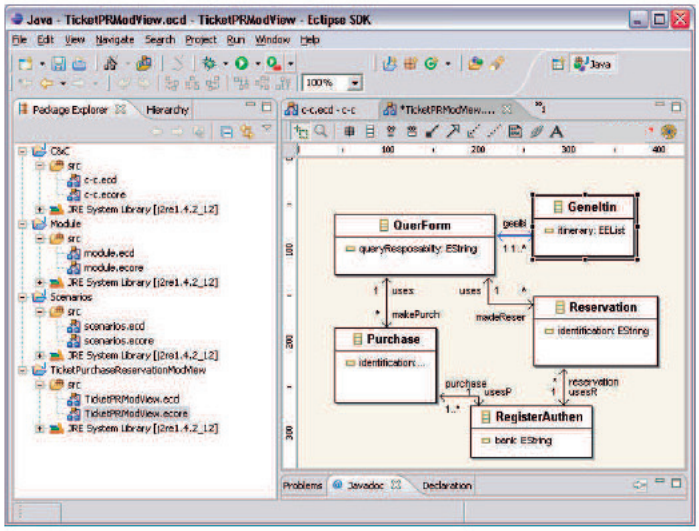


Fig. 6. The modular view model design of the reservation–purchase ticket system

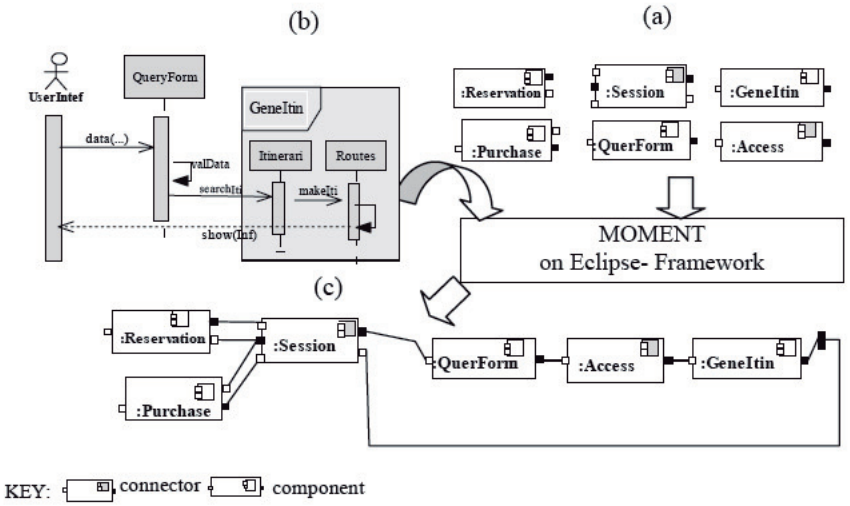


Fig. 7. The second transformation of models: (a) The C&C view model generated in the first transformation; (b) the scenario model; (c) The definitive C&C view model

The transformation generates the C&C view model shown in Figure 7(a). As this figure shows, a component was produced for each model, and each *use* relation generated a connector (*session and access*). However, these components and connectors are not linked with each other because their communication has not yet been modelled. A second transformation must be carried out to do this. The scenarios model is required for this second transformation. It is designed taking into account the requirement specification (not shown here). The designed model is shown in Figure 7(b) (partial).

The scenarios model and the C&C architectural view model are used as source models in this second transformation. As before, the MOMENT tool is used for this task; however, in this case, two source models are combined as shown in Figure 7. The final C&C architectural view model is then generated. This is shown in Figure 7 (c). This model and the modular architectural view model make up the software architecture of the purchase and reservation ticket system.

6 Conclusions

By modeling software architecture using MDA, we have proposed a novel way to design an architectural view. This approach can be extended to other types of views in software engineering.

The use of the transformation process among models proposed here, allows us to: a) establish meta-models for two views (modular and C&C), and identify and establish correspondence relations between them; b) generate a new view using another view as reference; c) refine the new view through a second transformation combining the new view and the scenarios model to obtain the C&C view model; d) adapt the MOMENT development tool to make this these transformations; e) apply a consistent transformation process attached to the standard specification of the OMG (MDA) by means of the QVT relation language and the MOF graphic notation to represent the relations and the transformation rules; f) adapt and use tools consistent with the language and notation used; g) apply a similar treatment to assure conformance between the C&C architectural view and the code (not included here); h) prove consistency (using rules *checkonly*) and contribute to improving the documentation.

This study has also shown that the application of MDA improves the design process of the software architectural views by doing it in a systematic and automatic way.

References

1. Bass, L., Clements, P., Kazman, R.: Software architecture in practice, 2ª ed. Addison-Wesley, Reading (2003)
2. Shaw, M., Clements, P.: The Golden Age of Software Architecture. IEEE Software 23(1), 31–39 (2006)
3. Muskens, J., Bril, R.J., Chaudron, M.R.V.: Generalizing Consistency Checking between Software Views. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pp. 169–180 (2005)
4. Fradet, P., Le Métayer, D., Périn, M.: Consistency checking for multiple view software architectures. In: Proceedings of 7th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 410–428 (1999)

5. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architecture, Views and Beyond. Addison-Wesley, Reading (2002)
6. IEEE Product No. SH94869-TBR, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (2000)
7. Philippe, K.: The 4+1 View Model of Architecture. Paper published in IEEE Software 12(6), 42–50 (1995)
8. OMG, Object Management Group: Model Driven Architecture (MDA) (2004) http://www.omg.org/cgi-bin/doc?formal/03-06-01
9. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison Wesley Professional, Reading (2003)
10. OMG: Revised Submission for MOF 2.0. Query/View/Transformations RFP (ad/2002-04-10); OMG Document ad/2005-07-1 (2005)
11. Unified Modeling Language 2.0. Superstructure, http://www.omg.org/docs/formal/05-07-04.pdf
12. Limon, C.R., Ramos, S.I., Torres, J.J.: Designing Aspectual Architecture Views in Aspect Oriented Software Development. In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganà, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3983, pp. 726–735. Springer, Heidelberg (2006)
13. Eclipse: http://www.eclipse.org/
14. The MOMENT Project, http://moment.dsic.upv.es/

Analyzing and Designing Software Architecture Views driven by their Relationships

Rogelio Limon Cordero[1], Isidro Ramos Salavert[1],
Maricela Morales Hernández[2], Jorge Zaráte Perez[2]
[1]*Department of Information Systems and Computation, Technical University of Valencia, Camino de Vera s/n E-46022 Valencia –*
{rlimon,iramos}@dsic.upv.es
[2]*Departamento de Sistemas y Computación, Instituto Tecnológico de Oaxaca*
Av. Victor B. Ahuja 125 esq. Calz. Tecnológico, Oaxaca, México
{mmorales, jorge.zarate}@itoaxaca.edu.mx

Abstract

Software architecture views represent the basic structures of a complex software system. By means of these views, it is possible to shape the different concerns that appear in the requirements and design phases. Each view specifies the elements that must be built up in the detailed design, and the relationships that must be established among them. This paper makes an analysis of the relationships present in two view types (modular and component-connector views). This work establishes how these relationships can be shaped by means of theirs styles in an operative way. This approach allows a better architectural design through the management of the relationships, using their styles as the basis for the architectural structures and leads to a more precise analysis in the design for architectural views.

1. Introduction

Software architecture is a key element in software development because it is the bridge between the requirements and design phases. Its design allows the user to determine the structures that will shape the software system and the way in which their elements communicate, before proceeding to their detailed design. Each structure is really an architectural view, which contains architectural element relationships with each other.

Architectural view design implies identifying and specifying the elements and establishing their relationships. This last task is perhaps one of the most important because the relationships constitute the basis of the architectural structure.

The design methods used in [1][2][3] establish architectural relationships by using a set of prototypical structures called architectural patterns as a reference, also known in other works as architectural styles. However, no analysis of these styles has been made in these methods nor have they been used in an operational way in the design process.

The aim of this paper is to research this area by considering the architectural style as a relationship instead of a type of view. In addition, a proposal is presented to model the relationships in order to make the styles operative in the design and analysis phases of the modular and component-connector architectural views, with all of this being driven by an iterative and descending method presented in [4].

This paper is structured as follows: Section 2 proposes meta-models for the types of modular and component-connector views to establish the relation between the styles and the other architectural elements. Section 3 describes the types of architectural relationships, which are represented with matrices and explains how the proposed mechanism operates. Section 4 proposes an adaptation of these matrices to the architectural styles. Section 5 proposes a quantitative analysis of the architectural relationships. Section 6 shows how the architectural styles are applied in the design method, which is illustrated in a case study. Finally, the conclusions are presented.

Limón, R., Ramos, I., Morales, M. Zárate, J. Analyzing and Designing Software Architecture Views driven by their Relationships. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007.

2 Architectural Views and their Styles

From the definition presented in [1], the architectural view is considered as a specific perspective of each structure of the architecture, whose formation rules are contained in a viewpoint [5]. We will use the term view-type as the specification that defines the elements that an architectural view contains and the type of relations that can be established among them.

There are several approaches about what the architectural view should be; among the more relevant, we can enumerate the following: SEI[1], SIEMES[2], Kruchten [6]. Even though there are some differences among them about how they interpret a view, there are coincidences in the type of elements and relations that are considered in each one. In all of these approaches, the components and connectors are considered as basic building blocks in some of their views, and the module element is included in different ways. The focus in this work is to analyze and deal with the relationships of elements of this kind. Therefore, the component-connector(C-C) and modular view-types [1] have been chosen as the most appropriate approaches to apply to our proposal. Furthermore these approaches contain the two types of relationships present in almost all software architectures of systems, dynamic and static relationships. The C-C view-type basically includes communication relationships (dynamic), and the modular view-type includes only structural relationships (static).

The relationships among the elements of a view-type adopt either some of the pre-established shapes known as architectural styles (or simply styles) [6] or similar shapes. Styles define a set of features and constraints that typify the relationship among the architectural elements. In this paper, the term basic style is used to denote a specific and well-delimited style, and the term heterogeneous style is used to denote when a style is composed of two or more basic styles.

The basic styles used in software architecture are studied in [7],[8]. In [7], each style is associated to a specific type of view, which means that an architectural style cannot be inside of more than one type of view. This is derived from the fact that each type of view deals with different types of concerns [5], and a specific style is used for shaping each type of concern.

Figure 1 shows the two meta-models proposed. Figure 1 (a) depicts the meta-model for designing a modular view type (<<Modular-VT>>), its elements are <<Modules>>, which are linked with each other according to three types of styles: <<Decomposition>>, <<Uses>> and <<Layered>>. The links established among the elements are relationships of dependence.

Figure 1 (b) depicts the meta-model for designing a component-connector view type (<<C-C-VT>>) specifying its relation with styles. This meta-model shows the elements and relationships that make up this type of view. The primary elements are <<Component>> and <<Connector>>, which are linked by their <<Ports>>. The <<Ports>> can contain <<Interfaces>> to allow several modes of access. Both the components and connectors are determined by the style types: <<Pipe&Filter>>, <<ClientServer>>, <<PeerToPeer>>, <<PublishSubscribe>>. Also, other styles can be included (<<OtherStyle>>).

Note that, as Figure 1 shows, the composition relationship in a <<ModStyle>> and in a <<C-CStyle>>, allows the integration of self-contained styles in both the (a) and (b) meta-models, which makes it possible to build heterogeneous styles.

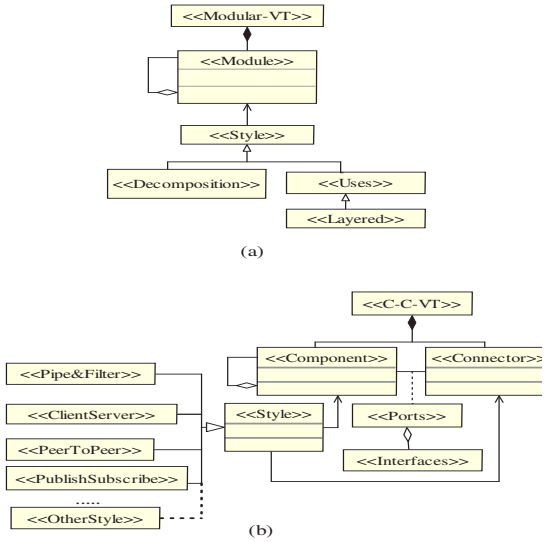


Fig.1 Meta-model for: (a) The modular view type, and (b) The C-C view type.

3. Establishing the Basic Architectural Relationships

Considering that an architectural style is a specific type of a relationship that is established between at least two architectural elements, a mechanism must be established to identify, represent, and analyze such relationships in order to improve the architectural design process and to achieve a quantitative analysis.

In this paper, we propose a strategy that creates a basis for constituting each style from the relationship between two elements.

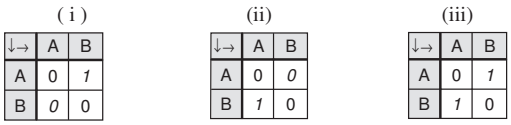


Figure 3. Relationship cases with R=2

Our approach is based on the design rules of [9], where a dependency matrix for analyzing relations among elements is proposed. This matrix has been used for the hierarchical and dependency relationships [10]. In this paper, a basis binary relationship [11] is used to represent the relationships in each one of the styles of the component-connector and modular view-types. This is done to be able to do the following: a) model the communication relationships among components, b) model the composition relationship among components and modules, c) model the use and layer relationships, and d) represent these relationships with matrices to obtain the architectural model.

3.1 Basic Relationship

Let's consider that a relationship between two elements (components or modules) named A and B can be represented by a relationship matrix as shown in Figure 2.

↓→	A	B
A	0	V_x
B	V_y	0

Figure 2. Relationship matrix between two elements (binary relation)

In the relationship matrix, Vx, and Vy (in a cell) represent values that indicate the relation between their corresponding row and column elements. Each value has a different interpretation to indicate the type of relationship. Using a representation of binary relational algebra [11], A Vx B means that A and B have a type of relationship that is represented by the value of Vx, where Vx and Vy are associated with a type of relationship (A_Relationship and B_Relationship) by means of a correspondence, i.e. types of relationships = {(0, null), (Vx, A_Relationship), (Vy, B_Relationship)}

Note that the matrix does not include self-relationships (reflexive) among the elements, which is indicated by the value 0 in their corresponding cells.

The range of values for Vx and Vy varies from 0 to R-1, where R is the number associated to the relationship types, and the ´-1` is due to the fact that the null relationship (0) is included in R.

In the matrix, R2–1 relationship cases between two elements can be generated (the non relation case is excluded). For instance, if two elements (A and B) can be related in only one way, then R=2 (null included), and the range of values for Vx and Vy will be {0,1}. Thus, 2 different relationship cases can be established between A and B. Figure 3 are shows all the possible cases with these values.

If the value 1 is associated with a dependence relationship, then each case is read as: (i) A is dependent on B, (ii) B is dependent on A, and (iii) A and B are mutually dependent. Note that the cases (i) and (ii) represent the same relationship type (simple dependent) because this relationship is symmetrical; on the other hand (iii) represents a different type of relationship (interdependence).

The following annotations complement this proposal: a) R depends on the number of styles that are considered in each type of view. b) Each matrix is used to analysis and build relationships, and to maintain these relationships for evolution and maintenance.

3.2 Communication Relationships

Communication relationships are given in the component-connector view-type, where a component communicates with another component by means of a connector. The components have a specific computational behavior, such as client, server, filter, object, or database. The connectors assume the role of coordinating and controlling the communication between two components, or among several components. Therefore, the connectors are in charge of the way communication is done. The interactions that are attributable to the connectors are the following: streams, data access, procedure calls to invoke services, send of messages, and events. These are described in [8].

Types of communication relationships are established according to what it is communicated and how it is dealt with. There are two basic sets of these relationships: a) communication of data (DC), similar to streams, and data access; and b) sending of control information (CC), similar to procedure calls, sending of messages, and events. Different type of communications can be formed using these two kinds of relationship by means of the relationship matrix.

The interpretation of a communication relationship using the relationship matrix is as follows: (see Figure 2) starting from the first row, the component of each

row (A or B) communicates something to the component of its corresponding column (A or B). The values Vx, and Vy in one cell are used to specify the type of communication that will be carried out by the connectors.

The values for generating the relationship matrices for the two types of communication described above are: R= 3; Vx, Vy = {0, 1, 2}, where their correspondences are: {(0, null), (1, DC), (2, CC)}. There are eight relationships that can be generated from these two basic relationships.

Figure 4 shows the possible cases that can be built with R=3, the cases presented in Figure 3 are not included. The following notation is used to clarify the communication relationships: → indicates communication of data in one direction; ↔ indicates communication of data in two directions; => indicates control communication in one direction; and <=> indicates control communication in two directions.

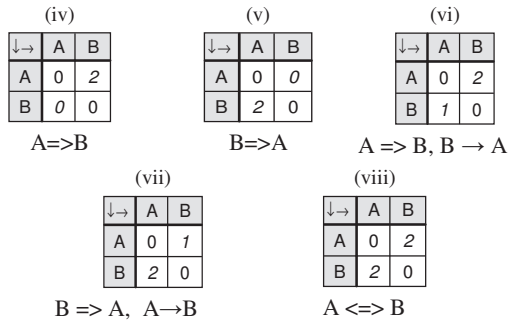


Figure 4. Relationship cases with R=3

Note that only five of the eight cases are different (see Figure 3 and Figure 4). Case (ii) is symmetrical with the case (i). The same happens with cases (iv) and (v), and with cases (vi) and (vii). The (i) and (iii) cases indicate that there is a communication of data from A component to B component, furthermore in case (iii) there is another communication from B to A. The cases (iv) and (viii) indicate that there is a control communication from A to B, furthermore in case (viii) there is another communication from B to A. The case (vi) combines a data communication with one control communication.

3.3 Composition Relationship

To accomplish the relations described in the meta-models of the modular and component-connector views (shown in Figure 1), the relationships among the components or among modules are called composition relationships, which must be taken into account. This type of relationship allows us to combine several basic styles in order to create a heterogeneous style. Therefore, the relationship matrix must be adapted for this purpose.

When an element (module or component) contains other elements, the corresponding rows and columns of the matrix relationship are divided according to the number of elements. These must be clustered in a container element which is shown in Figure 5(a). A specific case with k = 3 is shown in Figure 5 (b). It represents a relationship that is divided into sub-elements of the "D" element, where the D element has a relationship with the "A" element (Figure 5 (c)). This establishes the relations among sub-elements and their relation to the other elements.

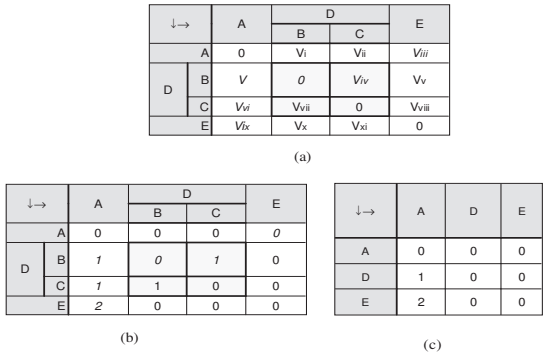


Figure 5. Composition relationship: (a) general, (b) a specific example for k=3, (c) hiding the sub-elements

3.4 Dependence Relationship

This type of relationship is only applied in modular view-types, because it corresponds to the links that are established when an architecture is divided into elements called modules. In this case, a module can be dependent on another module (or not be dependent on any module), because it requires using the services of another module. If both linked modules are in an even plane (layer) this relationship is called <<uses>>; however if these modules are in different planes its relationship is called <<layered>> (see Figure 1). Note that in this type of relationship, the flow of communication does not matter because their relationships only indicate how the elements are related structurally.

The relationship matrix for representing the dependence relationships is defined with R=2. Its possible cases are shown in Figure 3, where the value 0 indicates no dependence, and the value 1 indicates that there is a dependence. The matrix for composition

relationships is also used when a module includes others modules (<<decomposition>> style). However for a <<layered>> style it is necessary to specify the layer where a module is locted. For this reason, the number of layers is included in the upper left corner of the relationship matrix as shown in Figure 6. One column is added to indicate which modules of the different layers are linked.

Num. Layer		IdEle	A	D		E
				B	C	
	A	X	0	v	v	v
D	B	–	v	*0*	*v*	v
	C	–	v	v	0	v
	E	*Y*	v	v	v	0

Where: NumLayer = 0,1,2… V = {0, 1}
IdEle = Identification (literal or number) of a element of a matrix of different layer

Figure 6. Dependence relationship matrix (with sub-elements)

4. From the Relationship Matrix to the Architectural Style

4.1 Architectural Style Representations for the Component-Connector View-type

The component-connector view shows the dynamism of software architecture, therefore the communication links are established in the relationship between their elements. These links are typified by the architectural styles belonging to this view-type. The styles set the standard of the element relationships, and determine the behavior of the elements.

Each style is associated to a relationship matrix, where its elements are the components and where the cells of this relationship matrix are the connectors of the class that correspond to its style. The architecture is represented in UML 2.0..The basic notation employed here is shown in Figure 7.

The architectural styles included in Figure 1 (b) are represented by means of matrices and their corresponding UML representation, as shown in Figure 8.

In this case a matrix relationship with R=3 is used. This value makes it possible to generate the communication relationships of the styles considered in Figure (1) b).

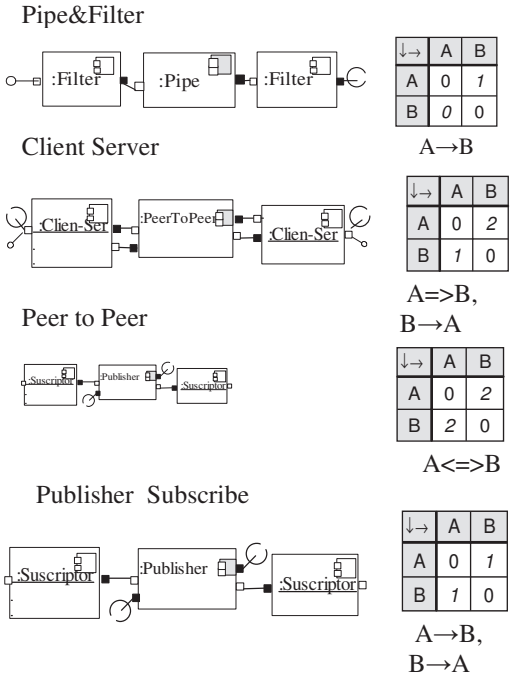
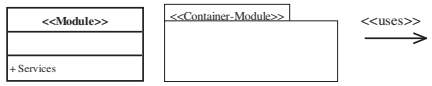


Figure 8. Matrices for basic architectural styles of the component-connector view and their corresponding UML representation

4.2 Architectural Style Representations for the Modular View

In this case, the dependence and composition relationships are used because the modular view only represents a static relationship; therefore, the value of R for the matrix is 2. The corresponding UML 2.0 notation for this view is shown in Figure 9.

The architectural styles included in Figure 1(a) are represented by means of matrices and their corresponding UML representation, as shown in Figure 9.



Key: UML 2.0

Figure 9. Basic notation for the modular view representation

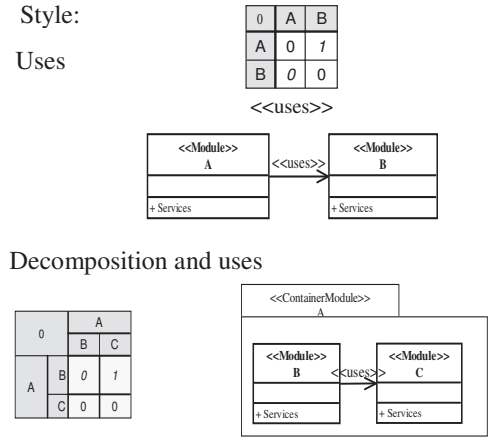


Figure 10. Matrices for basic architectural styles of a modular view, and their corresponding representation

5. Architectural styles Analysis

There are several methods for analyzing software architectures. All of them have as a main goal to assess quality attributes of the system, such as availability, security, modifiability, testability, manageability, performance, usability and son on. This is done using different techniques and different criteria for assessing one or several of these attributes. In a survey that was done on eight methods [12], it was shown how most of these techniques use qualitative criteria to assess the quality attributes of the system; for example ATAM [1][12], which is currently widely used. Very few methods use quantitative criteria to assess the quality attributes. It is import to note that in all methods the intervention of stakeholders in necessary for making decisions.

The goal in this work is not to propose a new method of analysis, but to improve the current methods through of the inclusion indicators that evaluate the relationship types used in the software architecture views. This indicators use the relationship matrices as objects of analysis. By means of these matrices we can determine how the architectural styles are used and how they affect some of the quality attributes of the system. Only a few indicators are included in this work to illustrate how they can be evaluated using relationship matrices.

The indicators quantify the architectural styles used in each architectural view, taking into account the relationship types that their styles have.

In a modular view-type the styles link the modules (<<Module>> and <<ContainerModule>>) through the *<<uses>>, <<layered >>, and <<decomposition>> relationships. The interest in these relationships is focused on how the modules are distributed, i.e. if a <<module>> is overused by others, or if a <<Cointainer-Module>> includes too many elements. The following indicators are identified to assess this:*

$UM_{i,m}$: means the module m is used by the module i.

LD_m : Level of dependence in a module *m*, is the number of modules that use module *m*.

DX : Maximum level of dependence in the view, i.e. $MAX(LD_m)$

IM_m : Number of modules included in the module *m*

TM : Total of container modules in the view

AM : Average number of modules included in the modular view.

MD_m : Modular density of module *m*, where *m* is a module of the <<container>> class.

The valuations in some indicators are done:

$LD_m = \Sigma_{i=1} UM_{i,m}$ where $i \neq m$

$MD_m = NM_m / AM$; where $AM = \Sigma_{i=1} NM_i / TM$

These can be evaluated using the relationship matrix of the view to be analyzed. For example UMi,m is obtained in a straightforward way accessing row i and column m from the matrix. LDm for the element m is calculated by summing all the values of each row of the column where element m is located, and so on for other indicators.

These indicators are useful for making comparisons between two or more views generated in the design process, or for restructuring a module when its MD > 1 or when it has a high value of LD.

A similar analysis is made for the C-C View-type, which is oriented to the communication relationship, however this is not included for reasons of brevity.

6. Applying the Styles in the Design of the Architectural View

6.1 Using Architectural Styles in the View Design Method.

Designing a view-type means applying a method to constitute its elements and the architectural

relationships, which, in this paper, are expressed in UML 2.0.

The design method proposed in [4] is used. The activities that are related to our proposal are shown in Figure 11. Our attention is focused on: "Shaping architectural relationships" which is the key to improving the design process by means of style relationships.

The starting point is the initial architecture derived from the functional requirements (expressed by the use cases) and the modules to be decomposed (using the specification of each use case). The found modules are linked or decomposed according to their architectural style; next, a component is derived from each module so the task now consists in finding the relation among the components that will constitute the architecture. The architectural relationships that are established with the architectural basic styles are applied for each architectural view. Their activities are detailed in Figure 11 (b).

The first activity consists of identifying a pair of elements (modules or components) that interact with each other by using the uses cases and scenarios as reference. The interaction is recorded in the relationship matrix, using the corresponding style. In the case of a first time link of these elements, a basic style will be constituted; otherwise, a heterogeneous style will be made, which implies a style reconstruction. Once that style is typified, the architectural representation is carried out.

The activities described above are repeated until there are no elements left to relate. These activities are illustrated by means of a case study in the following section.

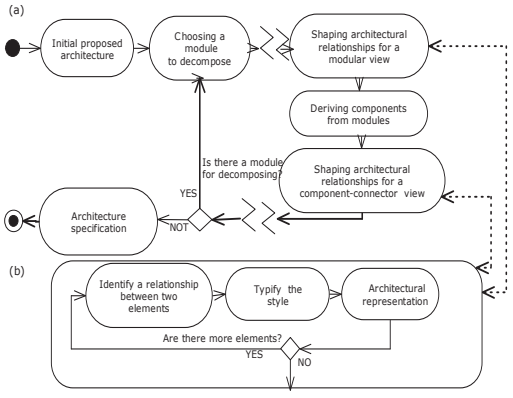


Figure. 11. (a) The architectural design method activities, (partially), (b) The activities to shape architectural relationships

6.2 Case Study

The case study is an extension of the case presented in [4]. This study consists of designing an architecture for a purchase and reservation ticket system for any kind of passenger transport (bus, air plane, train, etc), where the passengers can buy and reserve a ticket, or just search the Internet. After introducing the query, the system will generate a tentative itinerary as a first step.

With the intention to focus on the use of architecture styles proposed here, let's suppose the detection of elements (modules) in a first architecture derived from the use cases, which are shown in Table 1 ; and their scenarios are shown in the sequence diagram in Figure 12.

Table 1 Modules detected

Description	Main Modules
Query formation	QuerForm
Generating itinerary alternatives and routes	GeneItin
Making the reservation	Reservation
Purchasing a ticket	Purchase

Using the scenarios as the source, the type of modules to consider in the modular view and the relationships among them are detected. With this information the relationship matrices are shaped. Then the corresponding architectural representation for this view is built up, following the activities depicted in Figure 11(b). The matrix and representation for this view are shown in Figure 13.

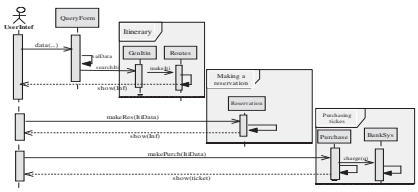


Figure 12. Sequence diagram for purchase and reservation of tickets

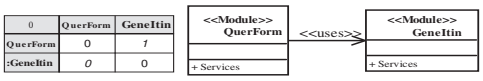


Figure 13. Matrix for the QueryForm and GeneItin modules and its architectural representation.

Once a partial creation of the modular architectonic view is done, their corresponding component-connector view must begin to be shaped. The modular architectonic view is used as a start point to begin the shaping of the component-connector view. Therefore, the first C-C view components becomes from the modular view modules. In the study case, the identification name of each module is then provided as a identification for each component of the C-C view-type.

Then, the information of the scenarios is again used to identify interactions among the components. The first identified interaction appears between the *QueryForm* and *GeneItin* components, where the former transmits the information of a query to the latter, which also generates alternative routes. In this case, the relationship matrix and its architectural representation are shown in Figure 14.

↓→	QuerForm	Geneltin
QuerForm	0	*1*
Geneltin	*0*	0

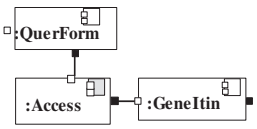


Figure 14. Matrix for the QuerForm and GeneItin components and its architectural representation (Pipe&Filter Style)

This process continues in an iterative way as depicted in Figure 10, until both views are accomplished. The terminal relationship matrices and their corresponding representation for each view are shown in Figure 15 and Figure 16.

(a)

0		Reservation	Make-Itin		Purchase
			QuerForm	Geneltin	
Reservation		0	1	1	*0*
Make-Itin	QuerForm	*0*	*0*	*1*	0
	Geneltin	*0*	0	0	0
Purchase		*1*	1	1	0

(b)

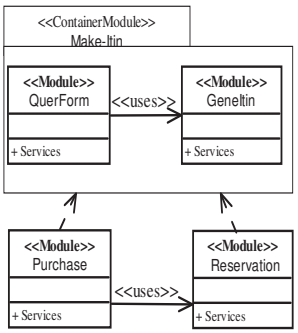


Figure 15. The modular view for the ticket purchase-reservation system, (a) its relationship matrix, and (b) its architectural representation,

(a)

Ticket Reservation-Purchase		Reservation	Make-Itin		Purchase
			QuerForm	GeneItin	
Reservation		0	2	0	*0*
Make-Itin	QuerForm	*0*	*0*	*1*	0
	GeneItin	*1*	0	0	1
Purchase		*0*	2	0	0

Ticket-Reservation-Purchase		Reservation	Makeltin	Purchase
Reservation		0	2	0
Makeltin	+	1	0	1
Purchase		0	2	0

(b)

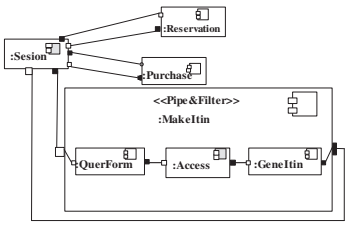


Fig 16 The component-connector view for the ticket purchase-reservation system, (a) its relationship matrix, and (b) its architectural representation

7 Conclusions

By modeling the architectural style as a type of relationship, we have proposed a novel way to design an architectural view. The identification and analysis of the basic relationships proposed here can be used in other view types.

The use of the relationship matrices allows us to: a) adapt some view types where there is dynamism among the element relationships; b) represent structural relationships as composition of architectural views; c) make a quantitative analysis using relationship matrices.

d) make the maintenance of the architecture easier since the element relationships are recorded (this is not addressed in this paper); e) generate a graphical architectural representation from the relationship matrix (in UML 2.0 notation).

Finally, it is shown that the application of the architectural styles improves the method of design of the software architectural views by making the analysis of relations in the architectural design process operational.

References

1. Bass L., Clements P., Kazman R., Software architecture in practice, Addison-Wesley, 2ª Ed. , ISBN 0-321-14595-9, 2003.
2. Hofmeister, C., Robert Nord, and Dilip Soni. Applied Software Architecture. Boston, MA: Addison Wesley, 2000.
3. Rozanski Nick, Woods Eoin, Software Systems Architecture, Working With Stakeholders Using Viewpoints and Perspectives, Addison-Wesley, ISBN 0-321-11229-6, 2005
4. Limon C. R., Ramos S.I., Torres J. J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development , Computational Science and Its Applications, LNCS 3983, pp, 726-735, 2006
5. IEEE Architecture Working Group, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.
6. Kruchten Philippe. The 4+1 View Model of Architecture, Paper published in IEEE Software Vol.12, No.6, pp. 42-50 Nov. 1995.
7. Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R. Stafford J. Documenting Software Architecture, Views and Beyond, Addison-Wesley, 2002.
8. Mehta N. R., Medvidovic N., Phadke S., Towards a Taxonomy of Software Connectors, Proceedings of the International Conference on Software Engineering- ICSE'00 (2000).
9. Baldwin, C.,Clark, K. Design Rules: Volume 1. The Power of Modularity. Cambridge, MA: The MIT Press. 2000.
10. Sangal N., Jordan, E. Sinha, V., Jackson D. Using Dependency Models to Manage Complex Software Architecture, Proceedings of the 20th annual ACM SIGPLAN Pages: 167 – 176, 2005.
11. R. C. Holt, Binary Relational Algebra Applied to Software Architectural, CSRI Technical Report 345, 1996
12. Dobrica L. and Niemelä E. , A survey on software architecture analysis methods, IEEE Transactions On Software Engineering, Vol. 28, No. 7, pp. 638-653, 2002

Verificación de Modelos Arquitectónicos Orientados a Aspectos

Jennifer Pérez
Universidad Politécnica de Madrid
28031 Madrid
jperez@eui.upm.es

Cristóbal Costa, Jose Ángel Carsí, Isidro Ramos
Dpto. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
46022 Valencia
{ccosta, pcarsi, iramos}@dsic.upv.es

Resumen

La mayoría de los errores cometidos en la producción de software provienen de las primeras fases de su ciclo de vida. Por este motivo, ha habido un incremento del número de enfoques que pretenden paliar los problemas que surgen en estas etapas, mediante una mejora de la asistencia al usuario durante el proceso de modelado. Un mecanismo fundamental que se debe proporcionar para guiar al usuario durante el modelado es la verificación de los modelos. La verificación de modelos permite detectar errores de modelado y no arrastrarlos al resto de etapas de desarrollo. En este trabajo se presenta cómo el enfoque PRISMA, siguiendo la propuesta de Desarrollo de Dirigido por Modelos (DDM), da soporte a la verificación de modelos arquitectónicos orientados a aspectos durante el proceso de modelado, y cómo este proceso de verificación se ha materializado en la herramienta PRISMA CASE.

1. Introducción

Hoy en día la producción de software fiable, de calidad, y fácilmente mantenible es todavía uno de los retos pendientes de la ingeniería del software. La mayoría de los errores cometidos en la producción de software provienen de las primeras fases de su ciclo de vida. Debido a que el coste de resolver estos problemas crece exponencialmente según avanza el proyecto, resulta de vital importancia centrarse en la mejora de las primeras etapas en lugar de postergarlo a etapas más tardías. Por ello, la mayor parte de los enfoques que pretenden paliar estos problemas, mejoran el desarrollo de software automatizando las tareas de las primeras etapas, basándose en el Desarrollo Dirigido por Modelos (DDM) [5][14].

La propuesta DDM es un paradigma de desarrollo software basado en modelos que forma parte del enfoque de *Ingeniería Dirigida por Modelos* (IDM) (más conocido por su término inglés *Model Driven Engineering*, MDE). El uso de modelos para el desarrollo de software proporciona grandes ventajas, como un alto nivel de abstracción e independencia de tecnología. El alto nivel de abstracción de los modelos permite trabajar con modelos, metamodelos y modelos específicos de dominio de igual forma. Mientras que la obtención de soluciones independientes de tecnología permite generar código para diferentes lenguajes y plataformas mediante técnicas de generación automática.

La generación automática de código evita los errores de correspondencia entre la semántica del modelo y el código de la aplicación, pero no previene los errores de modelado que comete el usuario. Con el objetivo de paliarlos, se hace necesario el proporcionar asistencia al usuario para guiarle durante el proceso de modelado. Este mecanismo de guiado debe estar bien soportado tanto por el enfoque como por la herramienta que materialice dicho enfoque.

La **verificación y validación** del software no sólo es posible realizarlas en la etapa de pruebas del ciclo de vida clásico, mediante pruebas de caja negra y caja blanca para el código de la aplicación. La verificación y validación también pueden realizarse en la etapa de modelado para detectar errores de modelado y de comportamiento del modelo. Por lo tanto, la **verificación y validación de modelos** debe formar parte del proceso de desarrollo de software e integrarse dentro de aquellos procesos que pretendan mejorar el desarrollo en las primeras etapas.

La **verificación de modelos** permite saber si un modelo satisface o no las restricciones definidas por su metamodelo. Si un modelo satisface todas y cada una de las restricciones que se establecen en su metamodelo, se puede afirmar que el modelo es correcto. Mientras que la **validación de modelos** permite conocer si el

Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos, JISBD'2007, pp.167-176 ISBN: 978-84-9732-595-0 **© 2007 Los autores, Thomson**

Pérez, J., Costa, C., Carsí, J. A.,Ramos, I. Verificación de Modelos Arquitectónicos Orientados a Aspectos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos. 2007.

comportamiento de un modelo es el esperado por el usuario y si un modelo cumple determinadas propiedades de calidad. La validación de propiedades y modelos se realiza mediante técnicas de prototipación y *model checking*.

PRISMA es un enfoque de desarrollo, basado en el paradigma DDM, que tiene como objetivo principal el dar soporte al desarrollo de arquitecturas software orientadas a aspectos e independientes de tecnología [18]. Su enfoque consiste en modelar y especificar completamente las arquitecturas, para que puedan ser compiladas a posteriori a diferentes plataformas tecnológicas y lenguajes de programación. Para dar soporte a este enfoque se ha desarrollado un prototipo de herramienta CASE (Computer-Aided Software Engineering) llamada PRISMA CASE [22].

En la actualidad, PRISMA CASE es capaz de generar código C# orientado a aspectos a partir de un modelo arquitectónico. Dicho código es ejecutable sobre la plataforma .NET gracias al middleware PRISMANET [21]. PRISMANET es el modelo específico de plataforma del modelo PRISMA (Platform Specific Model, PSM [14]).

Sin embargo, aunque el objetivo del proceso de verificación de modelos PRISMA es la generación código a partir de modelos correctos, este trabajo se va a centrar en el proceso de verificación. Por lo tanto, el objetivo de este trabajo es presentar cómo PRISMA da soporte, durante el proceso de modelado, a la verificación de modelos arquitectónicos orientados a aspectos. De esta manera, se previenen errores de modelado que se convertirían en errores mucho más graves en la siguiente etapa de desarrollo, que en PRISMA es la generación automática del código. En este trabajo se describe cómo PRISMA, a través de su metamodelo, define las propiedades y restricciones que debe tener y satisfacer todo modelo arquitectónico PRISMA, y cómo esto se materializa en un proceso de verificación de la herramienta PRISMA CASE que asiste al usuario en todo momento.

La estructura del artículo es la siguiente: el apartado 2 presenta cómo el modelo PRISMA define las restricciones que sus modelos arquitectónicos orientados a aspectos deben satisfacer; así como los tipos de restricciones y de verificación existentes. En el apartado 3 se presenta cómo la herramienta PRISMA CASE da soporte al modelado gráfico de modelos arquitectónicos PRISMA guiando al arquitecto en todo momento mediante el soporte que ofrece para la verificación de modelos. En el apartado 4 se detallan los trabajos relacionados y, finalmente, en el apartado 5 se presentan las conclusiones y los trabajos futuros.

2. Verificación en PRISMA

El enfoque PRISMA está compuesto por un modelo, un Lenguaje de Descripción de Arquitecturas Orientado a Aspectos (LDAOA) [19] formal basado en la lógica modal de acciones [24] y en el π-cálculo poliádico [11], una herramienta CASE y una metodología de desarrollo. El modelo PRISMA [20] permite la especificación de arquitecturas software orientadas a aspectos. Dicho modelo se caracteriza por ser un modelo arquitectónico con conectores [23] y un modelo orientado a aspectos simétrico [6]. Por lo tanto, los conectores son ciudadanos de primer orden del LDAOA de PRISMA, y tanto la funcionalidad como otros requisitos no funcionales (distribución, seguridad, persistencia, etc.) son considerados un asunto común de la arquitectura (*crosscutting-concern*) [8], [9], [17]. Tal y como establece el DSOA, en PRISMA los *crosscutting-concerns* (distribución, seguridad, funcionalidad, coordinación, persistencia, etc.) se modelan mediante aspectos (uno o más*)*, tratando así de forma homogénea el modelado de los requisitos funcionales y no funcionales. Un aspecto define el estado y el comportamiento de un *crosscutting-concern* específico del sistema software.

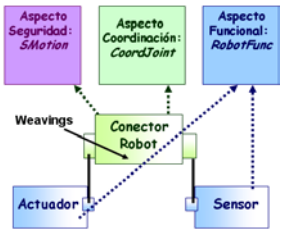


Figura 1. Ejemplo de modelo PRISMA

Un mismo aspecto es importado por todos aquellos elementos arquitectónicos (componentes y conectores) que necesitan el comportamiento del *crosscutting-concern* que define el aspecto (ver Figura 1). Por lo tanto, un elemento arquitectónico se define mediante un conjunto de aspectos que lo describen desde diferentes asuntos de interés (*crosscutting-concerns*) de la arquitectura. Un ejemplo es el conector que coordina al actuador y

al sensor de un robot teleoperado, que necesita tener en cuenta la seguridad y la coordinación para sincronizarlos. El conector, antes de enviar al actuador el movimiento que tiene que ejecutar el robot, comprueba que dicho movimiento es seguro para el robot y su operario. Posteriormente, el sensor le notifica el resultado del movimiento.

2.1. Jerarquía de modelos PRISMA

El proceso de desarrollo software dirigido por modelos que proporciona PRISMA está basado en la jerarquía de niveles MOF (*Meta-Object Facility*) [13]. En el nivel más alto de dicha jerarquía se encuentra el meta-metamodelo MOF, el cual define un lenguaje abstracto para definir metamodelos (ver nivel M3 de la Figura 2). Los metamodelos se describen en términos del lenguaje MOF y se sitúan en el nivel M2. Por ello, en el nivel M2 se encuentra el metamodelo PRISMA, donde se definen las propiedades de los modelos PRISMA (ver nivel M2, Figura 2). En el nivel M1 se encuentran cada uno de los modelos arquitectónicos PRISMA descritos en base al metamodelo PRISMA. Éstos definen tipos (aspectos, elementos arquitectónicos, etc) que modelan sistemas software, como pueden ser los sistemas bancarios, los sistemas de tele-operación, etc. (ver nivel M1, Figura 2). Por último, en el nivel M0 se encuentran las configuraciones arquitectónicas de los modelos definidos en M1, es decir, las instancias de los tipos definidos en M1 y sus interconexiones (ver nivel M0 Figura 2).

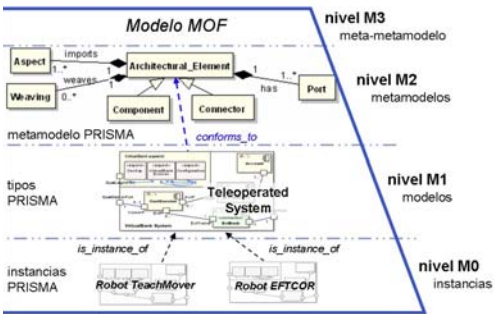


Figura 2. Niveles MOF en PRISMA

Esta jerarquía implica que existen dos tipos de verificación de modelos: verificación de modelos arquitectónicos y verificación de configuraciones arquitectónicas. La **verificación de modelos arquitectónicos** consiste en comprobar que los modelos arquitectónicos de tipos PRISMA satisfacen las propiedades y restricciones definidas en su metamodelo. Mientras que la **verificación de configuraciones arquitectónicas** consiste en comprobar que una configuración de instancias satisface el modelo del cual se han instanciado, que se cumplan los patrones de interconexión y composición de instancias. Este artículo se centra en explicar la verificación de modelos arquitectónicos.

2.2. El metamodelo PRISMA

El metamodelo de PRISMA [20] define y establece las propiedades de los modelos PRISMA de forma precisa, mediante metaclases, relaciones entre metaclases y restricciones. Las metaclases describen las propiedades y servicios de cada concepto del modelo. Además, las metaclases junto con sus relaciones, describen la estructura y la información necesaria para describir un modelo arquitectónico PRISMA. El metamodelo también describe aquellas restricciones del modelo PRISMA que no pueden describirse mediante la estructura o la información del metamodelo. Dichas restricciones están asociadas a una determinada metaclase del modelo, concretamente a la que se ve afectada por la restricción.

El proceso de verificación de modelos arquitectónicos PRISMA consiste concretamente en comprobar que los modelos: (1) contienen tipos con toda la información que establecen sus metaclases; (2) establecen las relaciones que unen a los tipos adecuadamente, así como la cantidad necesaria de relaciones entre ellos; y (3) satisfacen las restricciones del metamodelo que afectan a las metaclases. Esta verificación se debe aplicar durante el proceso de modelado de modelos arquitectónicos PRISMA y debe asistir en todo momento al arquitecto software.

El metamodelo de PRISMA se ha especificado usando el diagrama de clases de UML (*Unified Modelling Language*) 1.5 y el lenguaje OCL (*Object Constraint Language*) 2.0 [16]. UML se ha utilizado para definir las metaclases y sus relaciones; mientras que OCL se ha utilizado para especificar las restricciones del metamodelo. Se han elegido estos dos lenguajes debido a que son estándares de uso extendido, lo cual facilita la comprensión del metamodelo PRISMA, y como consecuencia del modelo PRISMA.

El metamodelo PRISMA es muy extenso y se encuentra estructurado en paquetes funcionales.

En este artículo nos vamos a centrar en tres de ellos para ilustrar la verificación de modelos arquitectónicos PRISMA.

- Aspectos

Un aspecto define el estado y el comportamiento de un *crosscutting-concern* específico del sistema software. El paquete *Aspects* del metamodelo PRISMA incluye todas las metaclases que son necesarias para especificar un aspecto. El estado y el comportamiento de los aspectos se definen mediante atributos, servicios, precondiciones, valuaciones, restricciones, *played_roles* y protocolos (ver Figura 3).

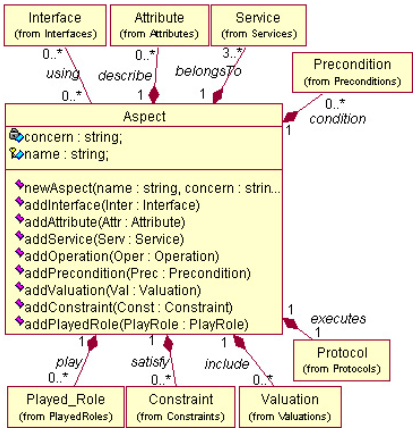


Figura 3. La metaclase *Aspect* del paquete *Aspects* del metamodelo PRISMA

La metaclase *Aspect* tiene dos atributos, *name* y *concern* (ver Figura 3). Estos atributos almacenan el nombre del aspecto y el *concern* que define el aspecto, respectivamente. En el ejemplo de la Figura 1, uno de los aspectos tiene como nombre *SMotion* y como concern *Seguridad.* Los aspectos pueden necesitar almacenar información para realizar correctamente su comportamiento. Por este motivo, la metaclase *Aspect* tiene una relación de agregación con la metaclase *Attribute*. Dicha relación define que un aspecto tiene un conjunto de atributos y además establece que un atributo sólo puede pertenecer a un único aspecto (ver la agregación *describe* de la Figura 3). Por lo tanto, un atributo no tiene entidad propia y sólo puede definirse dentro de un aspecto. Esta relación establece una restricción que el usuario en sus modelos debe preservar. Igualmente, cada una de las relaciones que la metaclase *Aspect* tiene con el resto de metaclases con las que se relaciona, establecen una serie de restricciones de cardinalidad que deben cumplirse durante el proceso de modelado (ver las relaciones *using, condition,* etc. en la Figura 3). Estas relaciones no se explican en detalle en este artículo por razones de espacio, pero se detallan en [18].

- Elementos arquitectónicos

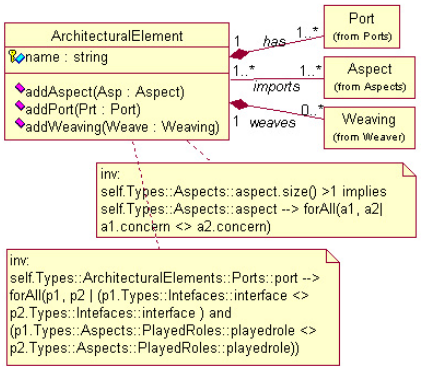


Figura 4. El paquete ArchitecturalElements del metamodelo PRISMA

En PRISMA existen dos tipos de elementos arquitectónicos: componentes y conectores. Un elemento arquitectónico se define mediante: (1) un conjunto de aspectos que lo describen desde diferentes *concerns* (por ejemplo: el *Conector* del robot importa un aspecto de coordinación y otro de seguridad); (2) las relaciones de sincronización entre aspectos (*weavings*) (el *Conector* del robot tiene un weaving que establece que, antes de solicitar al *Actuador* que ejecute un movimiento y que el *Sensor* lo lea, debe comprobarse que el movimiento es seguro a través del aspecto de seguridad); y (3) los puertos que publican su comportamiento (el *Conector* publica los servicios de envío de movimientos y de recepción de sus resultados). Por esta razón, la metaclase *ArchitecturalElement* tiene dos relaciones de agregación con las metaclases *Port* y *Weaving,* y una asociación con la metaclase *Aspect* (ver Figura 4). Estas relaciones establecen una serie de restricciones: (1) un elemento arquitectónico al menos tiene un puerto y un puerto pertenece a un único elemento arquitectónico, (2) un elemento arquitectónico importa al menos un aspecto y un aspecto puede ser importado por uno o más elementos arquitectónicos de la arquitectura, y (3) un elemento arquitectónico puede tener *weavings* asociados o no. Además, la metaclase *ArchitecturalElement* tiene dos restricciones que han sido definidas con OCL (ver Figura 4):

<<Un elemento arquitectónico no puede importar más de un aspecto del mismo concern >>
<<No existen dos puertos de un elemento arquitectónico que tengan la misma interfaz y played_role asociados, es decir, que publiquen el mismo comportamiento>>

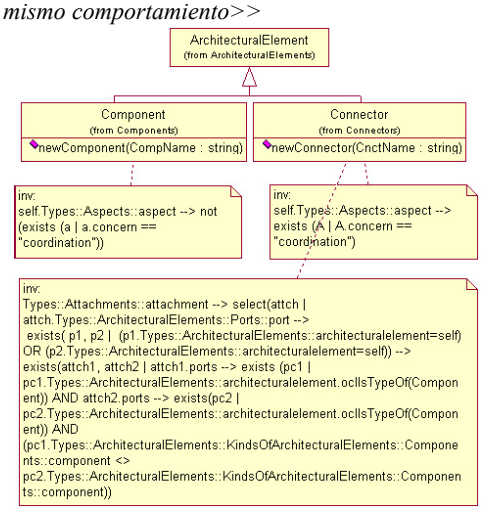


Figura 5. El paquete *KindsOfArchitecturalElements* del metamodelo PRISMA

Como los componentes y los conectores son elementos arquitectónicos, sus metaclases heredan las propiedades de la metaclase *ArchitecturalElement* (ver Figura 5). Los componentes y conectores se comunican mediante unos canales conectados a sus puertos llamados *attachments*. Tanto la metaclase *Component,* como la metaclase *Connector,* tienen asociadas restricciones OCL para definir las propiedades que caracterizan a ambos conceptos. Como un componente no actúa como coordinador entre componentes, la restricción OCL asociada a la metaclase *Component* establece que un componente no puede importar un aspecto de tipo coordinación. De la misma manera, como un conector actúa como coordinador, éste debe importar un aspecto de tipo coordinación. Por ello, en el ejemplo, el *Conector* del robot tiene asociado un aspecto de coordinación (ver Figura 1). Además, la metaclase *Connector* tiene otra restricción que establece lo siguiente:
<<Un conector debe tener al menos dos conexiones (attachments) asociadas y cada una de ellas debe conectar al conector con dos componentes diferentes >>

De ahí que el *Sensor* y el *Actuador* se comuniquen a través del *Conector* mediante dos attachments que unen sus puertos (ver Figura 1)

2.3. Tipos de verificación

A partir de las restricciones del metamodelo PRISMA presentadas en la sección anterior, es posible distinguir dos tipos de restricciones:

- Restricciones fuertes

Las restricciones fuertes son aquellas que siempre deben satisfacerse, independientemente del momento en el que se encuentre el proceso de modelado. Un ejemplo de restricción fuerte asociada al concepto de aspecto es su relación con los atributos. Un atributo nunca se podrá asociar a otro concepto que no sea un aspecto. Ésta es una restricción fuerte porque si se añadiera un atributo a otro concepto del modelo arquitectónico violaría el modelo PRISMA. Otro ejemplo de restricción fuerte es el hecho de que un componente nunca puede importar un aspecto de coordinación. Si el arquitecto le asocia un aspecto de coordinación a un componente, el modelo resultante no satisfaría el modelo PRISMA.

- Restricciones débiles

Las restricciones débiles son aquellas que pueden no satisfacerse durante el proceso de modelado, pero que una vez el modelo arquitectónico esté acabado, todas y cada una de ellas deben satisfacerse.

Los elementos arquitectónicos tienen muchas restricciones débiles asociadas. Por ejemplo: un componente debe importar al menos un aspecto y debe tener al menos un puerto. Éste es un ejemplo de restricción débil porque es posible definir un elemento arquitectónico sin establecer sus puertos y/o aspectos, para definirlos más tarde.

Las restricciones débiles son necesarias en todo proceso de modelado, ya que si se establecen las dependencias entre dos conceptos con restricciones fuertes ninguno de los dos conceptos podría modelarse. Por ejemplo, si se establece que todo puerto debe estar asociado a un elemento arquitectónico y que todo elemento arquitectónico debe tener asociado un puerto, no se podría crear ninguno de los dos, ya que la creación del elemento arquitectónico requeriría la existencia previa del puerto y viceversa.

Las restricciones débiles dotan de una mayor flexibilidad al proceso de modelado. El hecho de que existan restricciones débiles que no se satisfagan significa que el modelo arquitectónico

no se ha acabado. Sin embargo, pueden existir partes del modelo arquitectónico que sí estén finalizadas y que el arquitecto quiera verificar. De esta necesidad emergen dos tipos de verificación: verificación parcial y verificación total.

La **verificación parcial** consiste en aplicar únicamente aquellas restricciones que afecten a los elementos, conceptos o partes del modelo que hayan sido seleccionadas por el arquitecto para su verificación. Este tipo de verificación permite al arquitecto verificar incrementalmente el modelo; así como verificar elementos del modelo para su almacenamiento en repositorios y posterior reutilización en otros modelos. Por ejemplo, en el caso de PRISMA, la verificación de un componente específico consistiría en verificar que tiene todas las relaciones necesarias que establecen las metaclases *Architectural Element* y *Component,* y que satisface todas las reglas OCL que ambas tienen asociadas.

La **verificación total** es aquella que se aplica al modelo arquitectónico completo. Por lo tanto, la verificación total consiste en verificar todas y cada una de las restricciones que debe cumplir un modelo. En el caso de PRISMA, este proceso implica que se comprueben todas las restricciones del metamodelo PRISMA.

3. PRISMA CASE

El enfoque de desarrollo PRISMA es soportado por PRISMA CASE, un marco de desarrollo de arquitecturas software orientadas a aspectos que sigue la propuesta DDM. Dicha herramienta CASE facilita y guía al usuario durante el proceso de desarrollo mediante: (1) el modelado gráfico en lugar de utilizar un lenguaje formal, (2) el soporte a la verificación parcial y total de sus modelos arquitectónicos, (3) la generación automática de código C# y (4) la validación del comportamiento de la arquitectura mediante su ejecución sobre PRISMANET. Este artículo se va a centrar en mostrar en detalle cómo el enfoque PRISMA, a través de su herramienta PRISMA CASE, da soporte a las restricciones débiles y fuertes del metamodelo PRISMA, y a la verificación total y parcial de modelos arquitectónicos PRISMA. De esta manera, PRISMA CASE guía al usuario durante todo el proceso de modelado siguiendo la propuesta DDM [14][5]. El entorno de desarrollo que se ha utilizado para desarrollar PRISMA CASE ha sido DSL Tools (Domain Specific Language Tools) [2]. Se ha elegido este entorno porque permite crear, modificar, visualizar y usar lenguajes específicos de dominio para automatizar y mejorar los procesos de desarrollo de software.

3.1. Restricciones fuertes

Para que las restricciones fuertes sean verificadas durante el proceso de modelado, éstas deben estar definidas dentro de PRISMA CASE. Durante el desarrollo de PRISMA CASE se han realizado las siguientes tareas: (1) especificar el metamodelo PRISMA en DSL Tools y (2) desarrollar las restricciones de comportamiento gráfico para cada clase del metamodelo PRISMA.

- Metamodelo PRISMA en DSL Tools

El metamodelo de PRISMA ha sido introducido en DSL Tools describiendo cada uno de los conceptos del metamodelo PRISMA con las primitivas del lenguaje de meta-metamodelado de DSL Tools.

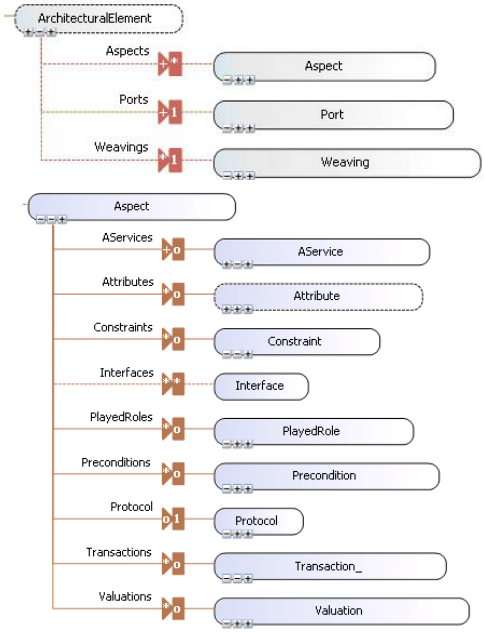


Figura 6. Definición de las metaclases *ArchitecturalElement* y *Aspect* del metamodelo PRISMA en DSL Tools

Las primitivas de clase y relaciones (herencia, embebidas y referenciales) de DSL Tools se han utilizado para definir las metaclases PRISMA y las relaciones entre ellas, respectivamente. La Figura 6 muestra la definición de las metaclases

ArchitecturalElement y *Aspect* en DSL Tools utilizando estas primitivas.

La representación gráfica de cada concepto PRISMA ha sido incluida en la herramienta gracias a que DSL Tools permite asociar una metáfora gráfica a cada uno de los conceptos del metamodelo. A partir del metamodelo definido y de las metáforas gráficas asociadas, DSL Tools genera automáticamente la herramienta de modelado. Esta herramienta generada ofrece como primitivas de modelado los conceptos del metamodelo con su correspondiente representación gráfica (ver *toolbox*, a la izquierda de la Figura 7). Como esta herramienta es generada a partir del metamodelo y de las restricciones desarrolladas en DSL, las relaciones de dependencia y asociación que se han establecido entre conceptos son satisfechas de forma inherente en el entorno de desarrollo resultante.

De esta forma, la metáfora gráfica asociada a cada metaclase del metamodelo PRISMA ya incorpora las relaciones de inclusión, definiendo elementos gráficos que agrupan un conjunto de conceptos. Por ejemplo, la metaclase aspecto tiene una metáfora gráfica que es un contenedor de otros conceptos, con los que mantiene una relación de inclusión. Véase en la Figura 7 cómo el aspecto de coordinación *CProcessSUC* del ejemplo incluye la definición de su atributo *TempHalfSteps*, donde guarda la posición de la articulación del robot. Por lo tanto, la propia representación del aspecto satisface la restricción en la que se establece que: <<*un atributo no tiene entidad propia y sólo puede definirse dentro de un aspecto*>> (ver sección 2.2, Aspectos). Lo mismo ocurre con el resto de conceptos que incluye la metaclase aspecto del metamodelo PRISMA (servicios, valuaciones, precondiciones, etc). Así mismo, como las relaciones referenciales entre conceptos también están definidas en el metamodelo, éstas son satisfechas de forma inherente. Por lo tanto, la herramienta de modelado guía al arquitecto, pudiendo utilizar únicamente las relaciones para enlazar los elementos permitidos. Por ejemplo, la relación entre un elemento arquitectónico y sus aspectos se plasma mediante la primitiva de modelado *ArchitecturalElementHasAspect* (cuya metáfora gráfica es una flecha) que permite enlazar un elemento arquitectónico y un aspecto. Si el arquitecto intentara usar esta primitiva para enlazar un elemento arquitectónico, por ejemplo con una interfaz, se visualizaría una señal de prohibición para denotar que es imposible enlazar esos dos conceptos con esa primitiva.

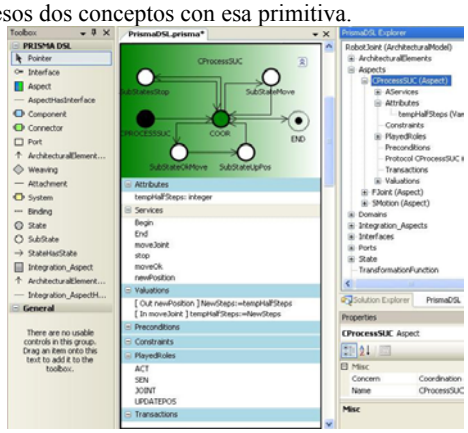


Figura 7. Representación gráfica de un aspecto en PRISMA CASE

- ***Restricciones de comportamiento gráfico***

Tal y como se ha presentado en la sección 2.2, existen restricciones que no es posible especificarlas mediante relaciones del metamodelo, por lo que deben especificarse mediante reglas OCL asociadas a las metaclases.

Con el objetivo de dar soporte en PRISMA CASE a este tipo de restricciones durante el proceso de modelado, se ha hecho uso del mecanismo de clases parciales que ofrece DSL Tools. DSL Tools permite asociar a cada uno de los conceptos del metamodelo clases C# parciales para consultar y actualizar sus propiedades, navegar entre clases, etc. En el metamodelo PRISMA se ha hecho uso de éstas para implementar las reglas OCL de cada metaclase y relación del metamodelo PRISMA.

Estas clases pueden asociarse a la metáfora gráfica del concepto o al concepto del modelo en sí. Las restricciones fuertes, como no se pueden violar en todo el proceso de modelado, son clases C# parciales asociadas a la metáfora gráfica para ni siquiera dar la oportunidad de que no se cumplan. La Figura 8 muestra la clase parcial asociada a la relación entre las metaclases *ArchitecturalElement* y *Aspect*, que se ha extendido para soportar aquellas restricciones fuertes asociadas a la importación de aspectos por parte de un elemento arquitectónico. Concretamente, esta clase parcial comprueba las

reglas OCL siguientes: *<<Un elemento arquitectónico no puede importar más de un aspecto del mismo concern>>* y *<<Un componente no puede importar un aspecto de coordinación>>*. Dichas reglas se comprueban mediante el método *CanCreateConnection* que incluye la clase parcial *ArchitecturalElementHasAspect*.

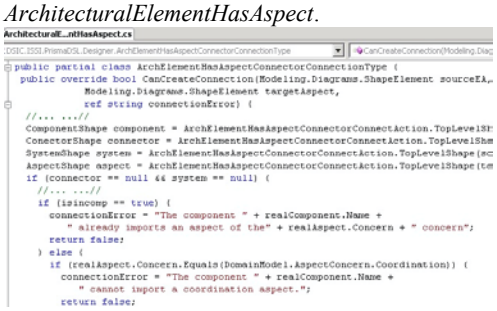


Figura 8. Clase C# parcial de la relación *ArchitecturalElementHasAspect*

De esta forma, cada vez que el arquitecto importe un aspecto desde un elemento arquitectónico, se ejecuta el método *CanCreateConnection*, que se encarga de comprobar ambas restricciones. El código fuente muestra que, en caso de no satisfacerse las condiciones de las restricciones, se lanza un error de conexión gráfica con su mensaje de texto asociado, impidiendo que el arquitecto pueda establecer la importación y mostrando una señal de prohibición. A partir de este ejemplo se puede concluir que las restricciones fuertes, al formar parte de la metáfora gráfica, se comprueban siempre sin solicitud explícita por parte del arquitecto.

3.2. Restricciones débiles

La definición de restricciones débiles en PRISMA CASE también se ha realizado mediante el desarrollo de clases C# parciales. Estas clases parciales están asociadas a un concepto del modelo en sí, en lugar de estar asociadas a la metáfora gráfica del concepto como las restricciones fuertes. Por este motivo, las restricciones débiles únicamente se comprueban si el arquitecto lo solicita o una vez se ha finalizado el modelo arquitectónico y se procede a la generación de su código C#. En la Figura 9 se muestra la clase parcial asociada a la metaclase *Component* en la que mediante el método *ValidateComponentHasAspect* se comprueba que un componente al menos importa un aspecto. Como puede observarse en el código de dicho método, a diferencia de las clases parciales asociadas al comportamiento gráfico, si la condición no se cumple, el error es registrado para mostrarse posteriormente al arquitecto, junto a todos los errores identificados en el proceso de verificación.

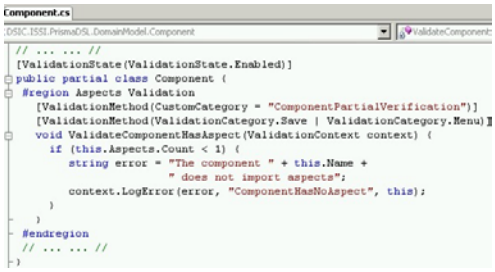


Figura 9. Clase C# parcial de *Component*

En PRISMA CASE se ha dado soporte no sólo a la verificación total, también a la verificación parcial. La verificación parcial se ofrece agrupada por tipos PRISMA: interfaces, aspectos, componentes, conectores, sistemas (componentes complejos) y *attachments*. Por ejemplo, si el arquitecto solicita la verificación de los aspectos, únicamente se comprobarán las restricciones débiles asociadas a la metaclase *Aspect,* es decir, las reglas de su clase parcial. De esta manera, el arquitecto puede verificar su modelo de forma incremental y localizar qué tipo está generando errores de verificación.

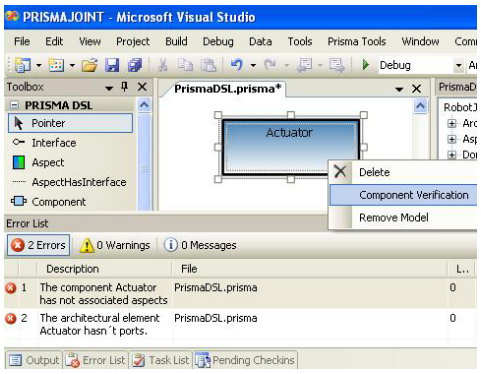


Figura 10. Verificación parcial: componente *Actuator*

Tras realizar la verificación, el arquitecto es informado de todos los errores detectados para que pueda ir subsanándolos uno a uno (ver ventana *Error List*,Figura 10).

Además, PRISMA CASE también ofrece la posibilidad de verificar un elemento determinado

del modelo arquitectónico. Por ejemplo, en la Figura 10 puede observarse la lista de errores generados a consecuencia de la verificación parcial de un solo elemento, el componente *Actuador* del ejemplo. Dicha verificación comprueba que el *Actuador* al menos tiene un aspecto y un puerto (ver sección 2.2).

4. Trabajos relacionados

En el campo del DDM se pueden identificar tres tipos de estrategias para realizar la verificación de modelos [12]: análisis, construcción y monitorización. Las estrategias de análisis definen las restricciones junto al metamodelo, mediante reglas OCL. Una herramienta de verificación es la encargada de comprobar, a posteriori, que los modelos construidos las cumplen. Estas estrategias realizan una verificación total del modelo. Existen numerosas herramientas que ofrecen verificadores OCL [25].

Las estrategias constructivas construyen los modelos aplicando reglas de transformación QVT [15] o ATL [1]. Las estrategias de monitorización definen los modelos mediante herramientas gráficas que impiden la creación de modelos inconsistentes. Ambas estrategias tienen en común que los modelos se construyen incrementalmente definiendo las restricciones de forma implícita. Sin embargo, esto tiene como inconveniente que las restricciones se encuentran dispersas entre las diversas reglas de transformación o la lógica de monitorización, dificultando su mantenimiento. Existen enfoques mixtos, como el de Bézivin y Jouault [7], en el que se utiliza el motor de las reglas de transformación ATL para verificar que se satisfacen las reglas OCL del metamodelo. Sin embargo, el principal inconveniente de estos enfoques es que las reglas OCL se proporcionan de forma separada al metamodelo.

Las propuestas que mayor difusión han alcanzado han sido aquellas que combinan las estrategias de análisis y de monitorización, beneficiándose de la flexibilidad de la primera y de la seguridad ofrecida por la segunda. Por ejemplo: Microsoft DSL-Tools [2], AMMA [10], XMF-Mosaic [25], EMF [3] y GME [4]. Estas herramientas ofrecen soporte a todas las etapas del proceso de DDM, aunque cada una de ellas lo hace a través de un meta-metamodelo diferente, unas estrategias de verificación de la consistencia diferentes y un proceso de generación de código distinto. La elección de una herramienta u otra depende de distintos factores. La herramienta DSL-Tools fue elegida para el desarrollo de PRISMA CASE por ofrecer mecanismos de monitorización y análisis para la verificación de la consistencia. La monitorización es ofrecida de forma implícita por su modelador gráfico previa introducción del metamodelo en la herramienta. El análisis de las restricciones es ofrecido mediante la personalización del código asociado a los conceptos del metamodelo. Además, la herramienta de DSL-Tools facilita la interoperabilidad con PRISMANET [21].

Finalmente cabe destacar que, a pesar de que en el campo de las arquitecturas software se está haciendo un gran esfuerzo para dar soporte a la validación de modelos, no existen propuestas sólidas para su verificación. Por lo tanto, PRISMA cubre esta necesidad existente en el campo.

5. Conclusiones y trabajos futuros

En este trabajo se ha presentado la necesidad de dar soporte a la verificación de modelos durante el DDM. Para ello, en primer lugar se han clasificado las restricciones a verificar en dos tipos: fuertes y débiles. A partir de esta clasificación, se ha definido la posibilidad de realizar una verificación no sólo total, sino también parcial e incremental, aumentando la flexibilidad del proceso de modelado y adaptando la asistencia a las necesidades del arquitecto en cada momento. Además, este trabajo presenta una metodología para dar soporte a la verificación de modelos durante el proceso DDM, así como los mecanismos necesarios para darle soporte.

Esta metodología ha sido aplicada a PRISMA, un enfoque que permite el modelado de arquitecturas software orientadas a aspectos siguiendo el DDM. El enfoque PRISMA define, a través de su metamodelo, las propiedades y restricciones que debe satisfacer todo modelo arquitectónico PRISMA. Dichas restricciones han sido analizadas para clasificarlas en fuertes y débiles, estableciendo como fuertes sólo las estrictamente necesarias. De esta manera, PRISMA proporciona una mayor flexibilidad al arquitecto durante el proceso de modelado. El metamodelo PRISMA se ha incluido como parte de la herramienta PRISMA CASE, en la que se ha dado soporte a: (1) el proceso de verificación total y parcial, (2) la diferenciación de restricciones

fuertes y débiles, y (3) la verificación incremental de los modelos, mediante su verificación parcial por tipos (aspectos, componentes, etc) o por elementos concretos. De esta forma, PRISMA se presenta como un enfoque que facilita el proceso de modelado de arquitecturas software orientadas a aspectos siguiendo un proceso de verificación bien definido y soportado.

Existen varios trabajos futuros como continuación de este trabajo. Un trabajo a corto plazo es la extensión de los niveles de modelado de PRISMA. Actualmente, PRISMA está definido siguiendo la jerarquía MOF. Sin embargo es necesario extenderla para así dar soporte a líneas de producto y a refinamientos en los procesos de modelado de modelos específicos de dominio. Asimismo, este trabajo futuro conlleva la extensión de la metodología de verificación propuesta y su implementación. Además, la propuesta PRISMA debe integrar mecanismos de *model checking* para poder validar el comportamiento y los atributos de calidad que requieran los modelos. Finalmente, la extensión del compilador de modelos PRISMA a otros lenguajes como Java o C++ sería un trabajo sencillo mediante la definición de las correspondientes plantillas de generación de código en DSL Tools. Así mismo, otro trabajo futuro asociado a éste consistiría en la definición de otros PSM para PRISMA.

Agradecimientos

Este trabajo ha sido cofinanciado por el Ministerio de Educación y Ciencia, en el marco del Programa Nacional de Investigación, Desarrollo e Innovación, proyecto META (TIN2006-15175-C05-01), y por el Fondo Europeo de Desarrollo Regional (FEDER).

Referencias

[1] Bézivin, J., Jouault, F. Using ATL for Checking Models. International Workshop on Graph and Model Transformation (GraMoT), ENTCS 152, 2006.

[2] Domain-Specific Language (DSL) Tools. http://msdn.microsoft.com/vstudio/DSLTools/

[3] Eclipse Modeling Framework (EMF) http://www.eclipse.org/modeling/emf

[4] GME, The Generic Modeling Environment, http://www.isis.vanderbilt.edu/Projects/gme/

[5] Greenfield, J., et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley, 2004.

[6] Harrison, W. H., et al. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. TR RC22685 (W0212-147), Thomas J. Watson Research Center, IBM, 2002.

[7] Jouault, F., Kurtev, I. Transforming Models with ATL. Workshop on Model Transformations in Practice, collocated with MoDELS, Jamaica 2005.

[8] Kiczales, G, et al. An overview of AspectJ. 15th Eur. Conf. on Object-Oriented Programming (ECOOP). LNCS 2072, London, UK, 2001.

[9] Kiczales, G., et al. Aspect-Oriented Programming. 11th Eur. Conf. on Object-Oriented Programming (ECOOP). LNCS 1241, Jyväskylä, Finland, 1997.

[10] Kurtev, I., et al. Model-based DSL frameworks. 21st Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Portland, Oregon, USA, 2006.

[11] Milner R., *The Polyadic π-Calculus: A Tutorial*. Laboratory for Foundations of Computer Science Department, University of Edinburgh, October 1993

[12] Modelware project. D1.5 Model Composition: Development of consistency rules. TR, WP1 Modelling technologies, 2006 http://www.modelware-ist.org

[13] Object Management Group. Meta-Object Facility 1.4 Spec. Technical Report formal/2002-04-03, 2002

[14] Object Management Group. Model Driven Architecture Guide v1.0.1, 2003. http://www.omg.org/

[15] Object Management Group. MOF QVT final adopted specification. Tech. Report formal/2005-11-01, 2005

[16] Object Management Group. The Unified Modeling Language. http://www.uml.org/

[17] Ossher H., Tarr P., *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* Symp. on Software Architectures and Component Technology: The state of the Art in Software Development, Kluwer, 2001.

[18] Pérez, J. PRISMA: Aspect-Oriented Software Architectures. Tesis Doctoral, Universidad Politécnica de Valencia, 2006.

[19] Pérez, J., et al., Designing Software Architectures with an Aspect-Oriented Architecture Description Language. 9th Int. Symp. on Component-Based Software Engineering (CBSE), LNCS vol. 4063, Springer, 2006.

[20] Pérez, J., et al. Dynamic Evolution in Aspect-Oriented Architectural Models. 2nd Eur. Wks. on Software Architecture (EWSA), LNCS 3527 Pisa, Italy, 2005.

[21] Pérez, J., et al. Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. 3rd Int. Conf. on .NET Technologies. Pilsen, Czech Republic 2005.

[22] PRISMA web page, http://prisma.dsic.upv.es

[23] Shaw M. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. Wks. on Studies of Software Design, January, 1994.

[24] Stirling C., *Modal and Temporal Logics*. Handbook of Logic in Computer Science, vol II, Clarendon Press, Oxford, 1992.

[25] Toval, A., Requena, V., Alemán, J.L. OCL Tools, http://www.um.es/giisw/ocltools/

[26] Xactium, XMF-Mosaic, http://www.xactium.com/

Dynamic Reconfiguration of Software Architectures Through Aspects

Cristóbal Costa[1], Nour Ali[1], Jennifer Pérez[2], José Ángel Carsí[1] , and Isidro Ramos[1]

[1] Department of Information Systems and Computation,
Polytechnic University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain
ccosta@dsic.upv.es, nourali@dsic.upv.es,
pcarsi@dsic.upv.es, iramos@dsic.upv.es
[2]Technical University of Madrid (UPM)
E.U. Informática, Ctra. Valencia km 7, 28051 Madrid, Spain
jeperez@eui.upm.es

Abstract. Currently, most software systems have a dynamic nature and evolve at run-time. The dynamic reconfiguration of software architectures has to be supported in order to enable their architectural element instances and their links to be created and destroyed at run-time. Complex components also need reconfiguration capabilities to evolve their internal compositions. This paper introduces an approach to support the dynamic reconfiguration of software architectures taking advantage of aspect-oriented techniques. It enables complex components to autonomously reconfigure themselves: they are capable of both having knowledge of their current configuration and reconfiguring themselves at run-time. This approach has been developed for the PRISMA aspect-oriented architectural model. A new kind of aspect has been created in PRISMA in order to provide dynamic reconfiguration services to each complex component; it is called the *Configuration Aspect*.

Keywords: Dynamic reconfiguration, software architectures, AOSD.

1 Introduction

Currently, most software systems have a dynamic nature that requires them to evolve and adapt to changes at runtime. The development of this kind of systems is a complex task since they are large and may consist of heterogeneous entities. As a result, a great effort has to be done in order to provide system reconfiguration at runtime. Software architectures [8] provide techniques for describing the structure of complex software systems in terms of architectural elements (components and connectors) and interactions among them. Software architectures have a hierarchical structure where components can be composed into complex components and these, in turn, can be composed into more complex components.

Dynamic reconfiguration [3] has to be supported in order to enable architectural element instances and links to be created and/or destroyed at run-time. In the last years, a lot of research efforts have been done to address dynamic reconfiguration of software architectures [1,2]. However, most of these approaches address dynamic

Costa, C., Ali, N., Pérez, J., Carsí, J. A., Ramos, I. Dynamic Reconfiguration of Software Architectures Through Aspects. Proc. of the First European Conference on Software Architecture (ECSA'07). 2007.
© Springer-Verlag Berlin Heidelberg

reconfiguration from a centralized point of view: a global entity is responsible of providing dynamic reconfiguration capabilities to all the architecture. However, architectures are often made up of heterogeneous components and each one has different reconfiguration needs. For this reason, complex components usually need to reconfigure their internal composition by themselves in an autonomous way.

Aspect-Oriented Software Development (AOSD) [4] proposes the separation of the crosscutting concerns of software systems into separate entities called aspects, avoiding the tangled concerns of software and allowing the reuse of the same aspect in different entities of the software system as well as its maintenance. Dynamic reconfiguration is a concern that crosscuts the architecture of those software systems that have a dynamic nature. In order to prevent the dynamic reconfiguration concern from being tangled with the rest of the architecture, a new kind of aspect called *Configuration* has been defined. Thereby, a *Configuration* aspect encapsulates the properties and the behaviour of this concern. Only few proposals have addressed dynamic reconfiguration through aspects [9], and their work is focused at the implementation level instead of dealing reconfiguration at the architectural level.

This paper presents an approach for supporting dynamic reconfiguration in software architectures by means of aspects. The approach enables PRISMA complex components to autonomously reconfigure themselves at runtime, through a *Configuration* aspect, whereas other systems or components of the software architecture are unaware of these dynamic changes.

2 PRISMA

PRISMA [5] is a model that integrates AOSD and software architectures in order to describe the architectural models of complex software systems. In PRISMA, architectural elements are specified by importing a set of aspects. Aspects are first-order citizens of software architectures and encapsulate the properties and the behaviour of *concerns* (safety, coordination, etc) that crosscut the software architecture. A PRISMA architectural element can be seen from two different views: the internal and the external. In the external view, architectural elements encapsulate their functionality as black boxes and publish a set of services through their ports. The internal view shows an architectural element as a prism (white box view). Each side of the prism is an aspect that the architectural element imports. The aspects of architectural elements are synchronized among them through *weavings* relationships.

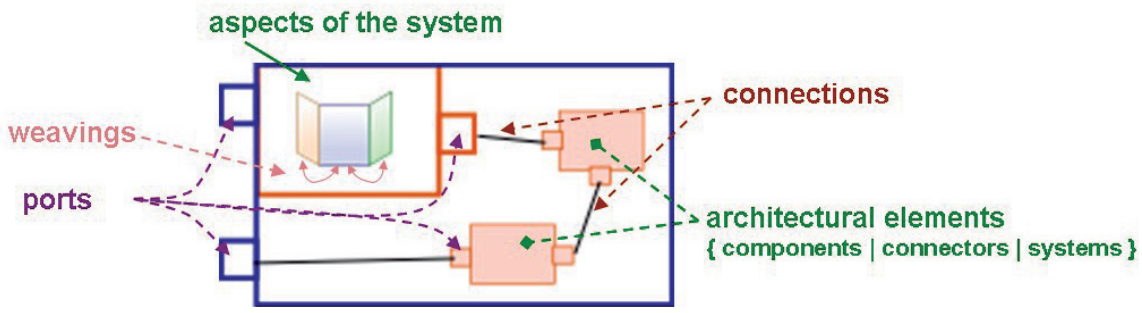


Fig. 1. PRISMA Systems

PRISMA has three kinds of architectural elements: components, connectors, and systems. Components and connectors are simple; systems are complex components. A component captures the functionality of software systems whereas a connector acts as a coordinator among other architectural elements. A PRISMA system is a complex component that includes a set of architectural elements (connectors, components and other systems), the connections among them, and its own aspects and weavings (see Figure1).

3 Dynamic Reconfiguration Through an Aspect

A PRISMA system is an architectural pattern that defines the type of architectural elements a system is composed of and the kind of valid connections among them. This architectural pattern is instantiated to a specific configuration in order to start the architecture execution. However, dynamic systems need to change their initial configuration several times as a result of its normal behaviour (always satisfying their architectural patterns). For this reason, the system configuration at run-time must be considered as a part of the system state. Thus, a system must have services to query its current configuration, and to change it at run-time. In this way, a system is aware of its current configuration and can decide whether to change it or not. A new concern has been created using the PRISMA Aspect-Oriented Architecture Description Language (AOADL) [6] to encapsulate these services into aspects. This is the *Configuration* concern. A *Configuration* aspect encapsulates every property and behaviour related to dynamic reconfiguration. Any system that needs reconfiguration capabilities to evolve its internal composition imports the *Configuration* aspect.

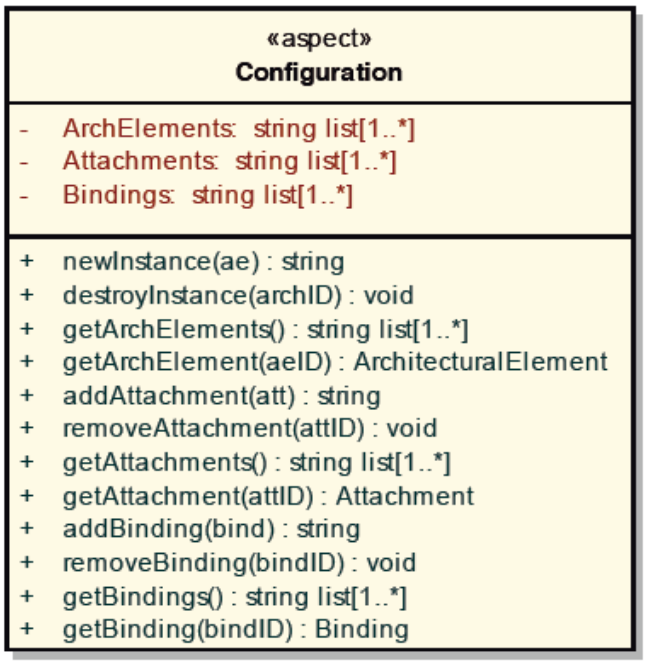


Fig. 2. Configuration Aspect

This aspect has a set of attributes that contain the current configuration of the system and a set of services that maintain and evolve this configuration. The attributes the *Configuration* aspect provides are dynamic lists which store *references* to: (i) the architectural element instances of the system at run-time (component, connector, and system instances), and (ii) connection instances of the system at run-time. The services that the *Configuration* aspect provides (see Figure 2) allow systems to know and modify the data that these attributes store (the system configuration): (i) create or destroy instances of system architectural elements, (ii) establish connections between system architectural elements, and (iii) provide query services in order to make the

system aware of its current configuration. In this way, reconfiguration characteristics are specified without extending the PRISMA AOADL with new primitives, and the reconfiguration concern is not tangled with other concerns.

The infrastructure where the software architecture is running is responsible for providing to each system instance the mechanisms of dynamic reconfiguration. PRISMA software architectures are executed by the PRISMANET middleware [7], which provides the mechanisms required for knowing and modifying the running configuration of each system instance at run-time. PRISMANET guarantees that the configuration dependencies among the different elements are preserved.

4 Conclusions and Further Work

This paper has presented an approach to dynamically reconfigure software architectures taking advantage of aspect-oriented techniques. This approach consists of providing each complex component with an aspect called *Configuration,* that provides dynamic reconfiguration services at run-time. As a result, this approach has the advantage of keeping dynamic reconfiguration properties and behaviour from being tangled with the rest of the architecture. This has been done without increasing the complexity of the PRISMA AOADL, only by using its original primitives.

Autonomous system instance reconfigurations must satisfy the constraints of the system architectural pattern. We are currently working on specifying how the reflection mechanisms ensure that the requested changes do not violate the restrictions described in the architectural pattern of the system.

Acknowledgements. This work is funded by the Dept. of Science and Technology (Spain) under the National Program I+D+I, META project TIN2006-15175-C05-01. This work is also supported by a fellowship from Conselleria d'Educació i Ciència (G. Valenciana) to C. Costa.

References

1. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. In: WOSS'04. Proc. of 1st ACM SIGSOFT Workshop on Self-Managed Systems, Newport Beach, California, pp. 28–33. ACM Press, New York (2004)
2. Cuesta, C.E.: Dynamic Software Architecture Based on Reflection (in Spanish). PhD Thesis, Department of Computer Science, University of Valladolid, Spain (2002)
3. Cuesta, C.E., Fuente, P.d.l., Barrio-Solárzano, M.: Dynamic Coordination Architecture through the use of Reflection. In: SAC 2001. Proc. of 2001 ACM Symposium on Applied Computing, pp. 134–140 (2001)
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
5. Pérez, J.: PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Polytechnic University of Valencia, Spain (2006)

6. Pérez, J., Ali, N., Carsí, J.Á., Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 123–138. Springer, Heidelberg (2006)
7. Pérez, J., Ali, N., Costa, C., Carsí, J.Á., Ramos, I.: Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. In: Proc. of 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, pp. 97–108 (June 2005)
8. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
9. Rasche, A., Polze, A.: Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft.NET. In: ISORC'03. Proc. 6th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing, Hakodate, Japan, pp. 164–171 (2003)

Establishing Relations among Software Architecture Views through MDA for SPL

Rogelio Limón C.[1], Ma. Eugenia Cabello E.[2], Isidro Ramos S.[3]

Polythechnic University of Valencia,
Camino de Vera s/n, 46022 Valencia, Spain
E-mail: [1]rlimon@dsic.upv.es, [2]mcabello@dsic.upv.es, [3]iramos@dsic.upv.es

Abstract

Software architecture views capture the concerns that are present in the steps previous to the detailed design phases of a software system. Each view represents a different structure of a system for analysis and documentation purposes. Although each view can be designed separately, they are linked because they describe the same system. These links must be explicitly established and maintained in order to achieve synergy in their design and also to preserve consistency among the views when they change with evolution or maintenance purposes. This work uses the Model-Driven Architecture (MDA) approach to establish the links among the architectural views by means of relations between models. This work shows how the relations among the Software Product Lines (SPL) views can be established by means of strategies used in MDA.

Key words: software architecture views, meta-models, models, MDA, software product lines.

1. Introduction

Nowadays, software architecture (SA) is used in the development of complex software systems to shape the fundamental structures that support the detailed design phase of these systems. Software architectural views (SAV) represent the different structures that a system can have and shape the concerns present at the design level in different ways.

Even though architectural views can be shaped separately, these views establish relations among each other that help maintain the cohesion of the system. These relations can be traced and their corresponding elements at lower levels of design can also be traced.

The relations of each architectural view must be identified and explicitly shaped in order to achieve the following objectives: i) generate a view using another view as reference, ii) make rules to assure conformance between the architectural view and the code, iii) preserve the consistency among views when a view is modified (for maintenance or evolution purposes), and iv) document the software architecture.

The studies made to date do not fully cover all the objectives stated above. For objective (i), there are methods such as the Attribute-Driven Design (ADD) approach presented in [1], which uses a view to design another view. However, the relations between them are not explicitly specified and are not recorded. For objective (ii) there are many proposed methods; however their solutions are incomplete as stated in [2]. For objective (iii), a specific algebra to deal with the consistency is used in [3], but it only focuses on one type of view, [4] deals with relations of

Limón, R., Cabello, M. E., Ramos, I. Establishing Relations among Software Architecture Views through MDA for SPL. Proc. of the 14th International Congress on Computer Science Research (CIICC'07). 2007.

multiple views in a formal way, but its concept of view is too specific to apply to other types of views. For objective (iv), there are many studies for documentation, e.g. [5].

Our work is ilustrated by means of a Software Product Lines (SPL) case study in the decision support systems domain. The SPL have emerged in an effort to control and minimize the high costs of the software development process and to reduce the time to market of these new products [6]. This approach is based on having a base design that is shared by all the family members. Thus, a specific product can benefit from the common parts of the software. The base design can be re-used in different products by adding different features that caracterize them.

This paper is structured as follows: Section 2 shows the relation between SAV and MDA. It explains how to use the MDA to relate SAVs with each other by means of transformations among the architectural models. Section 3 identifies the relations among the meta-models of the SAVs considered. Section 4 shows how to relate SAVs at the model level. Section 5 illustrates this approach in a case study. Section 6 presents our conclusions.

2. Relating SAVs and the Model-Driven Architecture

2.1 Software Architecture Views

The SA include by a set of views that represent the different structures of a software system [1]. The views can be used to separate concerns and several models can be used to represent them.

There are several approaches describing what an architectural view should be. Among the most relevant are the following: SEI [1], IEEE-1471 [5], and Kruchten [7]. Even though there are some differences in how they interpret a view, there are coincidences in the type of elements and relations that are considered in each one. In all of these approaches, components and connectors are considered as basic building blocks, and the module concept is included in different ways.

The focus of this work is to analyze and deal with the relations between (at least two) types of architectural views. Therefore, the component-connector (C-C) and modular view-types proposed in [1] have been chosen as the most appropriate approaches to apply to our proposal.

The main goal is to shape the correspondence rules at the meta-level and perform the transformation at the model level.

Figure 1(a) shows an example of two correspondence relations at the meta-model level: in this case, R1 indicates that each module element (Module) of the <<Modular View>> has a correspondence with an element (component) of the <<C-C View>>, and R2 indicates that a <<Uses>> of the <<Modular View>> has a correspondence with a connector element of the <<C-C View>>. These relations are used to perform the transformation at the model level as shown in Figure 1(b). This is an example of objective (i) of the relation of software architecture views, described in section 1.

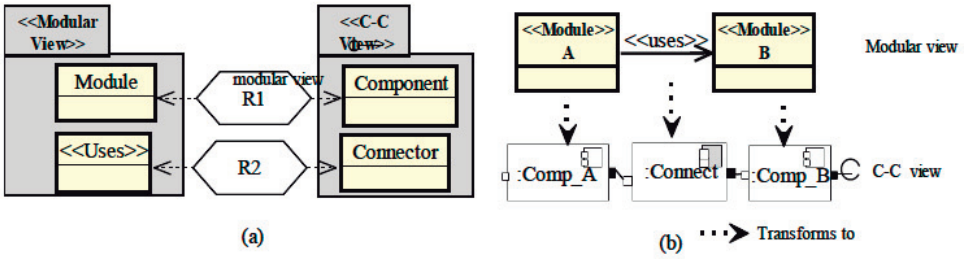


Figure 1. Example of: (a) Relations between the two views at the Meta-model level, (b) Transformation from a view model to another view model

2.2 The Model-Driven Architecture and the Software Architecture Views

The MDE is a novel paradigm that reduces the software complexity introduced by the current diversity in technology. It uses methodologies and support tools that are based on models with a high level of abstraction (meta-models) instead of those of a specific technology. MDA [8] is a proposal of the OMG (Object Management Group) for MDE.

MDA creates system models with a high level of abstraction. These models can be refined using transformations to obtain different models at the code level [9]. The abstract model is called PIM (Platform-Independent Model), and the model on a specific platform is called PSM (Platform-Specific Model). The PSMs are generated by a transformation from PIM to PSM.

SAVs can be shaped as PIMs, even though they represent specific systems. They have features that are applicable to a set of similar systems (families of software systems) so that the same SAV can be adapted to systems in different platforms, i.e., to different PSMs. Furthermore, SAVs are formed with elements of a high level of abstraction, which are by means of described languages (ADLs) or notations that are independent of the technological platform. These elements can be mapped to code level. Therefore, the MDA can be used not only to refine a SA, but also to make transformations among architectural views that are at the same level of abstraction [9].

2.3 Using MDA to Build Relations among SAVs

Three types of relations among the SAVs studied in this work have been identified: i) the relations that are present between the modular view and the C-C view. These relaciones generate a view at the model level using each other as reference; ii) the relations that are formed between the C-C view model and other models that define the variability of the SPL. One model captures the variability of the domain: V_1 (in the case study the variability of the decision systems domain), and other: V_2 captures the variability of the application domain (medical diagnosis, education diagnosis, etc.) Both models V_1 and V_2 conforms its meta-models MMV_1 and MMV_2 respectively. Model V_1 is used to refine the C-C view model; and iii) the relations established when a view is adapted to a specific application domain system (at different levels of abstraction) using model V_2.

The relations *i* and *ii* are shaped using the MDA frame as shown in Figure 2. The Meta Object Facility (MOF) [10] provides the language for expressing the meta-models. It is used to specify itself (level M3) and to specify the meta-models of the views and their interactions (level M2).

The relations R1 and R2 are also established at level M2. R1 is related to *i*, and R2 is related to *ii*.

The next level (M1) is where the transformations T1 and T2 are carried out. A first model is obtained by transformation T1, which is called skeleton model, using the modular view model and the variability of the domain conceptual model (DCM):V_1 as sources. This first version is

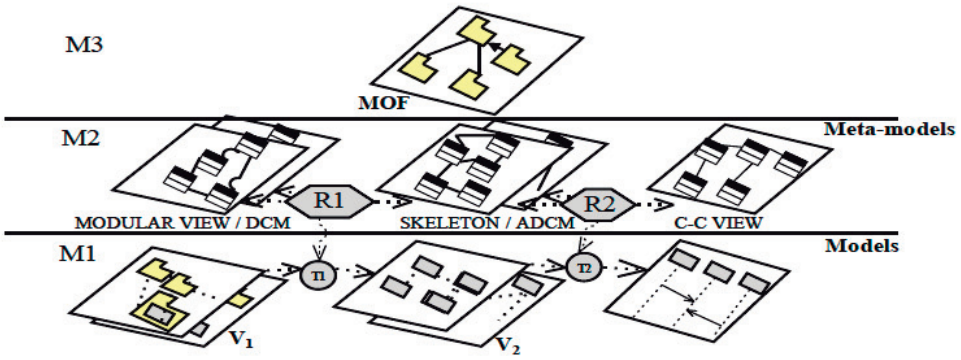


Figure 2. Relations among meta-models and transformation among models in the SPL scenario

refined by means of transformation T2 using the variability of the application domain conceptual model (ADCM):V_2 as a refinement. It is:

$$R1 \underset{def}{\subseteq} MM_{MODULAR\ VIEW} \times MMV_1 \rightarrow MM_{SKELETON\ C\text{-}C};$$

$$R2 \underset{def}{\subseteq} MM_{SKELETON\ C\text{-}C} \times MMV_2 \rightarrow MM_{C\text{-}C\ VIEW};$$

UML-class diagram meta-model is used as MMV_1 and MMV_2

2.4 Transformation process of the architectural views models.

In order to achieve the transformations among the view models, a process that allows the design of the C-C view using the modular view, the domain conceptual model:V_1, and the application domain conceptual model:V_2 as inputs, is proposed.

The tasks and elements involved in this transformation process are shown in Figure 3. The meta-model specifications of the modular view, the C-C view, and the UML-class diagram are the first elements that are required to start this process. Next the correspondence relations between the modular view and the skeleton view, and the UML-class diagram are identified, and specified by means of a language. The QVT-Relational language and the MOF notation are used for this task. All these tasks are performed only one time since their products (elements) are used as a template models. The design task of the models is the task that can be performance several times as required.

The modular view model, the DCM, and the relations established between the views of meta-model are used as the source in the transformation task, which produces skeleton view model.

Next, a second transformation is carried out to refine it to the C-C model. To do it the ADCM and the skeleton model are used as source to produce the C-C view model..

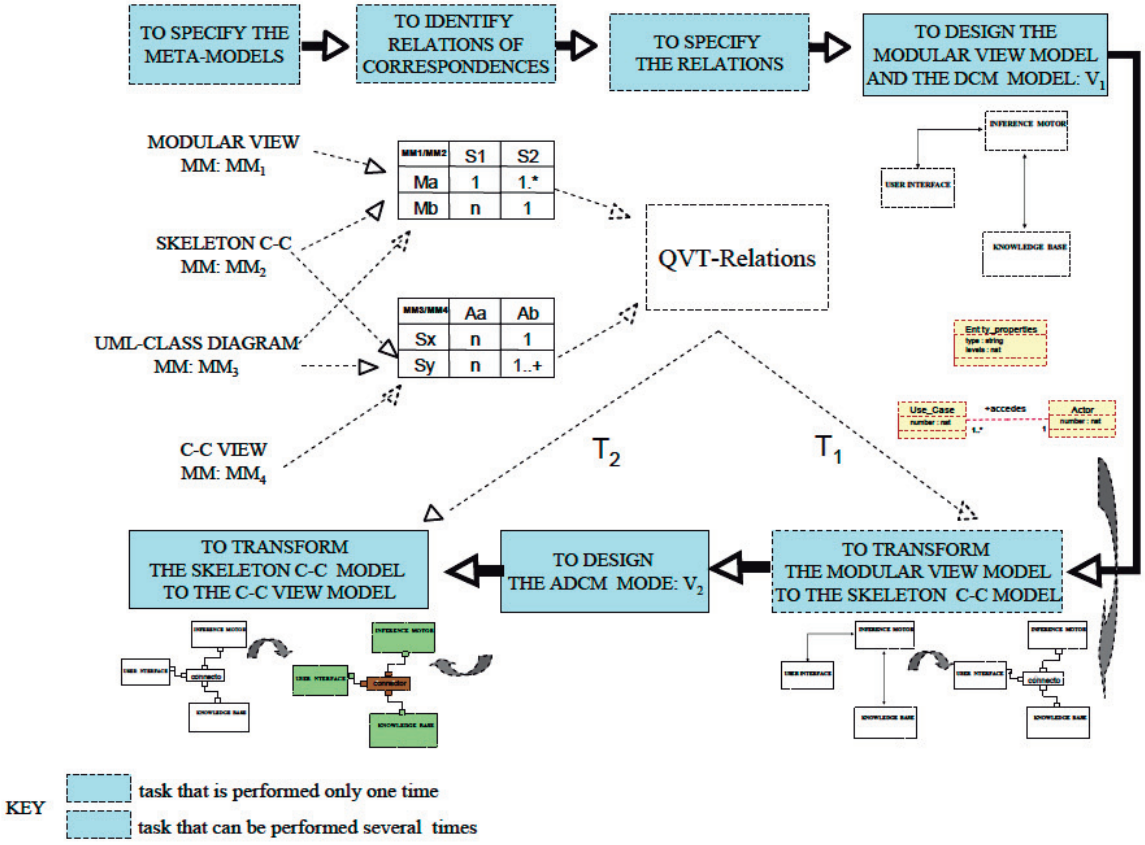


Figure 3. The tasks and elements involved in the transformation process of the architectural view models.

3. Specifying the Meta-Models

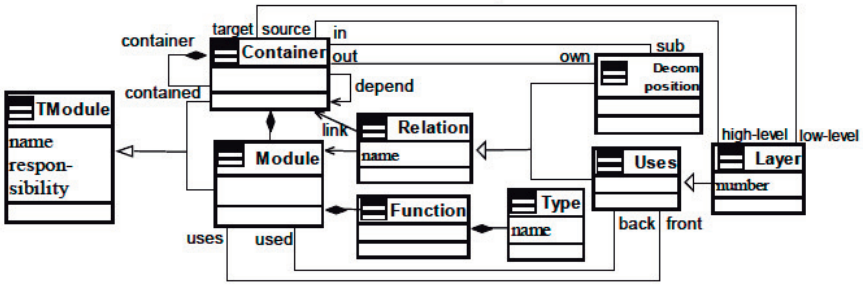
For the conformation and specification of the meta-models of the modular and C-C views, several proposals [4][5][7] have been analyzed. The MOF graphic notation has been used for the specification of these meta-models.

3.1 Specification of the Modular View Meta-Model : MM $_{\text{MODULAR VIEW}}$

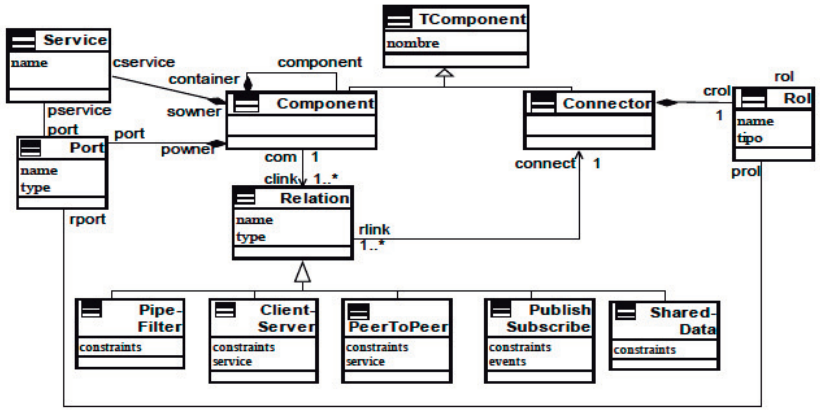
The main element considered for this view is the module. We have made a distinction between a module that is used as a container for more modules and the module that only carries out a single function.

Figure 4(a) shows the meta-model for this view. The relation class represents the styles presented in [4] (decomposition, uses and layers). These styles are treated as relations since what distinguishes one style from another is the type of relation. The labels in the links are useful for

indicating how the relation is made; for example in Figure 4(a) the '*used*' label indicates which element of the Module class will be used by another element of the Module class.



(a)



(b)

Figure 4 (a) Modular view meta-model (b) C-C view meta-model. (MOF notation)

3.2 Specification of the C-C View Meta-Model : MM C-C VIEW AND MM SKELETON C-C MODEL

The C-C view meta-model is depicted in Figure 4(b). This Figure shows the *Component* class and *Connector* class as the main elements. Both are derived from a more general component class (*TComponent*). In a similar manner to the above meta-model, the way in which these elements are related corresponds to their styles or relations (Pipe-Filter, Client-Server, PeerToPeer, Publish-Suscribe, Shared-Data). These styles are inherited from the *Relation* class. This class links the components by means of the *Connector* class.

The interactions among the components of this view are refined using the UML-classes meta-model; this meta-model specifies how the components (objects) are relationed. The UML-classes meta-model is the UML 2.0 meta-model [11].

4. Establishing relations among the meta-models

Before the relations are shaped, a language must be chosen to represent them correctly. For the specific case of MDA, OMG proposes the Meta Object Facility (MOF) to express the meta-models. It uses Query/Views/Transformations (QVT) [10] to establish the relations among these meta-models. Specifically, the QVT-relational language is used to describe the relations.

4.1 Identifying Correspondence Relations

The correspondence among each element of the meta-models previously specified must be identified, taking into account the way in which their rules are represented through QVT. The source and target meta-models must be identified first. In this case the source meta-models are the modular view and the MMV_1 variability meta-models. The target meta-model is the skeleton C-C model, since this model is obtained from the previous ones. This means that the rules considered in the relations for the elements of the modular view and V_1 , can only be of type "**check only**" (to verify the elements) and of type "**enforce**" to create the elements of the skeleton C-C model.

Note that not all the elements between the models considered have correspondence relations. Table 1 shows some of the identified correspondence relations, their name, type (according QVT), and the elements involved (classes).

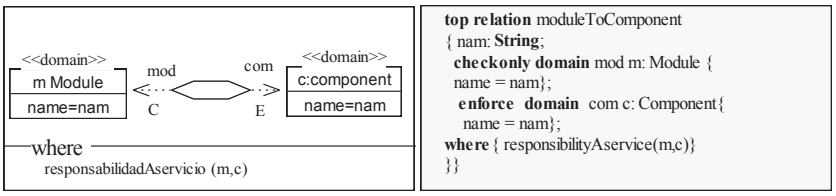
Relation Identification $R_1 = R_2$ X R_1	Type	Classes involved by view	
		Modular view	C-C view
moduleToComponet	top	Module	Component
functionToService	-	Module, Function, Type	Component, Service, Port
rUsoModToConnector	top	Uses, Module, Service	Connector, Role, Component, Port; Service
rCompoModToCom	top	Composition, module	Component

Table 1. Some relations identified in the modular and C-C meta-models

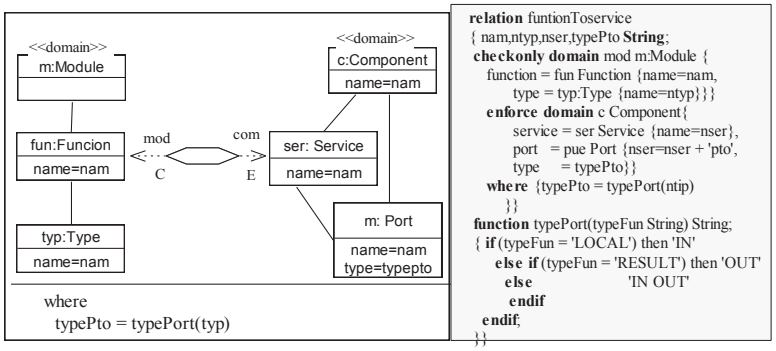
4.2 Specification of the relations between the software views.

The relations shown in Table 1 are specified by means of a MOF diagram. The code for these relations is written in the QVT-relational, this code and the diagrams are shown in Figure 5.

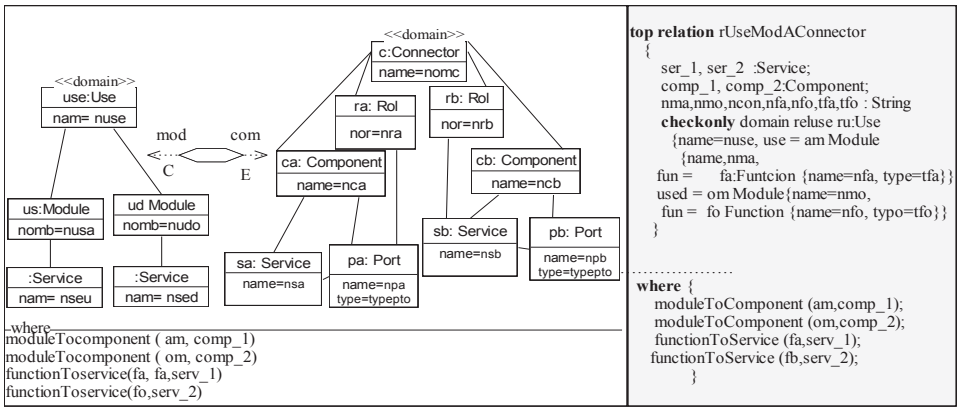
In order to gain clarity the prefix "skeleton" will be omitted from now, i.e "skeleton component" will be written as "component", and "skeleton connector" will be written as "connector."



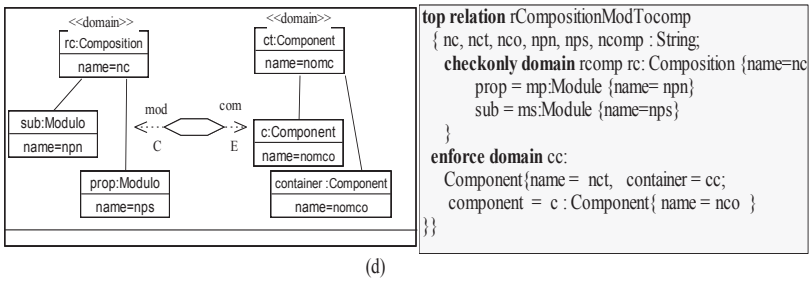
(a)



(b)



(c)



(d)

Figure 5. Diagrams and codes of the relations: (a) moduleToComponent, (b) functionToservice, (c) rUseModToConnector, (d) rCompositionModTocomp

Relation moduleToComponent.

This relation maps each module with a component. Figure 5 (a) shown two types of relations: the ***checkonly*** type and the ***enforce*** type. The object of the Module domain is of the ***checkonly*** type. In contrast, the object of the Component domain is of the ***enforce*** type. This creates an object of the *Component* class that is related to the Module class. The where clause indicates a call to the functionToService relation, which relates an object of the Module class with an object of the Component class. The diagram and the corresponding code are shown in Figure 5(a).

Relation functionToService.

This relation implies that a function of the module meta-model will generate a service of the Component class. When this occurs, the type of the Function will generate a port. In the where clause, the name of the port is obtained by calling the function typePort.

When the relation is executed, the classes that are in the source meta-model can only be verified and the classes that are in the target meta-model will be created (only the ServiceToPort class). The diagram and the corresponding code are shown in Figure 5(b).

Relation rUseModToConnector

The *'uses'* relation is transformed from the modular meta-class to a link between a connector and two components in the C-C meta-class. This is shown in Figure 5(c). The creation of objects for this relation is from 1 to n because a relation of two components is generated through a connector. Figure 5(c) shows part of the code to illustrate how the relations are invoked in the *'where'* clause to create the relations among module, component, and functionToService.

Relation rCompositionModToComp

The set of modules will also generate a set of components. However, in this case when a component is created (container) a subordinate is created (c) inside it. This is shown in Figure 5(d).

5. Applying transformation at the model level.

To illustrate how the transformations at the model level are applied in the transformation process (see Figure 3), we present a case study. This case study consists of designing decision support software architectures based on SPL. The stakeholders only introduce the variability of the domain by means of DCM:V_1, and the variability of the application domain by means an ADCM:V_2. After introducing these data, it will generate a base architecture "ad hoc" to the case study. The base architectures share a generic architecture: the decision support system architecture captured by the modular view.

As a previous step to the transformation, the modular architectural view model is designed from the specification given above using the method proposed in [12]. The MOMENT tool [13] developed on Eclipse [14] is used to design this model (and its corresponding meta-model) as shown in Figure 6. The modular view model is depicted in the Figure 6 (a). This model is made up of three modules: *InferenceMotor, KnowledgeBase*, and *UserInterface*. These are all linked

with each other through *dependence* relations. Next, the transformation task for producing the skeleton model (target) is performed (see Figure 6 (c)). To do this, the DCM model is used as the source (see Figure 6 (b)). As the Figure 6 shows, a component skeleton was produced for each module, and each *dependence* relation generate a connector skeleton.The relations established in section 4.2 are the rules that are applied to generate the new model (using the meta-model as templates). In this case, the *moduleToComponent* and *rUsoModToConnector* relations were applied. The transformation is carried out with the MOMENT tool.

The transformation generates the C-C view model shown in Figure 7. Since the generated skeleton model previously included hollows (i.e. these skeletons are empty components and connectors) because the capture of the variability has not yet been modelled. A second transformation must be carried out to do this. The ADCM model is required for this second transformation. It is designed taking into account the requirement specification (not shown here). The designed model is shown in Figure 7(b).

The skeleton model (see Figure 7 (a)) and the ADCM model are used as source models in this second transformation. As before, the MOMENT tool is used for this task; however, in this case, two source models are combined as shown in Figure 7. The final architectural view model is then generated. This is shown in Figure 7 (c). This model and the modular architectural view model make up the software architecture of the decision support system.

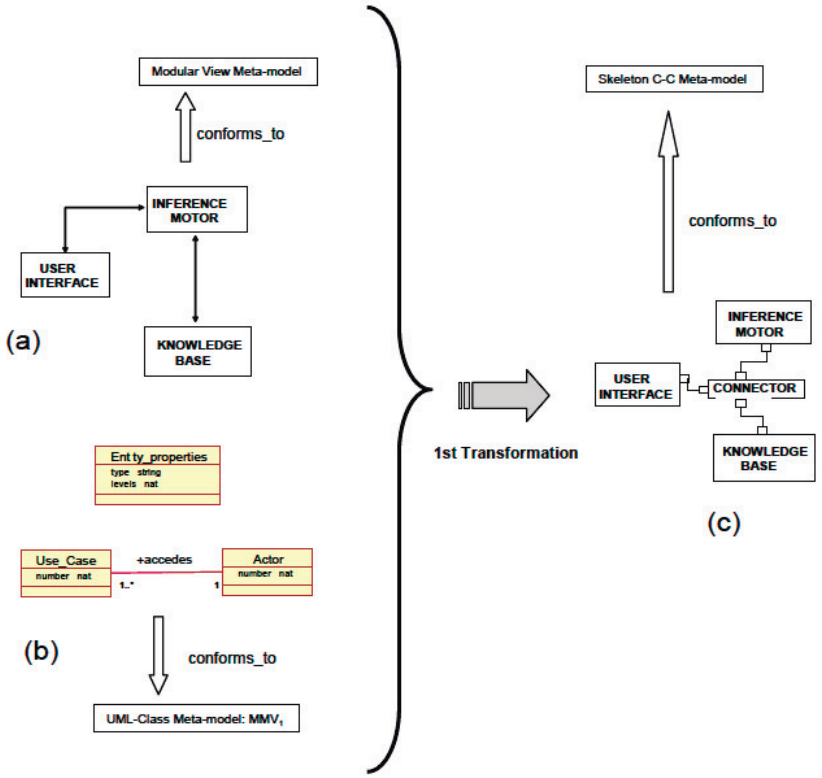


Figure 6. The first transformation of models: (a) The Decision Support System view modular model; (b) The Domain Conceptual Model; (c) The Skeleton C-C model generated in the first transformation.

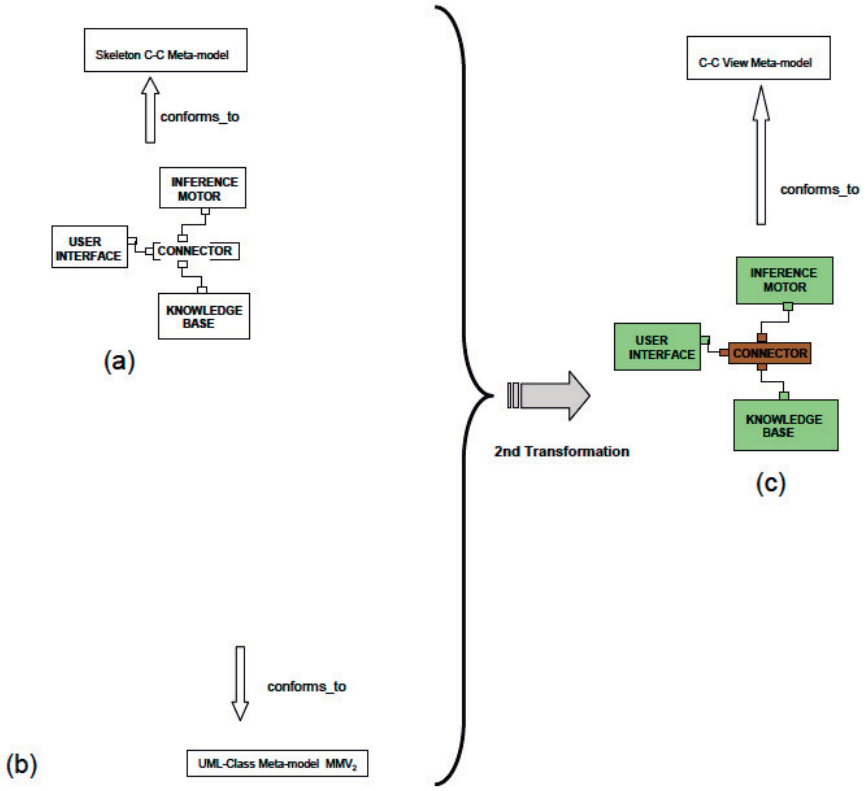


Figure 7. The second transformation of models: (a) The Skeleton C-C model generated in the first transformation; (b) The Application Domain Conceptual Model; (c) The C-C architecture view model generated in the second transformation. This is the definitive view model.

6 Conclusions

By modelling software architectures using MDA, we have proposed a novel way to design and produce an architectural view. This approach can be extended to other types of views in software engineering.

The use of the transformation process among models proposed here, allows us to: a) establish meta-models for two views (modular and C-C), and identify and establish correspondence relations between them; b) generate a new view using another view as reference; c) refine the new model through a second transformation combining the new view and the ADCM:V_2 to obtain the C-C view model; d) adapt the MOMENT development tool to make these transformations; e) apply a consistent transformation process attached to the standard specification of the OMG (MDA) by means of the QVT relation language and the MOF graphic notation to represent the relations and the transformation rules; f) adapt and use tools consistent with the language and notation used; g) apply a similar treatment to assure conformance between the C-C architectural view and the code (not here); h) prove consistency (using rules *checkonly*) and contribute to improving the documentation.

This study has also shown that the application of MDA improves the design process of the software architectural views by doing it in a systematic and automatic way.

Our C-C meta-model is the PRISMA meta-model [15], allowing us to compile the software architecture generated model using the PRISMA-CASE framework [16][17], producing .NET executable code.

References

[1] M. Shaw, P. Clements, The Golden Age of Software Architecture, *IEEE Software*, Volume: 23 Issue: 1 , pp 31-39 Date: Jan.-Feb. 2006.

[2] J. Muskens, R.J. Bril, and M.R.V. Chaudron, Generalizing Consistency Checking between Software Views, *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05),* pp. 169-180, 2005.

[3] P. Fradet, D. le Métayer, and M. Périn, Consistency checking for multiple view software architectures, *Proceedings of 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Pages: 410 - 428, 1999.

[4] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture, Views and Beyond*, Addison-Wesley, 2002.

[5] IEEE Product No. SH94869-TBR, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 2000.

[6] Clements P., and Northrop L.M., *Software Product Lines: Practices and Patterns,* SEI Series in Software Engineering, Addison Wesley, 2002.

[7] P. Kruchten, The 4+1 View Model of Architecture, *Paper published in IEEE Software* Vol.12, No.6, pp. 42-50 Nov. 1995.

[8] OMG, Object Management Group: Model Driven Architecture (MDA). 2004. http://www.omg.org/cgi-bin/doc?formal/03-06-01

[9] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise,* Addison Wesley Professional, ISBN 978-0-321-19442-8, 2003.

[10] OMG, Revised Submission for MOF 2.0. Query/View/Transformations RFP (ad/2002-04-10); OMG Document ad/2005-07-1(2005).

[11] Unified Modeling Language 2.0: Superstructure, http://www.omg.org/docs/formal/05-07-04.pdf.

[12] R. Limón, I. Ramos, and J.J. Torres, Designing Aspectual Architecture Views in Aspect Oriented Software Development, *Computational Science and its Applications*, LNCS 3983, pp, 726-735, 2006.

[13] The MOMENT Project, http://moment.dsic.upv.es/

[14] Eclipse: http://www.eclipse.org/

[15] Pérez J., *PRISMA: Aspect-Oriented Software Architectures,* PhD. Thesis of Philosophy in Computer Science, Polytechnic University of Valencia, Spain, 2006.

[16] Cabedo R., Pérez J., Carsí J.A. y Ramos I., Modelado y Generación de Arquitecturas PRISMA con DSL Tools, en *Actas del IV Workshop DYNAMICA*, Archena, Murcia, España, 2005

[17] Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I., PRISMANET: Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", en *Actas de JISBD'05*, Granada, España, , pp. 27-34, 2005.

Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 50 (2008) 969 990
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems

Jennifer Pérez [a,*], Nour Ali [b,1], Jose A. Carsí [b,1], Isidro Ramos [b,1], Bárbara Álvarez [c,2], Pedro Sanchez [c,2], Juan A. Pastor [c,2]

[a] Technical University of Madrid (UPM), E. U. Informática, Carretera de Valencia km.7, E 28031 Madrid, Spain
[b] Department of Information Systems and Computation Polytechnic, University of Valencia, Camino de Vera s/n, E 46022 Valencia, Spain
[c] Division of System and Electronic Engineering Polytechnic, University of Cartagena, Campus Muralla del Mar s/n, E 30202 Cartagena, Spain

Received 6 November 2006; received in revised form 31 July 2007; accepted 27 August 2007
Available online 15 September 2007

Abstract

Aspect Oriented Software Development (AOSD) has emerged as a new approach to develop software systems by improving their structure, reuse, maintenance and evolution properties. It is being applied to all stages of the software life cycle. In this paper, we present the PRISMA approach, which introduces AOSD in software architectures. PRISMA is characterized by integrating aspects as first order citizens of software architectures. This paper shows how the PRISMA methodology is applied to develop a case study of the tele oper ation system domain. We illustrate how the PRISMA approach can improve the development and maintenance processes of these kinds of industrial systems.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Methodology; Aspect Oriented Software Development (AOSD); Component based software development (CBSD); Aspect oriented software architectures; Tele operation domain; Software architecture design

1. Introduction

Aspect-Oriented Programming (AOP) [23] proposes the separation of the crosscutting-concerns of software systems into separate entities called *aspects*. This separation avoids the tangled code of software and allows the reuse of the same aspect in different entities of the software system (objects, components, modules, etc.). Despite the fact that the aspect-oriented paradigm emerged from the implementation level, its use is being extended to all stages of the software life cycle. As a result, Aspect-Oriented Software Development (AOSD) has emerged in order to extend the advantages that aspects provide to every stage. One interesting stage where AOSD is being introduced is the software architecture stage. This interconnects the requirements and the implementation stages [16] in order to analyze the software structure of complex, distributed, heterogeneous and reusable software systems [17].

Software architectures allow us to describe the structure of software systems by hiding low-level details and abstracting the high-level important features [40]. This structure is usually represented in terms of computational elements and the interactions among them. As a result, software architectures make software systems simpler and more understandable. Their main advantages are: the high-level of abstraction, the reuse of software, and the mechanisms to analyze properties, validate software requirements and estimate the cost of the development and maintenance processes. In addition, software architectures establish the bases and guidelines for designing complex and large software systems.

* Corresponding author. Tel.: +34 91 336 78 82.
E mail addresses: jenifer.perez@eui.upm.es (J. Pérez), nourali@dsic.upv.es (N. Ali), pcarsi@dsic.upv.es (J.A. Carsí), iramos@dsic.upv.es (I. Ramos), balvarez@upct.es (B. Álvarez), pedro.sanchez@upct.es (P. Sanchez), juanangel.pastor@upct.es (J.A. Pastor).
[1] Tel.: +34 96 387 73 50.
[2] Tel.: +34 96 832 54 80.

0950 5849/$ see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2007.08.007

Pérez, J., Ali, N., Carsí, J. A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J. A. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Journal of Information and Software Technology 50 (2008) . 2008.

Some proposals for the integration of software architecture and AOSD have emerged to take advantage of both approaches [9,7]. Each one introduces aspects in their Architecture Description Languages (ADLs) in a different way: as a component [31], as a connector among components [41], as a view of the architecture [21], etc.

PRISMA is an approach that integrates the software architecture and the AOSD approaches in order to take advantage of both approaches. The PRISMA approach is based on its model [37] and its formal Aspect-Oriented Architecture Description Language (AOADL) [39]. The PRISMA model is a symmetrical model because it does not consider functionality as a kernel entity different to aspects and it does not constrain aspects to specify non-functional requirements [19]; functionality is also specified as an aspect. As a result, PRISMA provides a homogeneous treatment to functional and non-functional requirements. The PRISMA model can be classified as a multidimensional model [33,34]. Aspects have been introduced in the PRISMA AOADL as a new concept rather than simulating the aspect using other architectural terms (components, connectors, views, etc.). In PRISMA, aspects are first-order citizens of software architectures and represent a specific behaviour of a *concern* (safety, coordination, etc.) that crosscuts the software architecture. The same aspect can be imported by each one of the architectural elements (components and connectors) that need to take into account this concern behaviour. As a result, an architectural element is defined by a set of aspects that describe it from different *concerns* of the architecture. The PRISMA approach takes advantage of the notion of aspect from the beginning of the system definition. The change of a property only requires the change of the aspect that defines it, and as a result, each architectural element that imports the changed aspect is also updated. In addition, it is important to emphasize that the PRISMA AOADL is a formal technology independent language that facilitates the automatic code generation and the validation of architectural and aspect features of the system.

This paper presents the methodology defined for developing software systems based on PRISMA aspect-oriented software architectures. This methodology follows the Model-Driven Development (MDD) paradigm. This paradigm is based on models that use automatic generation techniques in order to obtain the software product. The methodology is divided into three stages: detection of architectural elements and aspects, architecture modelling, and code generation and execution. These three stages are applied by the analyst of the software system in an iterative and an incremental way depending on his/her needs. In this paper, we focus on the architecture modelling stage as it is explained in detail in order to illustrate how the PRISMA combination of AOSD and software architectures can improve the development and maintenance processes of software.

The methodology is explained using a specific domain in order to illustrate its application to a real complex system. This domain is the robotic tele-operation systems. Tele-operated robots are software intensive systems that are used to perform tasks that human operators cannot carry out because of the dangerous nature of the tasks or the hostile nature of the working environment. The industrial case study where PRISMA is currently being put into practice is the EFTCoR system. The EFTCoR [13] is a tele-operated system that develops a family of robots whose purpose is to clean the hulls of ships. However, it is too broad and complex to present in a paper, so we illustrate the main ideas by showing the small-scale *TeachMover* tele-operated teaching robot to illustrate the PRISMA methodology. As a result, this paper presents the application of a software engineering approach to develop a robotic system. In this way, robotic systems have taken advantage of the benefits provided by the PRISMA software engineering approach, specifically, the reuse of components and aspects, as well as their maintainability, evolution, etc.

The structure of the paper is the following: Section 2 introduces the purpose of this work. Section 3 presents an overview of the PRISMA model in order to describe how PRISMA combines software architectures and aspects. Section 4 presents a brief summary of related work. Section 5 presents the PRISMA approach by focusing on the methodology to develop PRISMA aspect-oriented software architectures, putting PRISMA into practice by specifying the TeachMover robot software architecture. Section 6 details the advantages of the PRISMA framework. Finally, conclusions and further work are presented in Section 7.

2. Motivation

Software architectures allow us to describe the structure of software systems. This structuring improves software system understanding, development, and maintenance [40,15]. Software architectures are usually crosscut by common concerns of the system domain. These crosscutting-concerns are spread throughout the architectural elements of software architectures. As a result, the crosscutting-concerns are repeated in the architectural elements that they affect, and they are tangled with the other concerns of architectural elements [50]. The repetition of crosscutting-concerns throughout software architectures increases the volume of code and the difficulty of preserving the consistency of the software system when changes are introduced. In addition, tangled concerns make the maintenance of a specific concern more difficult because it is very difficult to locate the correct place where changes should be introduced. As a result, the best solution for dealing with the crosscutting-concerns is by modelling them in a separate entity. This solution is called Separation of Concerns (SoC) [35]. The Aspect-Oriented Programming (AOP) has emerged as an innovative way of applying SoC in software development [22,23,12].

AOP introduces a higher level of modularity of software and simplifies applications. These two properties generate

advantages such as: simpler code; easier development, maintenance and evolution; and a higher level of reusability. A demonstration of these advantages is presented in [23] by comparing the source code of a non-aspect-oriented application and an aspect-oriented application that were developed for the same software system. In this comparison, the aspect-oriented application had a clear advantage to the non-aspect-oriented one in the reduction of code. The results were the following:

Size of the non-aspect-oriented application: 35.213 lines of code,
Size of the aspect-oriented application: 4.628 lines of code,
Component program: 756 lines of code,
Three aspects programs: 352 lines of code,
Aspect weaver: 3.520 lines of code.

This example shows how code is reduced in aspect-oriented applications taking also into account the aspect weaver code. In addition, the aspect weaver code can be reused in other applications. However, different kinds of projects, with different requirements, sizes and programmers should be measured to make a real study about the time and space efficiency of aspect-oriented applications. The demonstration of [23] has been the starting point for studying the improvement of aspect-oriented programming. Other works have also confirmed this improvement such as the works of Garcia et al. [15] or Munelly et al. [28]. Garcia et al. compares the aspect-oriented implementation of the Gang-of-Four (GoF) design patterns [14] to their object-oriented implementation, and they demonstrate that most aspect-oriented implementations improve the separation of concerns to a great extent. On the other hand, Munelly et al. have implemented the same case study of a context-aware system using an aspect-oriented design, using an object-oriented design, and using a popular context-aware framework. After, they have evaluated the different implementations achieved using several characteristics and indicators, and they have illustrated that the aspect-oriented implementation improves the separation of concerns.

Crosscutting-concerns arise throughout the software life cycle. For this reason, despite the fact that AOP emerged from the implementation level, its use is being extended to all the stages of the software life cycle. As a result, Aspect-Oriented Software Development (AOSD) has emerged to gain the advantages that aspects provide in every stage of the software life cycle.

The combination of software architectures and AOSD for structuring software is a step forward in improving the development, maintenance and evolution processes of software, as well as in reducing their costs. The PRISMA approach combines AOSD and software architectures to provide the advantages that this combination offers.

There is a wide variety of domains that can take advantage of the PRISMA approach. We have focused on the tele-operation domain in order to cope with the problems of these real industrial systems using software engineering techniques. The importance of considering the software architecture in robotic tele-operated systems is well known [8]. However, despite the fact that tele-operated systems usually have many common requirements in their definition and many common components in their implementation, it is impossible for a single architecture to be flexible enough to cope with all of the variability of the domain. Therefore, a further step is needed to provide a flexible and extensible architectural framework to develop systems with different requirements and commonalities. In this sense, there have been numerous efforts to provide developers with frameworks such as [6,52] and [43]. All of them make very valuable contributions that simplify the development of systems. However, the way that the component-oriented approach has been applied may reduce part of its benefits. They are object-oriented or component-oriented frameworks that rely on object-oriented technologies and that highly depend on a given infrastructure (Linux O.S. and the C++ language). They are dependent on the technology and infrastructure and they use specific programming languages. As a result, a technology-independent language that provides mechanisms to define abstract software architectures that can be mapped into specific software architectures is needed.

At the same time, tele-operated systems have a wide range of common concerns in their domain. These concerns can be modelled as aspects in order to take advantage of AOSD. Some of these candidate aspects of the tele-operation domain are distribution, safety, mobility, security, coordination, etc.

The EFTCoR [13] is a tele-operated system that develops a family of robots whose purpose is to clean the hulls of ships. These robots are specifically designed for ship hull maintenance and the performance of different requirements. For example, in the EFTCoR system, its *safety* requirements are extremely important. This is due to the fact robot movements are very dangerous for the operators and the environment that surrounds them. Most of EFTCoR architectural elements must take these safety requirements into account. As a result, safety is presented as a crosscutting-concern in this system, and this concern should be developed by providing its separation from the rest of concerns in order to avoid tangled code inside architectural elements.

Therefore, an approach that integrates the definition of technology-independent software architectures and the separation of concerns can be a great contribution to the tele-operated domain. As a result, in this paper we present how the PRISMA approach copes with the tele-operation domain needs.

3. An overview of the PRISMA model

The PRISMA approach is based on the PRISMA model [37], its Aspect-Oriented Architecture Description Language (AOADL) [39], its methodology, and its tool [38]. The PRISMA model is described by its metamodel, which

allows the definition of the PRISMA properties in a precise way.

The PRISMA model allows the description of software architectures of complex and large systems. From the aspect-oriented point of view, PRISMA is a symmetrical model that does not distinguish a kernel or core entity to encapsulate functionality; functionality is also defined as an aspect [19]. One concern can be specified by several aspects of a software architecture, whereas a PRISMA aspect represents a concern that crosscuts the software architecture. This crosscutting is due to the fact that the same aspect can be imported by more than one architectural element of a software architecture. In this sense, aspects crosscut those elements of the architecture that import their behaviour (see Fig. 1). However, the concern is not tangled through the architecture, it is encapsulated inside one or more aspects.

A PRISMA architectural element can be seen from two different views: internal and external. In the external view (black box view), architectural elements encapsulate their functionality as black boxes and publish a set of services that they offer to other architectural elements (see Fig. 2b). These services are grouped into interfaces to be published through the ports of architectural elements. Each port has an associated interface that contains the services that are provided and requested through the port. As a result, ports are the interaction points of architectural elements.

The internal view shows an architectural element as a prism (white box view). Each side of the prism is an aspect that the architectural element imports. In this way, architectural elements are represented as a set of aspects (see Fig. 2a) and the *weaving* relationships among aspects.

Since PRISMA is a symmetrical aspect-oriented model that is applied at the architectural level, the weaving process indicates that the execution of an aspect service can trigger the execution of services in other aspects. In PRISMA, in order to preserve the independence of the aspect specification from other aspects and weavings, weavings are specified outside aspects and inside architectural elements. As a result, aspects are reusable and independent of the context of application and weavings coordinate the different aspects that form an architectural element. This way of specifying weavings achieves not only the reusability of the aspects in different architectural elements, but also the flexibility of specifying different behaviours of an architectural element by importing the same aspects and defining different weavings.

PRISMA has two kinds of architectural elements: components and connectors. A component is an architectural element that captures the functionality of software systems and does not act as a coordinator among other architectural elements; whereas, a connector is an architectural element that acts as a coordinator among other architectural elements. Connectors do not have the references of the components that they connect and vice versa. Thus, architectural elements are reusable and unaware of each other. This is possible due to the fact that the channels defined between components and connectors have their references (*attachments*) instead of architectural elements. Attachments are the channels that enable the communication between components and connectors. Each attachment is defined by attaching a component port with a connector port (see Fig. 3).

PRISMA components can be simple or complex. The complex ones are called systems. A PRISMA system is a component that includes a set of architectural elements (connectors, components and other systems) that are correctly attached. In addition, a system can have its own aspects and weavings as components and connectors. Since

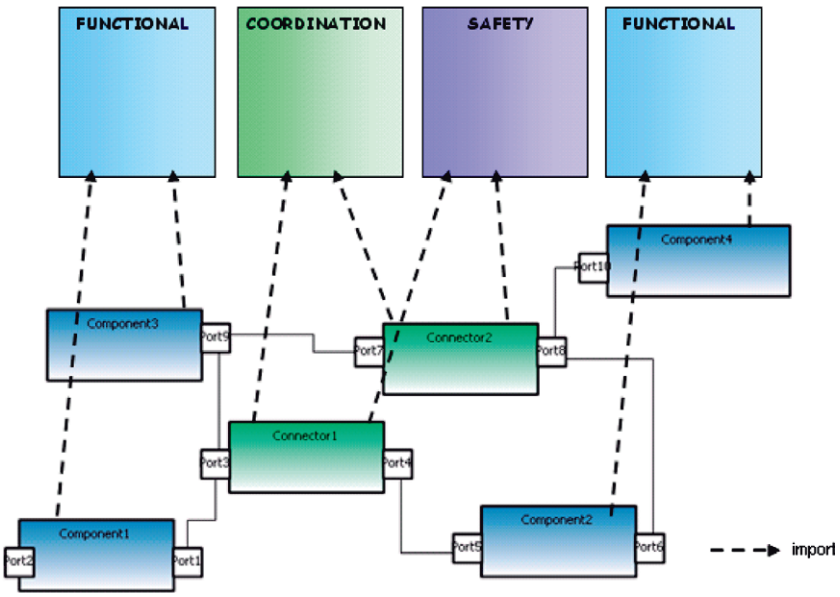


Fig. 1. Crosscutting concerns in PRISMA architectures.

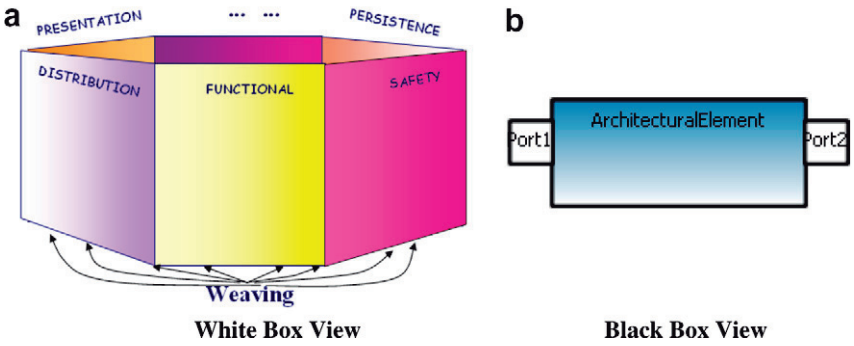


Fig. 2. Views of an architectural element.



Fig. 3. Attachments.

a system is composed by other architectural elements, the composition relationships among them must be defined. These composition relationships are called *bindings*. Bindings establish the connection among the ports of the complex component (the system) and the ports of the architectural elements that a system contains.

4. Related works

AOP introduces a set of new concepts that allow us to apply the AOP paradigm at the different levels of the software life cycle. The most import concepts are the base code, join point, pointcut, advice, aspect and weaving [11].

Base code: The base code is composed of the software units (modules, objects, components) of an application, which have been obtained as a result of a functional decomposition.

Join point: Join points are situated in the base code of an application. A join point is a semantic concept that defines a well-defined point of the execution of a base code. This point can extend the base code with the aspect code, thereby altering the execution flow of the original application.

Pointcut: A pointcut is a set of join points, which are candidates for injecting aspect code into base code at run-time, and their multiple instantiations. Pointcuts perform the weaving between the base code and the aspect code by capturing joinpoints.

Advice: An advice defines the code that should be executed at the join points of a specific pointcut. Advices are executed *after*, *before* or *instead* of the selected join points.

Aspect: An aspect is composed of pointcuts and an advice.

Weaving: The Weaving is the process that consists of coordinating the base code and the aspect code. This can be seen as the composition process of an aspect-oriented application.

In PRISMA, these concepts have been introduced at the architectural level in order to take advantage of AOP properties. The functional decomposition of PRISMA is performed by architectural elements, which allow us to structure the code by functional blocks. However, these functional blocks do not have a base code associated to them that implements their functionality (kernel, class, object, etc.). In PRISMA, functionality is also an aspect due to the fact that PRISMA follows a symmetrical aspect-oriented model. PRISMA architectural elements are composed by aspects and weavings. Since we do not make any difference between base and aspect code, our weaving process consists of coordinating the different aspects that an architectural element imports. In PRISMA, every aspect service is a joinpoint and those services that need to execute other services *after*, *before* or *instead* of them are pointcuts, and these other services that are executed as a consequence of pointcut services are the advice. As a result, the weaving is composed of a pointcut service, a temporal operator (after, before, instead) and one or more advice services.

PRISMA architectural elements provide the functional view of a software architecture, publishing through their ports, the services of the aspects that are necessary to interact with the rest of architectural elements. They made transparent to the user the fact that there is a clear separation of concern inside them and there is a weaving process that coordinates these concerns.

With regard to software architectures, a wide variety of ADLs have been proposed in the literature, and an interesting comparison among these languages has been made in the work by Medvidovic and Taylor [25]. Some proposals for the integration of software architecture and AOSD have emerged to take advantage of the two approaches. A comparison of these is presented in [9]. Each of these approaches, that are presented in [9], introduces aspects in their Architecture Description Languages (ADLs) in a different way: as a component AspectLEDA [31], JAC

[36], Jiazzi [24], as a connector among components CAM/DAOP [41], as a view of the architecture AVA [21], etc. On the other hand there are approaches such as FuseJ [48,49] that promote the idea that there are no aspects, there are only components. This assertion is well-based on the fact that aspects can be modelled using components as FuseJ, AspectLEDA, JAC, Jiazzi and others. However, it does not mean that components and aspects are the same thing. The semantics of these concepts is different as Kizcales explains in his work [23]. In addition, if aspects and components were the same, not only aspects could be modelled as components, but components could be also modelled as aspects. However, a component cannot be modelled as an aspect because a component usually implements properties of more than one concern. As a result, in order to be consistent with the meanings of both concepts, in PRISMA software architectures, an aspect is introduced as an aspect, i.e, the aspect is a new primitive in software architectures and it is not simulated by using an architectural term (components, connectors, views, etc.). PRISMA aspects are first-order citizens of the PRISMA AOADL.

The first tasks for developing software architectures are to identify which architectural elements make up the architecture and to detect the aspects that crosscut the software architecture. Several works have been proposed for the identification of components, they are usually based on a widely extended criterion that consists of the functional decomposition of software. One of these works applies this criterion to multi-agent systems [32] and another work applies this criterion to software architectures (the Architecture Based Design Method (ABD)) [3,1]. A detailed survey of criteria to identify software components can be found in [20]. Other works have been developed for the identification of aspects in requirements specification such as [4,27,29,45,47] and [53]. An extended survey of these can be found in [7]. The identification of aspects is one of the new challenges that have emerged as a consequence of introducing aspects in software architectures. However, there is not a unique standard guided process that is used to identify architectural elements and the aspects that crosscut them. In PRISMA, the detection of those architectural elements and aspects is performed from the requirements document in an intuitive way.

PRISMA takes advantage of the notion of aspect from the beginning of the system definition by specifying the aspects that are found in the requirements specification. These aspects are reusable and can be used during the rest of the development process as well as in the maintenance process. However, other approaches such as AspectLEDA require having an initial architectural specification of the system in order to introduce aspects. In this way, aspects can only emerge as new requirements of the system that are hooked to the base architecture, i.e., aspects are only used for evolution during the maintenance process.

In the PRISMA approach, the complete view of the software architecture is not lost because of the use of aspects, that is, the use of the view notion is not required to define aspects in software architectures as in AVA. We consider that the use of architectural views from an aspect-oriented point of view is an additional mechanism for analyzing software architectural features. PRISMA can obtain different architectural views by means of the *import* relationships that aspects have from architectural elements (see Fig. 6). As a result, a view of an aspect consists of the set of architectural elements that import it. For example, we can have an aspect-oriented view of the software architecture by considering all the architectural elements that import a specific aspect. In addition, we can have another view by considering all the architectural elements that import a kind of aspect concern; for example, analyze the view of all the architectural elements that import a safety concern.

At the configuration level, when the architectural elements are instantiated and a specific configuration is defined by connecting instances, the structural and architectural view of the system is not lost due to the fact that our components are connected by means of connectors instead of aspects as in CAM/DAOP. CAM/DAOP introduces aspects as connectors and the approaches of AspectLEDA and Jiazzi implement weavings as connectors in order to coordinate their ''component aspects''. This way of introducing AO concepts in software architecture is influenced by the fact that these approaches are based on ADLs that do not originally provide the notion of connector and simulate it by aspects. Another approach that does not provides the notion of connector is JAC, but furthermore it does not try to simulate the concept, it introduces the weaving process inside its ''component aspects'' losing the reusability of aspects.

However, as Shaw presents in her work [44], the specification of software systems with complex coordination protocols is too difficult without the connector architectural element. This is because the connector provides the separation of component interactions thereby achieving a higher level of abstraction, modularity, and a greater architectural view of the system. For this reason, PRISMA connectors are first-class citizens of the ADL. In addition, the notion of connectors provides PRISMA with a mechanism to focus on all the aspects that are needed to coordinate a set of components on an entity. For example, in the *TeachMover* case study, the connector needs a coordination aspect to synchronize the components as well as a safety aspect to check whether the movements are correct. Here, the connector uses the safety aspect as a means to coordinate components. As a result, connectors are preserved in our ADL and so, our approach can follow the notion of contract of Andrade and Fiadeiro [2]. This notion specifies the dynamic business rules of the system in connectors in order to obtain more maintainable architectures.

It is also important to preserve the fact that ADLs should be formal because it is the only way to analyze and reason about the obtained software architectures. For this reason, PRISMA AOADL is a formal language that allows us to validate properties and generate code. However, we lose the advantages of using an extended lan-

guage such as XML or Java, as in the CAM/DAOP and FuseJ proposals, respectively. This disadvantage is overcome thanks to the fact that XML specifications or Java can be generated from our ADL. In this way, our ADL can be interoperable with other frameworks since the ADL XML specifications can be used.

Finally, another approach that should be mentioned besides the ones presented in [9] is TranSAT [5]. Transat is an approach for managing the evolution of software architecture specifications using aspect-oriented programming principles. The approach starts from a core architectural model that either needs to be extended during its development process or needs to be evolved during its maintenance process. The mechanisms of extension and evolution are provided using AOP techniques. As a result, this approach incrementally obtains a complete software architecture with business and technical concerns from a business software architecture. Their models are hierarchical component models that define software architecture by means of composition relationships. The new technical concerns are modelled as components that refine the original component specification. As a result, the composition relationships allow the generation of other components that include these aspects without losing the black box view of components. The main advantages of this approach is that it is based on ADL for defining the core architectural model, the way that evolution is supported, the reusability preservation and the independence from a technological. However, a great limitation of this approach is the constraint of starting the development from a core architectural model without considering concerns from the beginning. Also, the fact that concerns are only technical and not more generic is another drawback. As a result, aspects in this approach are not introduced as a new concept for modelling software architectures. Software architectures are defined using a pure compositional ADL, and aspects only appear as an extension or evolution mechanism of software architectures.

In summary, PRISMA is a new contribution in the area which offers some advantages and which is novel in the way that it integrates aspects in software architectures. PRISMA combines AOSD and software architecture in a way that preserves the concepts of both approaches. In addition, its methodology is also a new contribution due to the fact that it applies the MDSD approach to aspect-oriented software architecture in order to automatically generate code from models.

5. The PRISMA software development methodology

The contribution of this paper is the methodology that PRISMA proposes in order to develop complex systems. The PRISMA methodology can be divided into three stages: detection of architectural elements and aspects, architecture modelling, and code generation and execution. These stages are applied by the analyst of the software system in an iterative and an incremental way depending on his/her needs.

5.1. 1st Stage: detection of architectural elements and aspects

The first tasks for developing software architectures are to identify which architectural elements make up the architecture and to detect the aspects that crosscut the software architecture. In PRISMA, the detection of those architectural elements and aspects is performed from the requirements document in an intuitive way.

The running example that we are going to use to illustrate our methodology is the *TeachMover* tele-operation system [51] (see Fig. 4). The *TeachMover* robot is a robotic arm that is frequently used to teach the fundamentals of robotics. This robot was specially designed for the purpose of simulating the behaviour of large and heavy industrial robots. The computational units (components) and the coordination units (connectors), which allow the architecture to synchronize components and permit the communication among them, can be identified from the manual of the *TeachMover* robot. In addition, the concerns that crosscut the software architecture can be detected. The mechanisms and criteria to detect concerns, components and connectors are out of the scope of this paper, so we will assume that these elements have already been detected.

5.1.1. Identification of architectural elements

The manual of the *TeachMover* shows that the robot is composed of a set of components that represent a set of joints, a set of connectors that coordinate the joints, and a set of complex components that allows the composition of joints to form a robot. These joints permit the robot's movements. They are the following: *Base*, *Shoulder*, *Elbow* and *Wrist*. In addition, it has a *Tool* to perform different tasks. In this case study, the tool is a gripper, whose open and close actions allow the robot to pick up and deposit objects (see Fig. 4). These joints are complex component types called SUCs (Simple Unit Controllers) in the tele-operation domain. They are complex because they are composed of simple components that allow the movements of the SUC: the *Actuator*, the *Sensor*, and the *Connector*

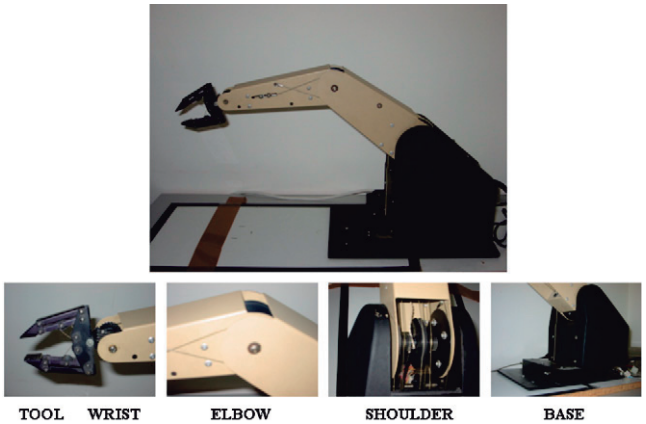


Fig. 4. The *TeachMover* Robot.

that coordinates them (see Fig. 5). *Actuators* and *Sensors* are basic components, which communicated with the hardware joints of the robot. The functionality of the Actuators and Sensors are the following:

- *Actuator*: An actuator sends commands to a joint of the robot. These commands are performed by the joint or the tool.
- *Sensor*: A sensor reads the results of the commands in order to know whether or not they have been performed successfully.

As a result, it is possible to identify that a SUC is a complex architectural element type that models the generic behaviour of a joint. However, two special SUCs have been identified in order to take into account the peculiarities of the wrist joint and the tool (see Fig. 5). The SUC, the *Wrist* and the *Tool* SUCs must be composed and coordinated in order to form the complete structure and functionality of a tele-operated robot. This composition has generated different levels of granularity. The architectural elements that participate in each level are types, and the number of instances of each one will be determined at configuration-time (Section 3.2.5 Step 5) depending on the tele-operated robot. They are the following:

Mechanism Unit Controllers (*MUCs*): This architectural element type represents the arm of the robot, which is composed of the *SUCs* and *Wrist* SUCs coordinated by means of a connector (see Fig. 5).

Robot Unit Controllers (*RUCs*): This architectural element represents the robot, which is composed of the MUCs and Tools coordinated by means of a connector (see Fig. 5).

The Architectural Model: This level represents the interactions between operators and robots throughout a connector (see Fig. 5).

From the manual of the robot the components of the robot and the different level of composition have been identified. These component types are illustrated in their corresponding level of granularity using a tree view in Fig. 5. In addition, the connectors needed to coordinate these components as well as to provide a suitable behaviour of the architecture have been identified. Connectors are necessary because PRISMA introduces connectors as first order citizens of architectural models. As a result, PRISMA avoids the dependencies among components that ADLs without connectors encounter by specifying coordination rules inside components [44].

Both architectural elements and the services that must be interchanged among them can be identified during this stage. These services are requested and/or provided by the architectural elements. Since a PRISMA interface is a set of services that is provided and/or requested by means of the ports of architectural elements, the identification of these services implies the identification of the interfaces. The ports of architectural elements and the interconnections among these ports can also be detected taking into account the identified interfaces.

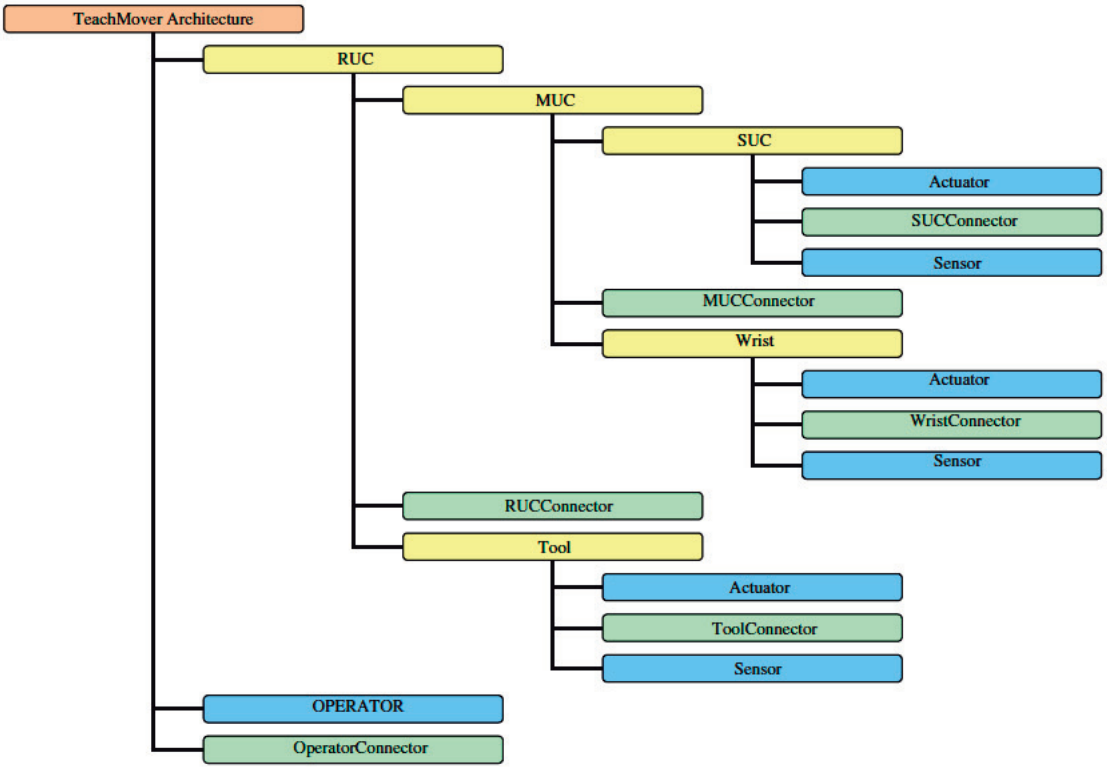


Fig. 5. Identification of architectural elements.

5.1.2. Identification of crosscutting-concerns

The concerns that crosscut the software system must be identified from the requirements specification in order to modularize them into reusable entities called aspects. In the case of the *TeachMover*, the crosscutting-concerns that we have identified in the case study are the following:

- *Functional*: The purpose of the *TeachMover* software system is to move the robot. The robot has a motor to accurately perform movements by half-steps. A half-step is an angular advance that is produced by a stimulating impulse. In the case of the *TeachMover*, the movements can be requested using half-steps or inverse cinematics (moving to a specific point in space). This functionality, together with the gripper functionalities allows the robot to move objects from an initial position to a final one. The movements of the robot are ordered by an operator from a computer.
- *Safety*: Safety directives are necessary for monitoring the *TeachMover* movements in order to make sure that the movements are safe for the robot, the operator, and the environment that surrounds them.
- *Coordination*: The inner behaviour of joints (SUCs) and the movements in which more than one joint has to be moved must be coordinated. In addition, the requests of the operator and the performance of the movements must be synchronized.

5.2. 2nd Stage: architecture modelling

Once the interfaces, aspects, architectural elements and their ports have been identified, the skeleton of the architectural elements and the aspects can be defined. The analyst is then ready to start the modelling process of the software architecture. The modelling stage of our methodology is the main contribution of this paper; that is, how can an aspect-oriented software architecture be modelled. This stage can be divided into five modelling steps: Interfaces, Aspects, Simple Architectural Elements, Complex Architectural Elements, and Configuration (see Fig. 6).

It is important to keep in mind that the enumeration of these steps is not a restrictive order. The enumeration simply indicates the dependencies between the different concepts that arise when the architectural model is being modelled. These dependencies are the following:

To configure an architectural model, the concepts that are instantiated during the configuration process must have been previously defined.

To completely define a complex architectural element, the architectural elements that it consists of must have been previously defined.

To completely define an architectural element, the aspects that it imports and the interfaces that their ports use must have been previously defined.

To completely define an aspect that uses interfaces, the interfaces must have been previously defined.

Even though the order of these steps can be different, it should be followed in order to completely define an architectural model. In other words, it does not mean that partial descriptions of the architectural elements, aspects or architectural models cannot be performed during the development process. The analyst can start the modelling process from either steps 1, 2, 3, or 4, obtaining partial solutions of the model, and can go backward or forward depending on his/her needs.

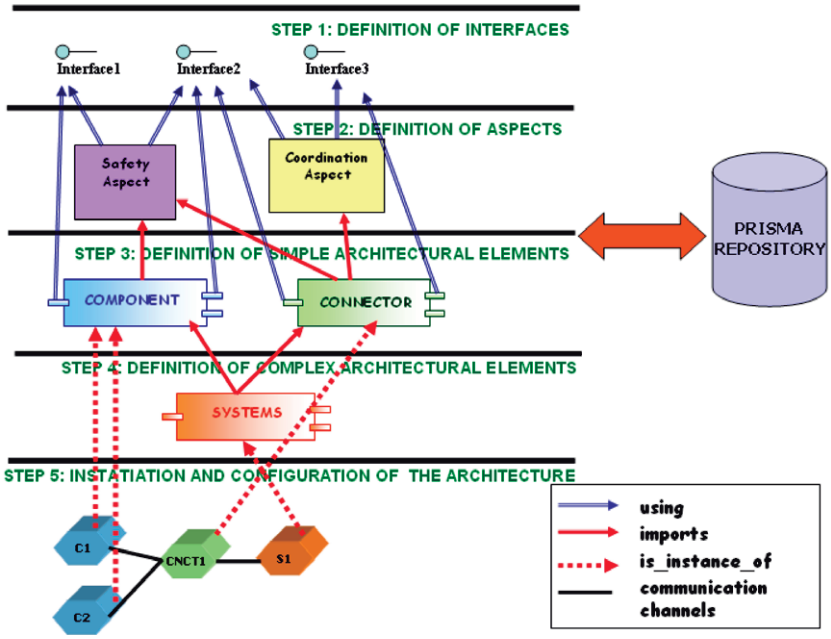


Fig. 6. The methodology of the PRISMA approach.

5.2.1. Step 1: Interfaces

Interfaces are specified in step 1 of the PRISMA architecture modelling stage. This is due to the fact that it is not necessary to previously define other elements of the model. Interfaces are stored in a PRISMA repository for reuse (see step 1, Fig. 6).

The specification of the interfaces identified in the stage presented in 3.1.1. consists of describing the interface services and their signatures. The signature of a service specifies its name and parameters. The data type and the kind (input/output) of parameters are also declared. An example of an interface is shown in Fig. 7.

5.2.2. Step 2: Aspects

As it has been previously mentioned, a PRISMA aspect encapsulates a concern that crosscuts the architectural elements of software architectures. This aspect semantics is different from any of the architectural terms in the software architecture discipline that currently exist. The notion of aspect has an entity of its own, and aspects are first-order citizens of the PRISMA AOADL.

Aspects are defined in step 2 of the PRISMA architecture modelling stage using the combination of two formalisms: a modal logic of actions [46] and a dialect of the polyadic π-calculus [26]. π-calculus is used to specify and formalize the processes of the PRISMA model, and the modal logic of actions is used to formalize how the execution of these processes affects the state of aspects. The kind of aspects and the number of each one depends on the software system. Aspects are reusable entities that define a specific behaviour of a *crosscutting-concern* and are, therefore, stored in a PRISMA repository (see step 2, Fig. 6). As a result, not only can aspects be used more than once in a software architecture description, they can also be reused in different software architectures.

There are aspects that specify the semantics of the services that are published by an interface (public services), there are aspects that specify the semantics of services that are not published by any interface (private services), and there are aspects that specify the semantics of both, public services and private services. The aspects that specify the semantics of public services must be defined after the interface of their services has been defined. This is why interfaces are defined in step 1 and aspects are defined in the step 2 of the methodology. However, this does not constrain the specification order. Either the interface is defined before the aspect, or the services are initially defined as private services of the aspect and are then changed to publish services by means of an interface. Furthermore, when the needed interfaces are reused from the repository, step 1 is not necessary.

The aspects are defined by taking into account the crosscutting-concerns identified in stage 3.1.1. The number of aspects for the same concern is decided by the analyst, taking into account criteria such as reusability and understanding. Depending on the analyst's criteria, he/she will define one aspect for a concern or several aspects for the same concern. For example, for the safety concern that crosscuts the software system, two safety aspects can be defined. Each of these aspects has a different safety behaviour. In this paper, due to space limitations, we are going to focus on the specification of a *safety* and a *coordination* aspect of the case study.

5.2.2.1. The safety aspect. The concerns (safety, coordination, functionality, distribution, etc.) that an aspect can specify in PRISMA is not constrained because the concerns vary depending on the system and the domain that is being modelled. A keyword is used to establish the concern of the aspect that is being specified. For example, in this case, the *Safety* word establishes that every property, service, or behaviour that is specified in the aspect is related to the safety concern (see Fig. 8). This keyword is a property of a PRISMA aspect called *concern*, whose value is provided when a specific aspect is defined. There is no predefined list of keywords; the value is introduced by the analyst when he/she starts the aspect modelling task.

The concern of an aspect and its name are detailed at the head of the aspect. Several constant attributes are declared in the body in order to store the information of the minimum and maximum values that have to be taken into account to preserve the safety of the *TeachMover* (see Section 1, Fig. 8). *Begin* and *end* services start and end the execution of an aspect, respectively (see Sections 2 and 5, Fig. 8). Also, several services to preserve the safety of the system are defined in the *SMotion* aspect using dynamic logic. Some of them are specified in Section 3 of Fig. 8. Due to space limitations, only the complete specification of the service *check* is presented. This service ensures that the requested movement is safe by determining whether it is between the minimum and maximum degree (see Section 4, Fig. 8). In addition, transactional operations are defined to execute a set of services atomically (see Section 6, Fig. 8). Finally, the protocol defines the process of execution of the aspect services using a dialect of π-calculus (see Section 7, Fig. 8).

```
Interface ISUC

  moveJoint(input NewHalfSteps: integer, input Speed: integer);
  cinematicsMoveJoint(input NewAngle: integer, input Speed: integer);
  stop();
  moveOk(output success: boolean);

End_Interface ISUC;
```

Fig. 7. The *ISUC* interface.

```
Safety Aspect SMotion
  Attributes
1  Constant
    minimum, maximum, minRoll, maxRoll, minPitch, maximumPitch: integer, NOT NULL; … …

  Services
2  begin( input InitialMinimum: integer, input InitialMaximum: integer, … …);
    Valuations
    [begin (InitialMinimum, InitialMaximum)]
      minimum := InitialMinimum,
      maximum := InitialMaximum;

3  in checkdistance(input NewX: integer, input NewY: integer, input NewZ: integer,
      output Secure: boolean); … …

   in check(input Degrees: integer,
            output Secure: boolean);
4    Valuations
     {(Degrees >= minimum) and (Degrees <= maximum)}
      [check(Degrees, Secure)]
       Secure := true;

     {(Degrees < minimum) or (Degrees > maximum)}
      [check(Degrees, Secure)]
      Secure := false;
    … …
5   end;

6  Transaction
    DANGEROUSCHECKING(input Degrees: integer, input CurrentSpeed: integer,
     output Secure: boolean): … …  ;… …
7  Protocol
    SMOTION = begin.CHECKING;
    CHECKING = check (Degrees, Secure).CHECKING + checkDistance(NewX, NewY, NewZ,
               Secure). CHECKING  + …  … + DANGEROUSCHECKING(Degrees, Secure).CHECKING
               + end;
   end Aspect SMotion;
```

Fig. 8. The safety aspect *SMotion*.

5.2.2.2. The coordination aspect. The coordination aspect defines the interactions needed to coordinate the sending of movements to the robot and the robot's answers. These answers notify whether or not the movements have been satisfactorily performed. In this case, the aspect uses the services of several interfaces. This is detailed at the head of the aspect (see Fig. 9), that means, that these services are public. As a result, these interfaces have been previously defined.

Since the request of a movement to the robot does not guarantee that it will be satisfactorily performed, the coordination aspect must coordinate the position of the joint that is synchronizing the movement request (see Sections 2 and 3, Fig. 9) with the movement notification of the robot (*moveOk* service, see Section 4, Fig. 9). At this point, the aspect changes the value of the robot position, i.e., the *half-step* attribute (see Section 1, Fig. 9).

The protocol of the *CProccessSUC* aspect coordinates the requested movements and the notification movements of the robot. It is composed of three processes *CProcessSUC*, *MOTION*, and *ANSWER* (see Section 5, Fig. 9). The *CProcessSUC* process starts the execution of the aspect (*begin*), initializes the attributes that need a value (*Not Null*), and starts the *MOTION* process (see Section 6, Fig. 9). This process either receives a movement request or ends the aspect execution (see Section 7, Fig. 9). Finally, this *MOTION* process continues with the *ANSWER* process, in which the coordination process waits for the sensor's answer and notifies of the failure or success of the movement (see Section 8, Fig. 9).

5.2.3. Step 3: Simple architectural elements

The definition of simple architectural elements is performed in step 3 of the PRISMA architecture modelling stage. The aspects that are defined in step 2 are used to completely define these architectural elements. An architectural element imports the aspects that define the concerns that it requires. For this reason, aspects must be defined before architectural elements.

The same aspect is imported by each architectural element that needs to take into account the same behaviour of this concern (crosscutting-concerns). As a result, an aspect can be imported by one or more architectural elements (see steps 2 and 3 of Fig. 6). It is important to note that the changes performed in an aspect also affect every architectural element that imports this aspect.

From the architectural element point of view, each architectural element is formed by a set of aspects. For example, the connector in Fig. 6 is made up of a safety aspect and a coordination aspect.

```
Coordination Aspect CProcessSUC using IMotionJoint, IRead, ISUC
  Attributes
   Variable
1   halfSteps: integer, NOT NULL;
    tempHalfSteps: integer;

   Derived
    angle: integer, Derivation FtransHalfstepsToAngle(halfSteps);

  Services
   begin (input InitialHalfSteps: integer);
      Valuations
        [begin (InitialHalfSteps)]
         halfSteps := InitialHalfsteps,

2   in/out movejoint(input NewHalfsteps: integer, input Speed: integer);
       Valuations
         [in movejoint (NewHalfsteps, Speed)]
          tempHalfSteps := NewHalfsteps;

3  in cinematicsmovejoint( input NewAngle: integer, input Speed:integer);
       Valuations
        [in cinematicsmovejoint(NewAngle, Speed)]
        tempHalfSteps := FtransAngleToHalfsteps(NewAngle);

4  in/out moveok(output Success: boolean);
        Valuations
         {Success=1}
         [in moveok(Success)]
         halfSteps:= halfSteps + tempHalSteps;

    end;

5  Protocol
6  CPROCESSSUC = begin(InitialHalfStep).MOTION
7  MOTION =
          (ISUC.movejoint?(NewHalfsteps, Speed)
          → IMotionJoint.movejoint!(NewHalfsteps, Speed).ANSWER)
          +
          (ISUC.cinematicsmovejoint?(NewAngle, Speed)
          → IMotionJoint.movejoint!(FTransAngleToHalfSteps(NewAngle, Speed).ANSWER)
          +
           end;
8  ANSWER= IRead.moveok?(Success) → ISUC.moveok!(Success).MOTION
End_Aspect CProcessSUC;
```

Fig. 9. The coordination aspect *CProcessSUC*.

In PRISMA, weavings weave the different aspects that form an architectural element and they are encapsulated inside the architectural element. The temporal order of the weaving process is described by temporal operations called weaving operators. These operators can be extended in future work. Initially, we have the *after*, *before*, *around*, *afterif*, *beforeif and insteadif* operators whose semantics are defined in the following:

A weaving that relates service s1 of aspect A1 and service s2 of aspect A2 can be specified using the following operators:

- A2.s2 **after** A1.s1: A2.s2 is executed after A1.s1.
- A2.s2 **before** A1.s1: A2.s2 is executed before A1.s1.
- A2.s2 **instead** A1.s1: A2.s2 is executed in place of A1.s1.
- A2.s2 **afterif** (**Boolean condition**) A1.s1: A2.s2 is executed after A1.s1 if the condition is satisfied.
- A2.s2 **beforeif** (**Boolean condition**) A1.s1: if the condition is satisfied, first A2.s2 is executed and then A1.s1is executed; otherwise, only A2.s2 is executed.
- A2.s2 **insteadif** (**Boolean condition**) A1.s1: A2.s2 is executed in place of A1.s1 if the condition is satisfied.

As a conclusion, a PRISMA simple architectural element is specified by a set of aspects, the weavings of aspects, and a set of ports that have an associated interface. Ports represent the points of interaction of architectural elements. The architectural elements that are defined in this step are types that are reusable by different software architectures because they are stored in the PRISMA repository. The storage of PRISMA architectural elements implies the storage of the aspects that they import. In addition, the reusability of the aspect can be due to the fact that it is reused in many architectural elements and also when the architectural element that imports the aspect is reused.

The simplest components that are found in the *Teach-Mover* system are actuators and sensors:

- *Actuator*: An actuator has two ports. To describe its functional behaviour, it imports the *FActuator* aspect that has been previously stored in the repository (see Fig. 10).




```
Component Actuator
 Ports
  ControlAct: IMotionJoint;
  HW: IMotionJoint;
 End_Ports;

 Functional Aspect Import FActuator;
End_Component Actuator;
```

Fig. 10. The Component *Actuator*.

- *Sensor*: A sensor has two ports. To describe its functional behaviour, it imports the *FSensor* aspect that has been previously stored in the repository (see Fig. 11).

In addition, the actuators and sensors for the tool and the wrist joints are defined as new architectural elements. They have the same functionality as the ones presented in Figs. 10 and 11. However, the signature of the services that they offer is different, that is, the number of parameters and their types is different.

Weavings are only specified when they are necessary to weave the execution of two services from different aspects. An example of weavings appears in the *SUCConnector* architectural element where the joint is moved only after the connector is sure that the movement is safe (see the weavings section of the *SUCConnector* in Fig. 12).

The invocation of the *movejoint* service (the second service of the weaving) of the *CProcessSUC* aspect triggers the execution of a weaving. When a weaving is specified, the weaving operator is chosen from the point of view of the service that triggers the weaving; that is, depending on whether the service needs the execution of a service *before*, *after*, or *instead* of it. As a result, the weaving of the *SUCConnector* (see Fig. 12) means that the *check* service of the *SMotion* safety aspect will be executed before the *movejoint* service of the *CProcessSUC* coordination aspect. The condition also establishes that the execution of the *movejoint* service must only be performed if the *Secure* parameter of the *check* service returns *true*.

The connectors that coordinate the actuators and the sensors of the tool and the wrist joints are also defined as new architectural elements. *SUCconnector*, *WristSUCconnector*, and *ToolSUCConnector* are architectural elements stored in the PRISMA repository. These three architectural elements import the same coordination (*CProcessSUC*) and safety (*SMotion*) aspects (crosscutting-concerns); however their behaviour is different because they have different weavings. This is a clear example of the reuse of aspects inside the same architectural model thanks to the fact that the definition of weavings outside aspects and inside architectural elements, and an example of how these aspects crosscut the behaviour of several architectural elements.

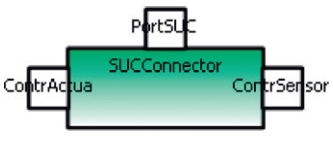
The *Actuator*, *Sensor*, and their connector *SUCConnector* are reusable architectural elements that can be used several times in the *TeachMover* architectural model or in another architecture specification such as the EFTCoR robot. In the case of the *TeachMover*, they have been specified in the simplest way. However, in the EFTCoR, they also have a distribution aspect in order to provide distrib-




```
Component Sensor
 Ports
  ControlSen: IRead;
  HW: IRead;
 End_Ports;

 Functional Aspect Import FSensor;
End_Component Sensor;
```

Fig. 11. The Component *Sensor*.




```
Connector SUCConnector
 Ports
  portSUC: ISUC;
  ContrActua: IMotionJoint;
  ContrSensor:IRead;
 End_Port;

 Coordination Aspect Import CProcessSUC;
 Safety Aspect Import Smotion;

 Weavings
  SMotion.check(FTransHalfstepsToAngle
                (NewHalfsteps), Secure)
  beforeIf (Secure = true)
  CProcessSUC.movejoint(NewHalfsteps, Speed);
   ……
  End_Weavings;
End_Connector SUCconnector;
```

Fig. 12. The Connector *SUCConnector*.

uted behaviour to these components. As a result, we have reused the *Actuator*, *Sensor* and *SUCConnector* architectural elements of the *TeachMover* in the architecture of the EFTCoR. We have modified them by adding a distribution aspect, a pair of weavings between the functional and distribution aspects of the *Actuator* and *Sensor*, and another pair of weavings between the coordination and distribution aspects of the *SUCConnector*. This is an example of reusability of defined components in other architectural models using the PRISMA repository. This reusability avoids having to start from scratch to build a new architectural model. The analyst can reuse the architectural elements in their original way or can reuse them and then introduce the changes that the new software system requires, just as we did with the actuator and sensor components of the EFTCoR.

An actuator and a sensor must be coordinated through a connector in order to separate their computations from their interactions [44]. The *SUCConnector* imports a coordination and a safety aspect in order to define its behaviour (see Fig. 12).

5.2.4. Step 4: Complex architectural elements

PRISMA complex components are called systems. Systems are defined in the 4 step of the PRISMA architecture modelling stage.

A PRISMA system is a complex component that imports a set of connectors, components, and other systems that must be correctly attached. A system is defined by using two kinds of communication channels: attachments and bindings.

To completely specify a system, the architectural elements that the system is composed of should be previously defined. In addition, the communication channels that permit the communication among them are defined. It is important to emphasize that the attachments are only defined if the system includes components and connectors that must be coordinated.

Systems are defined as patterns or architectural styles [18] that can be reused in any software architecture whenever they are needed. For this reason they are stored in the PRISMA repository.

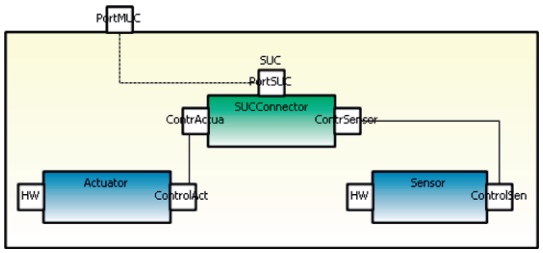
It is important to note that any changes that occur in aspects affect the architectural elements that import them and, consequently, affect the systems that import these architectural elements. Moreover, the changes that occur in architectural elements that are imported by a system also affect the system (see step 3, Fig. 6).

In the case of the *TeachMover*, there are several systems, at different levels of granularity. These systems are guided by the skeleton identified in the stage 3.1.1 (see Fig. 5). The SUC (Simple Unit Controller) system is composed of an actuator, a sensor, and the connector that synchronizes them. The system specifies the architectural elements that it is composed of and the communication channels among them (see Fig. 13).

The SUC system delegates the commands that it receives to the connector in order to perform the movements between the actuator and sensor in a synchronized way. For this reason, the *SUC* system and the *SUCConnector* have a binding between their *portMUC* and *portSUC* ports, respectively. In addition, two attachments haven been defined in order to establish the communication channels between the *Actuator* and the *SUCconnector*, and the *Sensor* and the *SUCconnector*, respectively. A *ToolSUC* and a *WristSUC* are also defined with their corresponding actuators, sensors, connectors and their relationships. The SUC system is stored in the PRISMA repository so that it can be reused as an architectural pattern.

The MUC (Mechanism Unit Controller) system is the third layer of granularity of the decomposition of the *TeachMover* system. It integrates a set of SUCs and a connector. The connector coordinates the SUCs in order to achieve a common goal. Specifically, the MUC of the *TeachMover* is composed of the generic SUC, the SUC of a wrist, and a connector that synchronizes them (see Fig. 14). As a result, the MUC of the *TeachMover* is matched with the arm of the robot and specifies the behaviour required to accurately perform its movements. The MUC is also stored in the PRISMA repository.

The RUC (Robot Unit Controller) system coordinates every part of the robot and is composed of the MUC of the arm and the SUC of the tool, which are synchronized through a connector (see Fig. 15). The SUC of the tool is




```
System SUC
 Ports
  PortMUC: ISUC;
 End_Ports;

 Import Architectural_Elements
                Actuator, Sensor, SUCconnector;
 End_Architectural_Elements;

 Attachments
   SUCconnector.ContrActua ←→
   Actuator.ControlAct;
   VarSUCconnector.ContrSensor ←→
   VarSensor.ControlSen;
 End_Attachments;

 Bindings
  SUCconnector.portSUC ←→ portMUC;
 End Bindings;
```

Fig. 13. The System *SUC* (*Simple Unit Controller*).

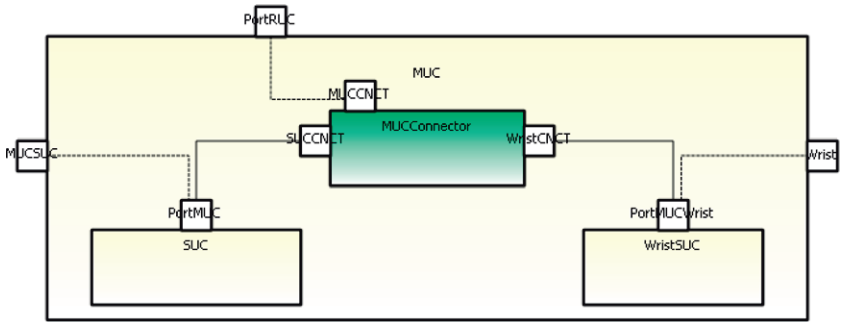


Fig. 14. The System *MUC* (*Mechanism Unit Controller*).

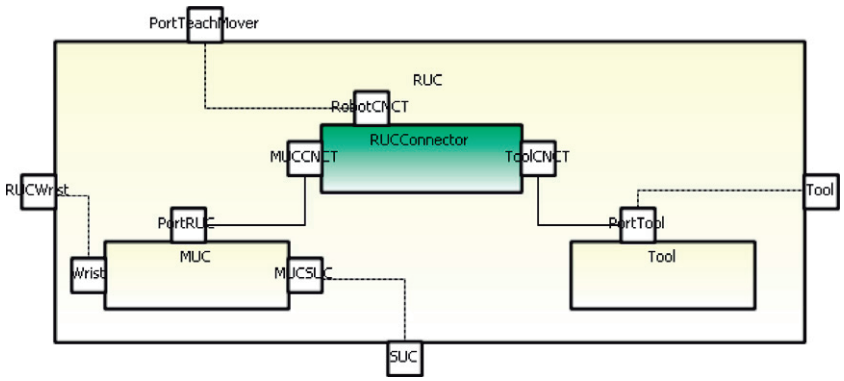


Fig. 15. The System *RUC* (*Robot Unit Controller*).

not inside the MUC system in order to easily allow changes in the tool. For example, the tool of the *TeachMover* is a gripper, but it can be changed for a paintbrush, a hosepipe, etc., in order to perform other tasks. The RUC is matched with the *TeachMover*. As a result, we obtain a software architecture that is easy to evolve.

Finally, the last layer of decomposition is composed of the operator, the robot (RUC), and the connector that coordinates them (see Fig. 16). This last level provides us the most abstract view of the software architecture, which is called the *Architectural Model*. It is important to emphasize that since the architectural model does not define a system that encapsulates it, bindings are do not need to be defined.

5.2.5. Step 5: Configuration of software architectures architectural elements

Finally, the architectural elements that have been defined in the previous steps and have been stored in the PRISMA repository are instantiated in step 5. In order to understand the step trace of the approach, it is important to take into account that instances have all the properties and behaviours of their architectural elements, and as a consequence, instances have the properties and behaviours of the aspects that their architectural elements import. As a result, when an architectural element is instantiated, the aspects that it imports are also instantiated in order to have their specific state and behaviour.

A specific software architecture is defined by connecting a set of instances of components, systems, and connectors with each other (see Fig. 6). For this reason, the instantiation of the architectural elements of a model and the definition of attachment and binding relationships among instances is necessary to obtain an executable architectural model.

To obtain the software architecture of the *TeachMover*, the *SUC* has been instantiated three times in order to obtain the base, shoulder, and elbow joints of the robot (see Fig. 13). The gripper and the wrist are instances of the *ToolSUC* and *WristSUC*, respectively. As a result of these instantiations, the instantiation of the MUC generates the configuration that appears in Fig. 17.

The RUC and the architectural model of the *TeachMover* are instantiated generating one instance for each one of them. This instantiation constitutes the last step of the modelling stage. As a result of this step, the software architecture of the *TeachMover* is completely defined.



Fig. 16. The architectural model of the *TeachMover*.

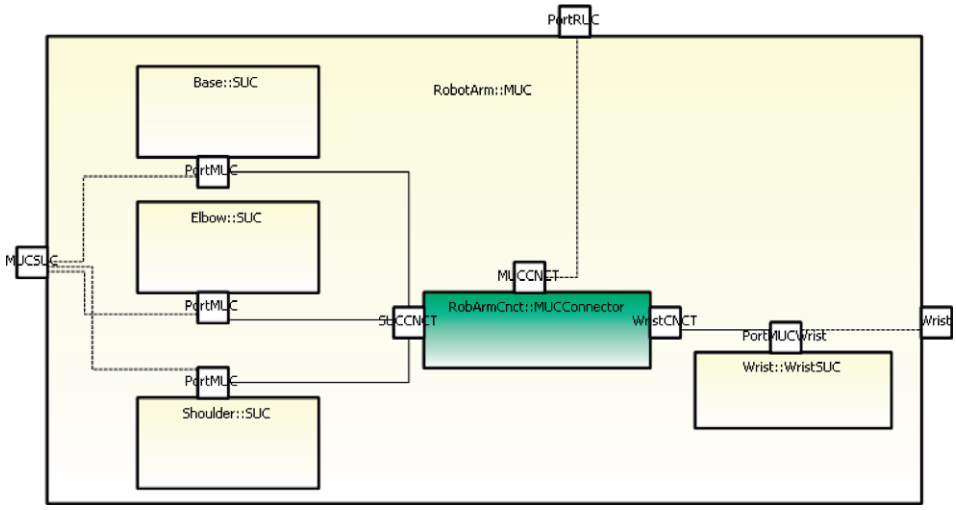


Fig. 17. The configuration *MUC* for the *TeachMover*.

5.2.6. Summary

The TeachMover example illustrates that the PRISMA approach allows the development of aspect-oriented software architectures as if aspects and architectural elements were building blocks. You can work with them in different ways to obtain different results. This flexibility and facility for working is achieved thanks to the fact that aspects and components are independent entities that can be imported in different entities of the same software architecture or in different ones.

The introduction of aspects in a separate entity allows for the *TeachMover* software architecture to separate the safety concern in the *SMotion* aspect. This aspect is only defined once and it is imported by the *SUCconnector*, *WristSUCconnector*, *ToolSUCconnector*, *MUCconnector and RUCconnector* connectors by referencing this aspect. As a consequence, the aspect is used by the instances of these connectors. For example, the *TeachMover* instances of the *SUCconnector* are the *BaseSUCconnector, Shoulder-SUCConnector* and *ElbowSUCConnector*. As a result of considering aspects to model the *TeachMover*, tangled code is avoided by separating coordination properties from safety properties. In addition, this separation of concerns can improve the maintenance process of the system due to the fact that the introduction of a change in the safety properties of the *TeachMover* only requires modifying the *SMotion* Aspect and, consequently, all the changes are propagated to the connectors and their instances.

The PRISMA repository also promotes the reuse of architectural elements and aspects in different software architectures. They can be used as they are stored in the repository or can be adapted to the features of the new architecture that is being modelled. For example, most of the architectural elements of the *TeachMover* were reused in the EFTCoR software architecture. The specific properties of the EFTCoR were easily adapted; for example, some of the robot pieces of the EFTCoR are distributed. This property was introduced by defining a distribution aspect and importing it from the reused architectural elements of the *TeachMover* that represent these pieces. It must be noted that reuse is limited for software architectures of different domains, but the PRISMA repository could be a great contribution for reuse in product families such as the tele-operation family. In addition, a large repository with good searching mechanisms could provide excellent support for the development and maintenance of software.

5.3. 3rd Stage: code generation and execution

Most ADLs are not expressive enough to define the complete behaviour of a software system, since they only describe the structure of systems. In PRISMA, aspects provide the mechanism to completely define not only the functional behaviour of the system, but also its non-functional requirements. The PRISMA AOADL is a formal language based on Modal Logic of Actions and Polyadic π-calculus that allows us: (1) not to worry about implementation details, (2) to verify aspect-oriented architectural models, (3) to generate code from architectural specifications and (4) to maintain the traceability from the software architecture to its implementation. Even though these are definitively advantageous characteristics, the use of a formal language is really difficult. For this reason, we provide a framework to model PRISMA software architectures using an intuitive and friendly graphical AOADL (see Fig. 18). In addition, the framework gives the analyst support to follow the PRISMA methodology in order to take maximum advantage of the PRISMA approach.

The framework has been developed using DSL tools [10], which has provided us a simple way of following the MDD paradigm by developing the model compiler and the modelling tools of the PRISMA framework. The model compiler has been developed using the template files that DSL tools provides for the code generation. We have programmed our PRISMA templates to be able to generate C# code and the textual AOADL from software architec-

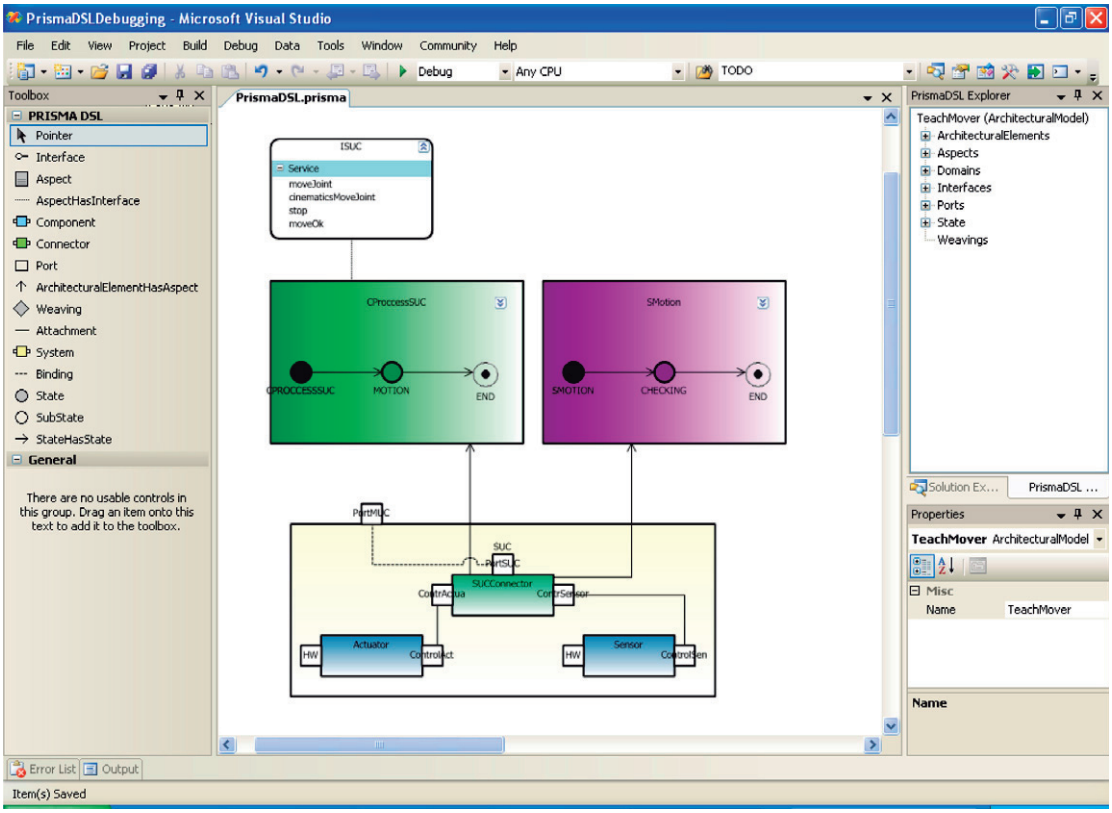


Fig. 18. The PRISMA modelling tool.

tures that have been modelled graphically using the PRISMA modelling tools. Fig. 19 shows the equivalences between the graphical representation of an interface, and the AOADL and C# code that have been generated by executing the code generation templates of the PRISMA compiler.

In addition, Fig. 20 shows the complete support for developing executable PRISMA architectural models following the MDD paradigm and using the DSL tools. The complete PRISMA framework is called PRISMA CASE. The Figure shows the architecture of the framework and how the parts that compose the PRISMA framework allow us to automatically generate executable C# code from architectural models that have been modelled using the modelling tool. This generation is possible thanks to the code generation templates (model compiler), which isolate the specification from the source code preserving their independence. Until now, PRISMA CASE generates PRISMA aspect-oriented C# code that is executable in .NET framework thanks to our PRISMANET middleware, which gives support to aspect execution over .NET technology [38].

The PRISMA methodology is put into practice using the PRISMA CASE, except for the 1st Stage that the identification of architectural elements and aspects is made by hand. The 2nd Stage is developed using the PRISMA Type Modelling Tool and the PRISMA Configuration Modelling Tool. The STEPS 1 4 are developed using the PRISMA Type Modelling Tool, where all the reusable types (interfaces, aspects, simple architectural elements and complex architectural elements) and the architectural model of the software system are modelled in a graphical way by drawing and dropping the PRISMA modelling primitives. The STEP 5 is developed using the PRISMA Configuration Modelling Tool, where the configuration of the initial architecture of a specific system is modelled by instantiating the types and the architectural model that has been defined in previous STEPS. Finally, the 3rd Stage is developed using the PRISMA Model Compiler, to generate the code of the types and its instances. The code of the types is generated by executing the PRISMA Model Compiler from the PRISMA Type Modelling Tool, and the code of the instances is generated by executing the PRISMA Model Compiler from the PRISMA Configuration Modelling Tool. Next, the execution of the generated code joint the PRISMANET middleware can be launched from the PRISMA Configuration Modelling Tool. Once the aspect-oriented software architecture is executed the user can interact with it using the PRISMA Generic GUI, which allows the user to execute services, query the value of attributes and validate the correct behaviour of each of the architectural elements that compose the architecture.

6. Discussion

Although this paper has placed emphasis on the architecture modelling stage of the methodology, the PRISMA

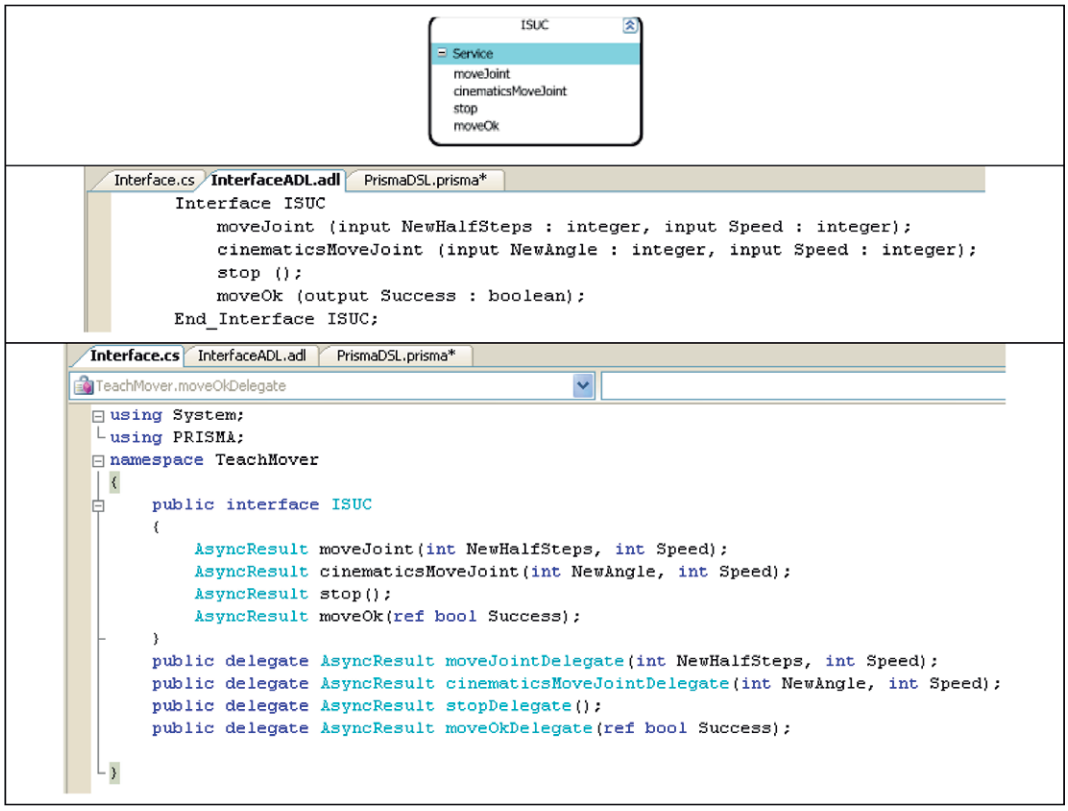


Fig. 19. The PRISMA code generation.

approach includes a model, an AOADL that is textually and graphically supported, a technological middleware (PRISMANET), and a tool (see Fig. 20). The PRISMA methodology is supported by a framework for the construction of aspect-oriented software architectures that are independent of specific technologies.

The first stage of the PRISMA methodology is the identification of the aspects and the architectural elements of a software architecture. Currently, this identification is performed in an intuitive way from the requirements document of the system. However, we are working in this stage in order to propose guidance for the user in this identification task [29,30].

The modelling stage of the PRISMA methodology is based on the PRISMA metamodel. It has been defined by integrating aspects and software architectures. In this way, software architectures can be constructed gaining the advantages of two different paradigms: the aspect-oriented paradigm (AOP) and the component-based paradigm. One of these advantages is the reusability of software at different levels of granularity. As an example, the reusability level of the SUC system (see Fig. 13) is illustrated in the following:

6.1. Analysis of the reusability of the SUC system

The reusability of the *SUC* is analyzed by taking into account the system, its architectural elements and the aspects that these architectural elements import. In addition, the reusability is going to be analyzed at the type level and at the configuration level.

Fig. 21 shows the number of times that an aspect is reused by different architectural elements and the names of the architectural elements that import the aspect. The aspects are enumerated in the first columns of the two tables. These aspects are the ones that have been used for defining the SUC system. The first table details the reusability at the type level, and the second table details the reusability at the configuration level, respectively. From these tables, it is possible to conclude that: the aspect *SMotion* is defined once and it is reused by five architectural elements at the type level and by seven architectural elements at the configuration level. The aspects *FActuator FSensor* and *CProcessSUC* have been defined once and they are reused by one architectural element at the type level and three architectural elements at the configuration level.

On the other hand, Fig. 22 shows the number of times that an architectural element type is reused at the configuration level of the architecture by means of its instantiation.

In addition to the *SUC* rates of reusability inside the *TeachMover* software architecture, the reusability of aspects and architectural element types by other software architectures, such as the EFTCoR software architecture, must be taken into account. For example, in the case of the EFTCoR we have imported every aspect and architectural elements of the TeachMover without hav-

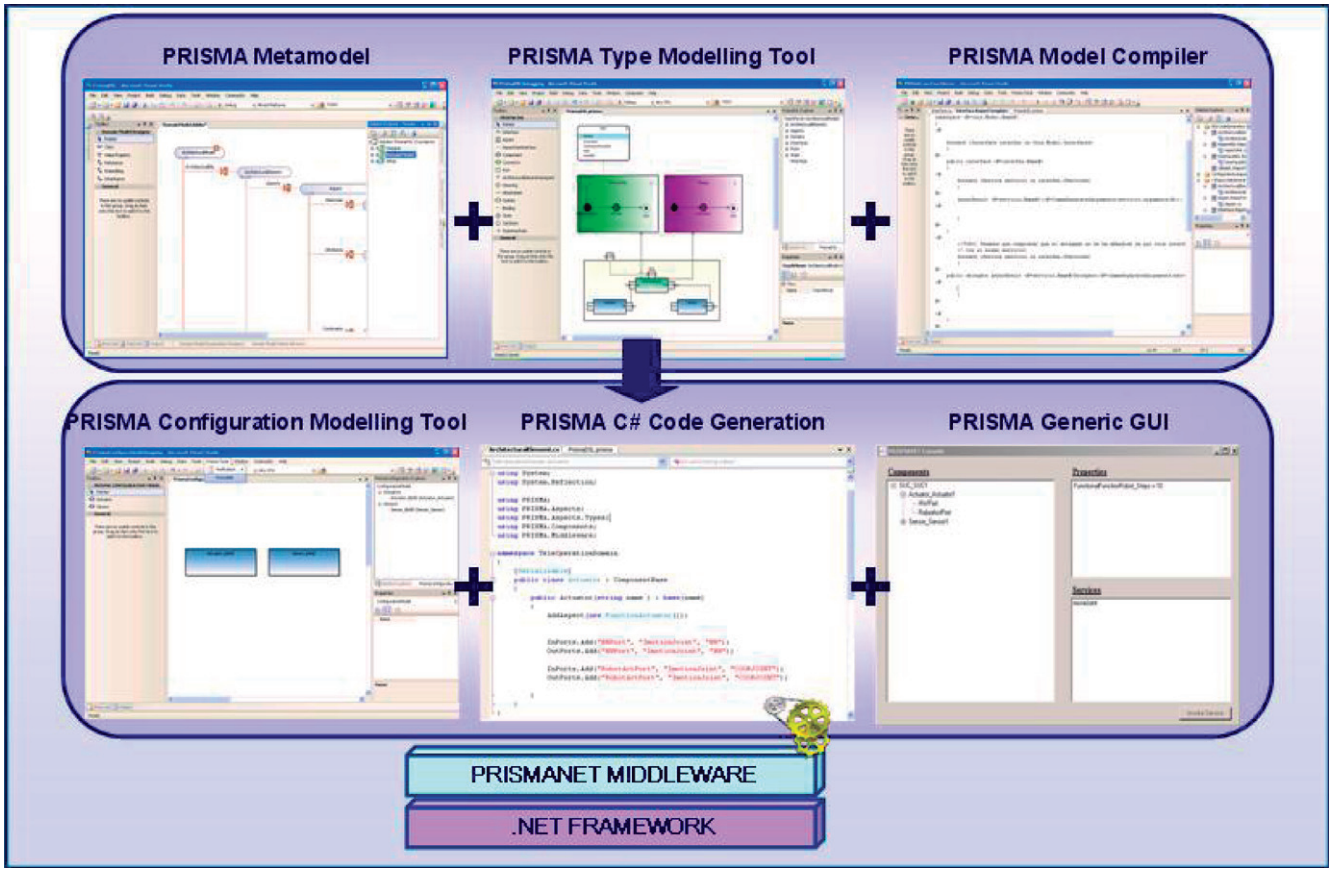


Fig. 20. The PRISMA framework.

Reusability of Aspects (Type Level)		ARCHITECTURAL ELEMENTS	
		Number	Name
ASPECTS	SMotion	5	SUCConnector, WristSUCConnector, ToolSUCConnector, MUCConnector, RUCConnector
	CProcessSUC	1	SUCConnector
	FActuator	1	Actuator
	FSensor	1	Sensor

Reusability of Aspects (Configuration Level)		ARCHITECTURAL ELEMENT INSTANCES	
		Number	Name
ASPECTS	SMotion	7	BaseCnct, ElbowCnct, ShoulderCnct, WristCnct, ToolCnct, RobotArmCnct, RobotCnct
	CProcessSUC	3	BaseCnct, ElbowCnct, ShoulderCnct
	FActuator	3	BaseAct, ElbowAct, ShoulderAct
	FSensor	3	BaseSen, ElbowSen, ShoulderSen

Fig. 21. Reusability of aspects.

ing to start the modelling process from scratch. Some of the reused aspects have been modified introducing new properties and others had not to be modified. Some of the reused architectural elements have been modified adding new aspects and/or weavings and others have been preserved in their original version.

Maintainability and evolution are also gained from integrating the aspect-oriented paradigm (AOP) and the component-based paradigm. This is due to the fact that aspects

Reusability of Architectural Elements		ARCHITECTURAL ELEMENT INSTANCES	
		Number	Name
ARCHITECTURAL ELEMENT TYPES	SUC	3	BASE,ELBOW, SHOULDER
	SUCConnector	3	BaseCnct, ElbowCnct, ShoulderCnct
	Actuator	3	BaseAct, ElbowAct, ShoulderAct
	Sensor	3	BaseSen, ElbowSen, ShoulderSen

Fig. 22. Reusability of architectural elements.

and architectural elements facilitate the task of locating where the changes must occur. For example, in the case of the TeachMover, safety properties are located in only one place, in the aspect *SMotion*. As a result, if a change in the safety properties occurs, the safety aspect is only modified instead of changing the five architectural elements that import it at the types level. But not only the number of changes is reduced, but also the time invested in locating where the change must be introduced is improved. In our case, the safety properties of the aspect have to be only searched; whereas in a non-aspect-oriented approach, the architectural elements with tangled concerns have to be searched and the location of the safety properties is more difficult. Finally, another important advantage is the reduction of code. As it was demonstrated by Kizcales et al. [23], the lines of code are reduced. Our proposal avoids the replication of code. For example, t the lines of the safety aspect SMotion were avoided to be repeated five times at the type level.

The DSL tools development environment has been chosen in order to provide a framework for the PRISMA approach. DSL tools allow us to define PRISMA metamodel, associate a graphical notation to each metamodel concept and its instantiation, and implement the C# code generation templates of PRISMA. The PRISMA framework provides mechanisms to optimize the programming, reusability, and modularity of code. Developers use the graphical notation of the PRISMA AOADL to build their software architectures using the methodology presented in this paper. After, the models are verified they can generate automatically the code. The *TeachMover* software architecture has been modelled, its code has been generated and it has been executed in the.NET platform. As a result, PRISMA has made it possible to move a robot using an aspect-oriented architecture that is executed in .NET technology (see demonstrations in [42]).

7. Conclusions and future work

In this paper, the PRISMA approach which introduces aspects in software architectures has been presented in detail. It has been designed to integrate aspects as a new concept and as first-order citizens of software architectures. PRISMA is presented as a symmetrical model because it does not distinguish a kernel entity that encapsulates functionality; functionality is also specified as an aspect. In addition, other concerns related to non-functional requirements can be weaved with other non-functional concerns. In this way, PRISMA shows that non-functional concerns can have the same relevance as functional ones.

PRISMA offers a clear approach in which both aspects and software architectures are smoothly integrated with a clear semantics. Aspects are defined independently of architectural elements and the final software architecture of the system. Architectural elements are defined by means of the importation of aspects. As a result, aspects crosscut the architectural elements of PRISMA software architectures. PRISMA is presented with a methodology, which is divided into five steps, to allow the development and maintenance of software systems in a simple and flexible way.

The precise semantics of the PRISMA language gives us the chance to verify model properties and to compile models to obtain specific executable systems. A PRISMANET middleware [38] has been developed to support PRISMA on .NET platform. We are currently developing compilers for specific infrastructures such as C#, Web Services, XML, etc.

The case study presented in this work is a suitable way to illustrate how the PRISMA approach serves to develop complex software. The example is bounded enough to be easily understood. It shows how the PRISMA approach enables the deployment of software and provides flexibility to the development of software systems.

The ideas developed in the case study have been applied to specific products of the EFTCoR and *TeachMover*. The benefits of applying PRISMA to the case study are mainly: (1) the definition of reusable concerns that crosscut the software architecture called aspects and the storage in a repository, (2) the definition of reusable architectural elements by reusing aspects, (3) the definition of complex architectural elements reusing other architectural elements, (4) building the formal software architecture of the system using a friendly graphical notation independently of technology and (5) the reduction of the development time as code is automatically generated from the graphical notation to a specific technology.

It is important to emphasize that this paper is a proof of the emergent convergence of two areas of engineering that have historically been separated; software engineering and robotics engineering. In this paper, we have applied a software engineering approach to the development of a robotic system. In this way, this development has taken advantage of the good properties provided by this software engineering approach, especially reuse of components, maintainability, evolution, etc.

As future work, we are going to introduce validation techniques in our modelling tool and to compare aspect-oriented and non-aspect-oriented applications to be able to measure the advantages that PRISMA provides. This measurement should be made taking into account different case studies and domains. In addition, we are going to extend the language to support more mechanisms for specifying architectural patterns. We want to provide support for the COTS importation in the PRISMA methodology. Finally, we are going to implement an assistant for the PRISMA repository and the code generation patterns for other technologies and languages.

Acknowledgements

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, META project TIN2006-15175-C05-01.

References

[1] B. Álvarez, P. Sánchez, J.A. Pastor, F. Ortiz, An Architectural Framework for Modeling Teleoperated Service Robots, International Journal of Information, Education and Research in Robotics and Artificial Intelligence, Cambridge University Press, 2005.
[2] L.F. Andrade, J.L. Fiadeiro, Architecture based evolution of software systems, in: M. Bernardo, P. Inverardi (Eds.), Formal Methods for Software Architectures, LNCS 2804, Springer Verlag, Bertinoro, Italy, 2003.
[3] F. Bachman, L. Bass et al., The Architecture Based Design Method, Technical Report CMU/SEI 200 TR 001, Carnegie Mellon Univer sity, USA, January (2000).
[4] E. Baniassad, S. Clarke, Finding aspects in requirements with Theme/Doc, in: Proceedings of the Workshop on Early Aspects, held with the International Conference on Aspect Oriented Software Development (AOSD), Lancaster, UK, 2004.
[5] O. Barais, E. Cariou, L. Duchien, N. Pessemier, L. Seinturier, Transat: a framework for the specification of software architecture

evolution, in: Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT), held with ECOOP, Oslo, June, 2004. <http://wcat04.unex.es/bib>.
[6] H. Bruyninckx, B. Koninckx, P. Soetens, A Software Framework for Advanced Motion Control, Department of Mechanical Engineering, K.U. Leuven, OROCOS project inside EURON, Belgium, 2002.
[7] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto, J. Bakker, B. Tekinerdogan, S. Clarke, A. Jackson, Report synthesizing state of the art in aspect oriented requirements engineering, architectures and design, Lancaster University, Lancaster, AOSD Europe Deliverable D11, AOSD Europe ULANC 9, 18 May, 2005, pp. 1 259, <http://www.comp.lancs.ac.uk:8080/c/portal/layout?p 1 id=1.94>.
[8] E. Coste Manière, R. Simmons, Architecture, the backbone of robotic system, in: Proceedings of the IEEE International Conference on Robotics & Automation, IEEE Computer Society, San Francisco, USA, April, 2000, pp. 505 513.
[9] C. Cuesta, M.P. Romay, P. De La Fuente, M. Barrio Solórzano, Architectural aspects of architectural aspects, in: Proceedings of the Second European Workshop on Software Architecture, LNCS 3527, Springer Verlag, Pisa, June 2005.
[10] Domain Specific Language (DSL) Tools, <http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx>.
[11] R. Douence, D. Le Botlan, Report Towards a Taxonomy of AOP Semantics, AOSD Europe Deliverable D11, AOSD Europe INRIA 1, INRIA, CNRS, 7 July, 2005, pp. 1 13, <http://www.comp.lancs.ac.uk:8080/c/portal/layout?p 1 id=1.94>.
[12] T. Elrald, R.E. Filman, A. Bader, Aspect oriented programming: an introduction, Communications of the ACM 44 (10) (2001).
[13] C. Fernández, J.A. Pastor, P. Sánchez, B. Álvarez, A. Iborra, Cooperative robots for hull blasting in European shiprepair industry, IEEE Robotics and Automation Magazine (RAM) (2005).
[14] E. Gamma et al., Design Patterns: Elements of Reusable Object Oriented Software, Addison Wesley, Reading, 1995.
[15] A. García, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa, Modularizing design patterns with aspects: a quantitative study, in: Proceedings of the 4th international conference on Aspect oriented software development, ISBN:1 59593 042 6, Chicago, Illinois, 2005, pp. 3 14.
[16] D. Garlan, Formal modelling and analysis of software architecture: components, connectors, and events, in: M. Bernardo, P. Inverardi (Eds.), Formal Methods for Software Architectures, LNCS 2804, Springer Verlag, Bertinoro, Italy, 2003.
[17] D. Garlan, in: J. Marciniak (Ed.), Software architecture, Wiley Encyclopedia of Software Engineering, John Wiley & Sons, 2001.
[18] D. Garlan, M. Shaw, in: V. Ambriola, G. Tortora (Eds.), An introduction to software architecture, Advances in Software Engineering and Knowledge Engineering, vol. 1, World Scientific Publishing Company, New Jersey, 1993.
[19] W. Harrison, H.L. Ossher, P.L. Tarr, Asymmetrically vs Symmetrically Organized Paradigms for Software Composition, IBM Research Report RC22685 (W0212 147) Thomas J. Watson Research Center, IBM, December, 2002.
[20] D. Irastorza, A. Jaime, O. Díaz, Component Based Design: alternatives for the partition step, in: Proceedings of the Conference on Software Engineering and Databases (JISBD), Spain, 2000. (in Spanish).
[21] M. Katara, S. Katz, Architectural views of aspects, in: Proceedings of the International Conference on Aspect Oriented Software Development (AOSD), ACM Press, March, 2003.
[22] G. Kiczales, E. Hilsdale, J. Huguin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, in: Proceedings of the 15th European Conference on Object Oriented Programming, LNCS 2072, Springer Verlag, Budapest, Hungary, June 18 22, 2001.
[23] G. Kizcales, J. Lamping, A. Mendhekar, C. Maeda, Aspect oriented programming, in: Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP), LNCS 1241, Springer Verlag, Jyvaskyla, Finland, June 9 13, 1997.
[24] S. McDirmid, W.C. Hsieh, Aspect Oriented Programming with Jiazzi, in: Proceedings of the 2nd International Conference on Aspect Oriented Software Development (AOSD), Boston, Massachusetts, USA, March, 2003, pp. 70 79.
[25] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions of Software Engineering 26 (1) (2000).
[26] R. Milner, The Polyadic pi calculus: a Tutorial, Logic and Algebra of Specification, Springer, 1993, pp. 203 246.
[27] A. Moreira, J. Araújo, A. Rashid, A concern oriented requirements engineering model, in: Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE), LNCS 3520, Springer Verlag, Porto, Portugal, June 13 17, 2005.
[28] J. Munnelly, S. Fritsch, S. Clarke, An aspect oriented approach to the modularisation of context, in: Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom), New York, USA, 2007.
[29] E. Navarro, P. Letelier, I. Ramos, Goals and quality characteristics: separating concerns, in: Proceedings of the Early Aspects 2004: Workshop on Aspect Oriented Requirements Engineering and Architecture Design, collocated to OOPSLA 2004, Vancouver, Canada, Monday, October 25, 2004.
[30] E. Navarro, I. Ramos, J. Pérez, Software requirements for architectured systems, in: Proceedings of 11th IEEE International Requirements Engineering Conference, Monterey, California, USA, IEEE Computer Society, ISBN 0 7695 1980 6, September 8 12, 2003, pp. 365 366.
[31] A. Navasa, M.A. Pérez, J.M. Murillo, Aspect modelling at architecture design, in: Proceedings of the Second European Workshop on Software Architecture, LNCS 3527, Springer Verlag, Pisa, June 2005.
[32] P. Noriega, C. Sierra, Auctions and multi agents systems, Latin American Journal of Artificial Intelligence 6 (1998) 68 84.
[33] H. Ossher, P. Tarr, Multi dimensional separation of concerns and the hyperspace approach, in: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2000.
[34] H. Ossher, P. Tarr, Multi Dimensional Separation of Concerns using Hyperspaces, IBM Research Report 21452, April, 1999.
[35] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (12) (1972) 1053 1058.
[36] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond Aubry, L. Martelli, JAC: an aspect based distributed dynamic framework, Software Practice and Experience 34 (2004) 1119 1148.
[37] J. Pérez, N. Ali, J.A. Carsí, I. Ramos, Dynamic Evolution in Aspect Oriented Architectural Models, in: Proceedings of the Second European Workshop on Software Architecture, LNCS 3527, Springer Verlag, Pisa, June 2005.
[38] J. Pérez, N. Ali, C. Costa, J.A. Carsí, I. Ramos, Executing aspect oriented component based software architectures on .NET technology, in: Proceedings of the Conference on .NET Technologies, vol. 3, No.1 3, ISSN 1801 2108, 2005.
[39] J. Pérez, N. Ali, J.A. Carsí, I. Ramos, Designing software architectures with an aspect oriented architecture description language, The 9th International Symposium on Component Based Software Engineering (CBSE 2006), Malardalen University, Vasterås near Stockholm, Sweden, June 29th 1st July 2006.
[40] D. Perry, A. Wolf, Foundations for the study of software architecture, ACM Software Engineering Notes 17 (4) (1992) 40 52.
[41] M. Pinto, L. Fuentes, J.M. Troya, DAOP ADL: An architecture description language for dynamic component and aspect based development, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), LNCS 2830, Springer Verlag, Erfurt, Germany, September 22 25, 2003.
[42] PRISMAWeb Page, <http://prisma.dsic.upv.es>.
[43] K.U. Scholl, J. Albiez, B. Gassmann, MCA: an expandable modular controller architecture, in: Proceedings of the 3rd Real Time Linux Workshop, Karlsruhe University, Milano, Italy, 2001.

[44] M. Shaw, Procedure calls are the assembly language of software interconnection: connectors deserve first class status, in: Proceedings of Workshop on Studies of Software Design, January, 1994.
[45] G.M.C. Sousa, J.B. Castro, Towards a Goal Oriented Requirements Methodology Based on the Separation of Concerns Principle. Workshop on Requirements Engineering (WER), ISBN 8587926071, Piracicaba SP, Brasil, November, 27 28, 2003.
[46] C. Stirling, in: Modal and temporal logicsHandbook of Logic in Computer Science, vol. 2, Clarendon Press, Oxford, 1992.
[47] S. Sutton, I. Rouvellou, Modeling of software concerns in cosmos, in: G. Kiczales (Ed.), Proceedings of the 1st International Conference on Aspect Oriented Software Development (AOSD), 2002, pp. 127 133.
[48] D. Suvée, B. De Fraine, W. Vanderperren, A symmetric and unified approach towards combining aspect oriented and component based software development, in: Proceedings of the 9th International Symposium on Component Based Software Engineering (CBSE 2006), Malardalen University, Vasterås near Stockholm, Sweden, June 29th 1st July 2006.
[49] D. Suvée, W. Vanderperren, D. Wagelaar, V. Jonckers, There Are No Aspects. Electronical Notes in Theoretical Computer Sciences (ENTCS), Special Issue on Software Composition, vol. 114, January, 2005, pp. 153 174.
[50] P. Tarr, H. Ossher, W. Harrison Jr., S.M. Sutton, N degrees of separation: multidimensional separation of concerns, in: Proceedings of the 21st International Conference on Software Engineering (ICSE), Communication of the ACM, New York, 1999, pp. 107 119.
[51] The TeachMover Robot, <http://www.questechzone.com/microbot/teachmover.htm>.
[52] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, H. Das, The CLARAty architecture for robotic autonomy, in: Proceedings of IEEE Aerospace Conference, vol. 1, Montana, USA, 2001, pp. 121 132.
[53] Y. Yu, J.C. Leite, J. Mylopoulos, From goals to aspects: discovering aspects from goal models, in: Proceedings of the 12th IEEE International Conference on Requirements Engineering (RE), IEEE Computer Society, Kyoto, Japan, September, 2004.

Comparing Architecture Description Languages for Mobile Software Systems

Nour Ali, Carlos Solís, Isidro Ramos
Polytechnic University of Valencia
Department of Information Systems and Computation
Valencia, 46022, Spain
+34 96 387 7007

{nourali, csolis, iramos}@dsic.upv.es

ABSTRACT

Software architecture is a technique which aids the development of complex and dynamic systems. Architecture Description Languages (ADLs) describe software architectures using a textual syntax or a graphical notation. However, not many ADLs have provided primitives for describing software architectures of distributed and mobile software systems. This paper presents a comparison among existing ADLs that have addressed distributed and mobile software systems. The features chosen for the comparison have been taken from the work of Roman et al. [15] which propose a framework for viewing mobility. The features taken into account in this work are how ADLs support the notion of location, mobility, coordination, middleware, a graphical notation, and tools. The work presented in this paper proposes to be a starting point for discovering whether ADLs have properly supported mobility or not.

Categories and Subject Descriptors

D.2.11 Software Architectures

General Terms

Design, and Languages

Keywords

Mobile software systems, distributed systems, software architectures, ADLs, comparison.

1. INTRODUCTION

Software Architecture is a discipline that focuses on the design and specification of overall system structure. A system structure is organized from elements that are composed into more complex ones until the overall system structure is obtained. Software architecture also deals with structural issues related to its elements such as their communication protocols, their data access, their functional assignment, and their physical distribution [16].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAM'08, May 10, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-022-7/08/05...$5.00.

Architecture Description Languages (ADLs) are textual or graphical languages that describe software architectures for aiding their understanding and communication. Medvidovic and Taylor [12] explain that an ADL provides the description of the building blocks of software architectures. These building blocks are components, connectors and configurations. Many ADLs can be found in the literature, each emerged for different purposes and domains.

Software architectures is an appropriate technique for describing distributed and mobile software systems due to the fact that it can aid in developing this kind of systems and in analyzing their non-functional properties such as performance, reliability, and security. In addition, different architectural decisions can be made in allocating architectural elements to the nodes of a network in order to fulfil certain properties such as response time or latency. However, not many ADLs encountered in the literature have provided primitives for describing software architectures of distributed and mobile software systems.

This paper presents a comparison among existing ADLs which have described distributed and mobile systems. These are Darwin [9], C2Sadel [11], Community [8], MobiS [5], LAM Model [17], π-ADL [13], Con-Moto [7] and Ambient-PRISMA [1]. The main features used for the comparison have been proposed by Roman et al. [15]. Other works have compared ADLs with different features. However, there is no existing comparison performed for analyzing how ADLs have supported mobile software systems.

This paper is structured as follows: Section 2 motivates why ADLs can be useful for describing mobile software systems. Section 3 explains the features which are used in comparing the different ADLs. Section 4 presents the comparison. Section 5 presents conclusions and further work.

2. ADLs AND MOBILE SYSTEMS

An important characteristic of distributed systems is location-transparency. This means that elements of a distributed system should be provided with the same functionalities as if they were in a centralized system. This implies that:

- Clients are not aware of where their servers are located in order to perform communications.
- Resources are accessible even though they are distributed

Ali, N., Solís, C., Ramos, I. Comparing architecture description languages for mobile software systems. Proc. of the First International Workshop on Software Architectures and Mobility at ICSE'08. 2008.

- Elements can move without others noticing it i.e. they are accessible and able to communicate with elements and resources in their new location.

In software architectures, components define computation and connectors are responsible for coordination. In this way, software architecture is a technique that can be used for separating computation from coordination. This is important for providing elements with transparent distributed communication and adaptability such as mobility.

Software Mobility is mainly used to achieve application level adaptation such as service customization, dynamic functionality extension, disconnected operation support or fault tolerance [4]. A widely used definition for Code Mobility [4] (or software mobility) is the following:

<<Code Mobility is the capability to reconfigure dynamically, at runtime, the binding between the software components of the application and their physical location within a computer network>>

It can be noticed from the previous definition that code mobility is the capability of a distributed application to connect and disconnect code fragments to/from different locations at run-time. In addition, hardware mobility (or physical mobility) can be seen as the disconnection and connection of devices in a network. Software architecture is a useful approach for modelling mobile systems due to the fact that they are an example of complex and dynamic systems that need to be adapted and reconfigured to network resources, or non-functional requirements such as fault tolerance, security, and performance.

Basically, ADLs that support the description of mobile software architectures should provide mechanisms for describing dynamic systems. Surveys have been already performed on different ADLs which support dynamic systems such as in [6] and [2]. However, no works have been performed in order to survey ADLs basing on features which are unique and essential for mobile systems such as how a notion of location has been provided in ADLs, or how and an architectural element can move from a location to another. This paper proposes to be a starting point for discovering whether ADLs have properly supported mobility or not.

3. FEATURES FOR COMPARING ADLs FOR MOBILE SOFTWARE SYSTEMS

There are many features that can be used in order to compare and analyze the ADLs. However, the features that are of interest in the scope of this paper are the ones related with distribution and mobility. The paper of Roman et.al [15] describes a framework for viewing mobility. It is an appropriate starting point for discovering features that are essential for models that need to support mobility. They propose that space and coordination are the most important dimensions to be considered for dealing with mobility, and models providing mobility are differentiated from others basing on how they assume the unit of mobility, location and context which highly depends on the coordination mechanisms that a model provides.

In the following, the features that have been considered in the comparison are explained:

- Location: Locations represent the different positions where a mobile entity can move in space. Locations have to be explicitly dealt as first-class entities and be distinguished from other entities of a model. As a result, space can be modelled, and it can be specified where a mobile entity can or cannot move into.
- Location-Awareness: This feature determines whether an entity can be aware of its current location or not. This feature is important because it allows an entity to take decisions depending on its current location.
- Mobility Support: This feature describes how a model supports the movement of a mobile entity. Basically, mobility support determines what implications are caused when an entity is moved such as a reconfiguration of an architecture or a change in a value.
- Unit of Mobility: This feature represents the smallest entity of a model that is allowed to move. This feature is important because it represents which entities of a model can be mobile and which cannot.
- Migration Decision: This feature specifies when and what causes an entity to move. This feature is associated with the terms objective and subjective mobility. Objective mobility is the movement of an entity caused by the environment. Subjective mobility is the movement of an entity caused by the same entity.
- Coordination: The coordination mechanisms that a model supports determine the context that is seen by each entity. The coordination mechanisms should be specified separately from the functionality of the entity. Since a basic characteristic of ADLs is the separation between coordination and computation, all of them support this characteristic. Therefore, what differentiates the coordination feature in ADLs is whether they support connectors or not.
- Formalism: Models have to be formal enough in order to enable a precise description of the semantics of the distribution and mobility properties. The formalism chosen should provide the primitives needed to specify mobility features.
- Graphical Support: This feature has not been mentioned by Roman et.al [15]. Some ADLs provide graphical support in order to be more usable and understandable. However, the objective of this comparison is to discover which ADLs provide a graphical notation for specifically describing the distribution and mobility primitives. A basic graphical notation for distributed systems is specifying where entities are distributed.
- Middleware: Middleware implement distribution and mobility primitives that a model provides. In this comparison the objective is to discover to what extend ADLs have been used to implement executable distributed and mobile applications.
- Tool Support: This feature has not been mentioned by Roman et.al [15]. However, it has been considered in the comparison of [12]. A tool supports a developer by providing facilities and guides. Some of the facilities can be modelling support, verification, code generation, code execution, or simulation. The objective of this

comparison is to discover what facilities ADL tools provide for distribution and mobility.

4. COMPARING ADLs FOR DISTRIBUTION AND MOBILITY

This section presents how each ADL supports the features presented in section 3. The ADLs chosen for the comparison are: Darwin [9], C2Sadel [11], Community [8], MobiS [5], LAM Model [17], π-ADL [13], Con-Moto [7] and Ambient-PRISMA [1]. These ADLs have been chosen because they were the only ones found in the state of art which have declared to support primitives for distributed and mobile software systems.

4.1 Location

The surveyed ADLs have provided the notion of location in an implicit or an explicit way. For example, π-ADL has used composite components to represent locations. However, there is no difference between a composite component representing a location and a composite component which does not represent a location. MobiS and LAM model also have implicitly provided locations in a similar way to π-ADL. For example in MobiS, the same primitive which is called space is used for representing components which can be nested, locations and channels. In the LAM model, locations are called components. Components can be seen as a composite component which contains an internal connector that connects agents (subcomponents) to the location.

Other ADLs have provided an explicit notion for locations by defining special kinds of architectural elements. For example, Con-Moto has two kinds of components: physical and logical components. Physical components model devices and logical components model software. C2Sadel and Ambient-PRISMA have defined special kinds of connectors for explicitly representing locations. In C2Sadel, a border connector represents each host of a network. In Ambient-PRISMA, ambients model a boundary which separates what is in and out of it (similarly to the concept of ambient of Ambient Calculus [1]). As a result, an ambient can be a host, a LAN, a geographical area, or any boundary of a network. In C2Sadel and Ambient-PRISMA it is argued that connectors are appropriate for modeling locations because connectors coordinate architectural elements with distributed architectural elements for communication purposes.

In the previous approaches, software architecture concepts have been used in order to define the notion of location. On the other hand, other approaches have provided values which represent locations. For example in Darwin, an expression "*F@k"* is used in order to indicate that the integer value *k* represents a machine where the component *F* is located. In Community, an abstract data type models locations and a value of this abstract data type represents a location.

In addition, ADLs can be classified depending on the type of locations they provide, i.e., whether locations are flat or hierarchically structured. MobiS, and Ambient-PRISMA support hierarchically structured locations. Community, Darwin, C2Sadel, LAM model, π-ADL, and Con-Moto provide flat locations.

4.2 Location-Awareness

Community and Ambient-PRISMA are explicitly location aware because architectural elements know where they are located. This is because location is part of the state of each architectural element. The other approaches are not explicitly location aware but their architectural elements can become when they request it if another architectural element in the software architecture is modeled in order to provide location-awareness. For example in Darwin [10], a component called LOCATE which stores the locations of instances is modeled. Any component which needs to be location aware has to request the LOCATE component.

4.3 Mobility Support

Excluding Darwin, all the ADLs surveyed have used their ADLs for supporting mobile systems.

Mobility is supported in a way or another depending on whether the notion of location has been introduced as an architectural element or a variable. Basically, mobility has been supported by reconfiguring the software architecture when the notion of location has been introduced as an architectural element. In the case of Community, mobility is supported by changing the value of the variable which represents location.

ADLs which support hierarchical locations only allow step by step migration upwards or downwards in the hierarchy of locations. These include MobiS, and Ambient-PRISMA. On the other hand, C2Sadel, LAM model, Community, π-ADL and Con-Moto only flat migration is supported.

In the following, how ADLs define their unit of mobility and how they support the migration decision is explained:

4.3.1 Unit of Mobility

Each ADL has defined its own unit of mobility. In MobiS, spaces are mobile, i.e., components, locations, and channels. In Community, fine grained component or connector mobility is provided. In π-ADL and the LAM model, subcomponents can be mobile. In Con-Moto, logical components can be mobile. In C2Sadel components are mobile. Finally in Ambient-PRISMA, components, connectors, composite components and ambients can be mobile.

It can be noticed from the above that the only approaches which allow mobility of connectors are Community and Ambient-PRISMA. This is an important property to be considered because connectors provide the coordination mechanisms of components As a result, coordination mechanisms have to be enabled to move with components.

Analyzing the unit of mobility allows us to detect which ADLs support physical or software mobility. Physical mobility can be supported by allowing locations to be mobile. MobiS and Ambient-PRISMA allow locations to be mobile. As a result, these can provide the modeling of physical mobility. In addition, Community allows modeling physical mobility because the variable which represents location can change when a component is modeled as a physical device.

4.3.2 Migration Decision

Most of the ADLs provide the specification of both objective and subjective moves. This is because most ADLs provide mechanisms for distinguishing what can be controlled by the environment and what can be controlled by an architectural element. For example, Community provides explicit distribution connectors that specify which components can change the location of other components as well as provides input location variables and output location variables. Input location variables

are under the control of the environment whereas output location variables are controlled by the design.

However, in C2Sadel only objective moves are available because an external component is in charge moving components. In Con-Moto, it is not clear what causes the decision of mobility.

4.4 Coordination

An important characteristic of ADLs is that they separate coordination from computation. However, not all ADLs provide connectors as first class entities. In this way, complex coordination mechanisms concerned with distribution and mobility cannot be modeled. MobiS, Darwin, and Con-Moto do not provide connectors. In MobiS, component interaction is defined by the coordination model. In Darwin, components are bound to other components by simple bindings. In Con-Moto, it is distinguished between physical connections which connect physical components and logical connections which connect logical components.

C2Sadel provides different kinds of connectors including ones which are concerned with communication protocols. As it has been commented in section 4.1, border connectors on each host are located in order to provide distributed communication for components. Community as it has been commented in section 4.3.2 provides distribution connectors which synchronize the mobility of components. LAM model provides an internal connector on each location in order to connect each agent (component) with its location. π-ADL provides explicit support for the definition of first class connectors. However, there has not been an example found on how π-ADL supports connectors for distribution and mobility. Ambient-PRISMA provides ambients which are a special kind of connector for coordinating architectural elements for distributed communication purposes. Ambients are also used in order to provide mobility support. Thus, an ambient coordinates a mobile architectural element and its destination ambient in order to allow the destination ambient to receive the mobile architectural element.

4.5 Formalism

All the ADLs surveyed have been based on a formal (or semi-formal) foundation. Darwin, π-ADL, Con-Moto are based on π-calculus for modeling interaction and composition. Although π-calculus is appropriate for systems where its structure is dynamic, it has been argued in many works such as [15] that it is not the most appropriate formalism for distributed and mobile systems because it does not provide a notion for location. C2Sadel is based on a first-order logic with pre-and post conditions for modeling component behaviour. However, its mobility primitives are not formal. Community is based on category theory and Mobile UNITY. The LAM model is based on a Predicate/Transition nets (a variant of Petri Nets) for modeling the behavior of it architectural elements. MobiS is based on multiple tuple-space and multiset rewriting. Ambient-PRISMA is based on a variant of Ambient Calculus called Channel Ambient Calculus [14] for modeling how architectural elements can enter and exit ambients (mobility) and how software architecture can be reconfigured when an architectural element is moved.

4.6 Graphical Support

All the surveyed ADLs provide a graphical notation. However, only Con-Moto and Ambient-PRISMA provide a graphical notation for distributed configurations. Darwin provides a graphical notation for describing configurations of components. C2Sadel provides a graphical notation for specifying how components and connectors are connected. Community also provides a graphical notation for specifying software architectures with components and connectors. MobiS provides an informal graphical notation which its authors have used in their papers where spaces are drawn as ovals. However, since MobiS does not have a unique primitive for locations, the reader does not distinguish a component from a location. π-ADL has defined a UML profile for defining its software architectures in a graphical way. However, no graphical notation for distribution is given. The LAM model uses Prt Nets notation (similar to Petri Nets). However, this notation is not very friendly due to the fact that it only specifies the sending and receiving of Petri Nets messages.

Con-Moto provides an explicit graphical notation for distribution. For example in Figure 1, the *uniPOS Server* and the *uniPOS Client* are physical components and the *Offer*, *OfferLogic* and the *OfferStorage* are logical components. It can be observed the difference between physical and logical connectors.

Ambient-PRISMA also provides an explicit notation for distribution. For example Figure 2 shows a hierarchy of ambients where the *Root* ambient locates two ambients called *ClientSite* and *AuctionSite*. The *ClientSite* ambient locates the *Customer1* component and the *AgentCustCnct* connector. The AuctionSite ambient locates the *Procurement1* component, the *AuctionHouseCnct1*, and the *AuctionHouse1* system.

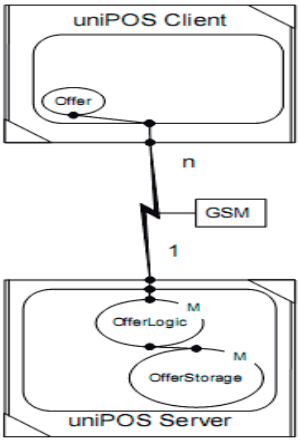


Figure 1.A structural model in Con Moto taken from [7]

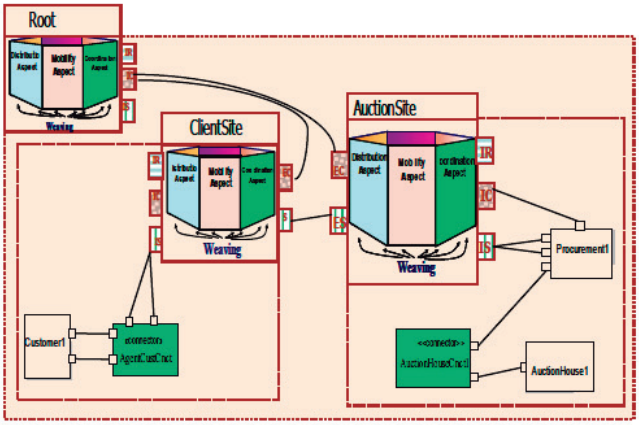


Figure 2. Graphical notation of Ambient-PRISMA

4.7 Middleware

Not all ADLs have focused on developing applications from their ADL specifications. However, it is of great use to demonstrate that your high level specifications will finally execute on a specific technology. The only approaches which have been implemented on a specific platform have been Darwin, C2Sadel and Ambient-PRISMA.

Darwin has been used in the Common Object Request Broker Architecture (CORBA) environment to specify the overall architecture of distributed component-based applications [10]. C2Sadel has implemented a framework which supports remote method invocations and code on demand mobility. Ambient-PRISMA has implemented a middleware called Ambient-PRISMANET which provides remote method invocations, agent mobility, distributed transactions, and automatic distributed deployment.

4.8 Tool Support

MobiS and the LAM model do not provide any tool support. Darwin provides a textual and graphical modeling tool, a compiler which generates C++ code, and its framework provides automatic distributed configuration. C2Sadels tool models software architectures textually and graphically, and generates application skeletons. Community has a graphical modelling, animation and simulation tool. π-ADL provides code generation to Java, visual modelling, and verification tools. Con-Moto provides a simulation tool. Ambient-PRISMA provides a graphical modeling tool, verification, and code generation of distributed and mobile applications for .NET.

5. CONCLUSIONS AND FURTHER WORK

This paper has used features mainly proposed by Roman et al. [15] for comparing ADLs that have addressed the description of distributed and mobile software systems. It has been noticed that ADLs can implicitly describe structures of this kind of systems. However, it is important to introduce explicit primitives for modeling more complex scenarios for mobility.

From the features that have been taken into account in the comparison, the state of art of the ADLs that have dealt with distributed and mobile systems can be summarized as follows:

- The notion of location has been introduced as: a new type of connector, a new type of component, a value of a data type, and a composite component. In the case when it has been introduced as a composite component the notion of location is quite ambiguous since the same concept is used for both representing locations and hierarchical compositions.
- The support of mobility highly depends on how the notion of location is introduced in the ADL. When location is represented by a value, mobility is supported by the change of this value. When location is supported by architectural element (component, connector, composite component) mobility is supported by reconfiguring the software architecture.
- Components are the units of mobility in most ADLs. Community and Ambient-PRISMA are the only ADLs that allow connectors to be mobile.
- Few ADLs allow their elements to be location aware.
- Most ADLs provide objective and subjective mobility.
- Most ADLs provide connectors as first-class entities in order to provide complex coordination mechanisms. However, the interesting approaches are those that provide special coordination mechanisms for distribution such as Community, C2Sadel, and Ambient-PRISMA.
- Most ADLs have been formalized by using formalisms that do not explicitly provide first-class entities for distribution and mobility.
- Most ADLs provide a graphical support to model components, connectors and configurations. However, an ADL that provides the graphical support for distributing architectural elements is needed. Providing a graphical support for describing the distribution and mobility issues allow the ADL to be more usable.
- The only ADLs that have been implemented on a specific middleware have been Darwin, C2Sadel, and Ambient-PRISMA. However, Darwin does not provide mobility and the mobility that C2Sadel allows is restricted. Ambient-PRISMA provides mobility for mobile agents.
- Each ADL provides tool support with a different focus. However, much more work has to be done in order to provide tools that support the analysis, validation, and verification of mobile systems.

This work is a starting point which can be extended for including more features which are essential for mobile software systems. One of these features is the support of non-functional requirements such as security, resource availability, bandwidths, latencies, etc.

Other future work which have to be done are to specify a specific case study of the mobility domain using the different ADLs surveyed in this paper, then empirically evaluate quality attributes such as the effort, understandability or expressiveness and finally compare the results of the resulting experiments.

6. ACKNOWLEDGMENTS

This work is funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, META project TIN2006-15175-C05-01.

7. REFERENCES

[1] Ali, N., Millan, C., Ramos, C. 2006. Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model. LNCS 4276, 1633-1649.

[2] Bradbury, J. S., Cordy, J. R., Dingel, J., Wermelinger, M. 2004. A survey of self-management in dynamic software architecture specifications. In Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed systems (Newport Beach, California), 28-33.

[3] Cardelli, L. 1998. Abstractions for Mobile Computation. In Vitek, J. and (Eds.), C. J., editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects, LNCS 1603, Springer Verlag, 51-94.

[4] Carzaniga, A., Picco, G.P., Vigna, G. 1997. Designing Distributed Applications with Mobile Code Paradigms. In Proceedings of the 19th Conference on Software Engineering (ICSE'97), R. Taylor, ed., ACM Press, 22–32.

[5] Ciancarini, P., Mascolo, C. 1998. Software architecture and mobility. In Proceedings of 3rd International Software Architecture Workshop (Orlando, FL, November). ISAW-3. In D. Perry and J. Magee, editors. ACM SIGSOFT Software Engineering Notes, 21-24.

[6] Cuesta, C. 2002. Dynamic Software Architecture based on Reflection. PhD. Thesis, University of Valladolid. (In Spanish)

[7] Gruhn,V., Schafer,C. 2004. An Architecture Description Language for Mobile Distributed Systems. In Proceedings of the First European Workshop on Software Architecture (EWSA 2004), Springer-Verlag, 212-218.

[8] Lopes, A., Fiadeiro, J.L., Wermelinger, M. 2002. Architectural Primitives for Distribution and Mobility. In Proceedings of 10th Symposium on Foundations of Software Engineering, ACM Press, 41-50.

[9] Magee, J., Dulay, N., Eisenbach S. and Krammer, J. 1995. Specifying Distributed Software Architectures. In Proceedings of the 5th European Software Engineering Conference (Sitges, Spain). ESEC 95, 137-153.

[10] Magee, J., Tseng, A., Kramer, J. 1997. Composing Distributed Objects in CORBA. In Proceedings of the Third International Symposium on Autonomous Decentralized Systems (Berlin, Germany), 257-263.

[11] Medvidovic, N., Rakic, M. 2001. Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility. In Proceedings of the Software Engineering and Mobility Workshop (Toronto, Canada, May).

[12] Medvidovic N., Taylor R.N. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions of Software Engineering, Vol. 26, Nº 1, January 2000.

[13] Oquendo, F. π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Software Engineering Notes, Vol. 29 Nº3, May, 2004.

[14] Philips, A. "Specifying and Implementing Secure Mobile Applications in the Channel Ambient System", PhD thesis, Imperial College London, October 2005.

[15] Roman, G., Picco, G., Murphy, A. 2000. Software Engineering for Mobility: A Roadmap. In Proceedings of the International Conference on Software Engineering - Future of SE Track (Limerick, Ireland), 241-258.

[16] Shaw, M. and Garlan, R. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, April, 1996.

[17] Xu, D., Yin, J., Deng, Y., Ding, J. A Formal Architectural Model for Logical Agent Mobility. IEEE Transactions on Software Engineering, Vol. 29, Nº 1, January, 2003.

MORPHEUS: a supporting tool for MDD

Elena Navarro[1] & Abel Gómez[2] & Patricio Letelier[2] & Isidro Ramos[2]

[1] Department of Computing Systems, University of Castilla-La Mancha, Spain.
Elena.Navarro@uclm.es

[2] Dep. of Information Systems and Computation, Polytechnic University of Valencia, Spain.
{agomez, letelier, iramos}@dsic.upv.es

Abstract: Model-Driven Development (MDD) approach is gaining more and more attention both from practitioners and academics because its positive influences in terms of reliability and productivity in the software development process. ATRIUM is one of the current proposals following the MDD principles as the development is driven by models and a tool, MORPHEUS, support both its activities and models. This tool provides facilities for modelling, meta-modelling, and analysis and integrates an engine to execute transformations. In this work, this tool is presented describing both its architecture and its capabilities.

Keywords: Model-Driven Development, Requirements Engineering, Software Architecture

1. Introduction

Software development process is always a challenging activity, especially because systems are becoming more and more complex. In this context, the Model-Driven Development [24] (MDD) approach is gaining more and more attention from practitioners and academics. This approach promotes the exploitation of models at different abstraction levels, guiding the development process by means of transformations, so that traceability and automatic support becomes a reality. MDD has demonstrated positive influences for reliability and productivity of the software development process due to several reasons [24]: it allows one to focus on the ideas and not on the supporting technology; it facilitates not only the analysts get an improved comprehension of the problem to be solved but also the stakeholders obtain a better cooperation of the software development; etc. With those aims, MDD exploits models both to properly document the system and automatically or semi-automatically generate the final system. This is why the software development is shifting its attention [1] from "everything is an object", so trendy in the eighties and nineties, to "everything is a model."

ATRIUM [12][15] (Architecture generaTed from RequIrements applying a Unified Methodology) has been defined following the MDD principles, as models

Navarro, E., Gómez, A., Letelier, P., Ramos, I. MORPHEUS: a supporting tool for MDD. Proc. of the18th International Conference on Information Systems Development (ISD2009). 2009.

drive its application, and the tool MORPHEUS (see [13] for demos) has been built to support its models and activities. This methodology has been defined to guide the concurrent definition of requirements and software architecture, paying special attention to the traceability between them. In this context, the support of MORPHEUS is a valuable asset allowing the definition of the different models; maintaining traceability among them; supporting the necessary transformation, etc. This paper focuses on MORPHEUS and its support to a MDD process.

This paper is structured as follows. After this introduction, a brief description of ATRIUM is presented in section 2. Section 3 describes the supporting tool of ATRIUM, MORPHEUS. Related works are described in section 4. Finally, section 5 ends this paper by presenting the conclusions and further works.

2. ATRIUM at a glance

ATRIUM provides the analyst with guidance, along an iterative process, from an initial set of user/system needs until the instantiation of the proto-architecture. ATRIUM entails three activities to be iterated over in order to define and refine different models and allow the analyst to reason about partial views of both requirements and architecture. Fig. 1 shows, using SPEM [21] (Software Process Engineering Metamodel), the ATRIUM activities that are described as follows:

- *Modelling Requirements*. This activity allows the analyst to identify and specify the requirements of the system-to-be by using the *ATRIUM Goal Model* [18], which is based on KAOS [5] (Knowledge Acquisition in automated specification) and the NFR (Non-Functional Requirements) Framework [2]. This activity uses as input both an informal description of the requirements stated by the stakeholders, and the CD 25010.2 *Software product Quality Requirements and Evaluation Quality model (SQuaRE* [9]). The latter is used as framework of concerns for the system-to-be. In addition, the architectural style to be applied is selected during this activity [15].

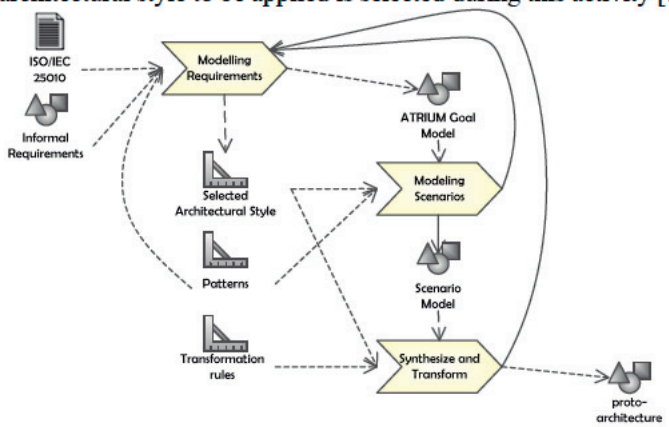


Fig. 1. An outline of ATRIUM

- *Modelling Scenarios*. This activity focuses on the specification of the *ATRIUM Scenario Model,* that is, the set of *Architectural Scenarios* that describe the system's behaviour under certain *operationalization* decisions [16]. Each ATRIUM Scenario identifies the architectural and environmental elements that interact to satisfy specific requirements and their level of responsibility.
- *Synthesize and Transform*. This activity has been defined to generate the proto-architecture of the specific system [14]. It synthesizes the architectural elements from the ATRIUM scenario model that build up the system-to-be, along with its structure. This proto-architecture is a first draft of the final description of the system that can be refined in a later stage of the software development process. This activity has been defined by applying *Model-To-Model Transformation* (M2M, [4]) techniques, specifically, using the QVT Relations language [20] to define the necessary transformations. It must be pointed out that ATRIUM is independent of the architectural metamodel used to describe the proto-architecture, because the analyst only has to describe the needed transformations to instantiate the architectural metamodel he/she deems appropriate. Currently, the transformations [15] to generate the proto-architecture, instantiating the PRISMA architectural model [22], have been defined. PRISMA was selected because a code compiler exists for this model.

ATRIUM has been validated in the context of the tele-operated systems. Specifically, the EFTCoR [8] project has been used for validation purposes. The main concern of this project was the development of a tele-operated platform for non-pollutant hull ship maintenance operations whose main structure is shown in Fig. 2. In this paper, we are going to use the specification made of the Robotic Device Control Unit (RDCU) to show how MORPHEUS provides support to each activity of ATRIUM. The RDCU is in charge of commanding and controlling in a coordinated way the positioning of devices along with the tools attached to them.

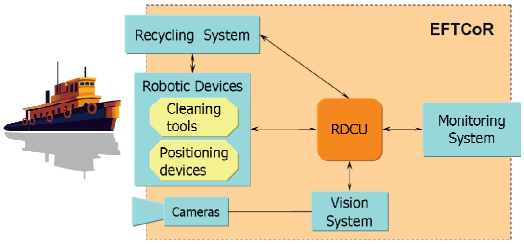


Fig. 2. Describing the EFTCoR platform

3. MORPHEUS: a MDD supporting tool

The main idea behind MORPHEUS is to facilitate a graphical environment for the description of the three models used by ATRIUM (Goal Model, Scenarios Model, and PRISMA Model) in order to provide the analysts with an improved legibility

and comprehension. Several alternatives were evaluated such as the definition of profiles, or the use of meta-modelling tools. Eventually, we developed our own tool in order to provide the proper integration and traceability between the models.

Fig. 3 shows the main elements of MORPHEUS. The *Back-End* layer allows the analyst to access to the different environments, and to manage the projects he/she creates. Beneath this layer, the different environments of MORPHEUS are shown, providing each one of them support to a different activity of ATRIUM. The *RepositoryManager layer* is in charge of providing the different environments with access to the repository where the different models and metamodels are stored. In addition, each one of the graphical environments (Requirements Model Editor, Scenario Editor, and Architecture Model Editor) exploits Microsoft Office Visio Drawing Control 2003 [25] (VisioOCX in Fig. 4, Fig. 9, Fig. 13) for graphical support. This control was selected to support the graphical modelling needs of MORPHEUS because it allows a straightforward management, both for using and modifying shapes. This feature is highly relevant for our purposes because all the kinds of concepts that are included in our metamodels can easily have different shapes, facilitating the legibility of the models. In addition, the user is provided with all the functionalities that Visio has, that is, she/he can manage different diagrams to properly organize the specification, make zoom to see more clearly details, print the active diagram, etc. In the following sections, each one of the identified environments is described.



Fig. 3. Main architecture of MORPHEUS

1.1 Requirement Environment

As described in section 2, *Modelling Requirements* is the first activity of ATRIUM. In order to support this activity, the *Requirements Environment* was developed. From the very beginning of the EFTCoR project, one of the main problem we faced was how the requirements metamodel had to change to be adapted to the specific needs of the project. With this aim, this environment was developed with two different work contexts. The first context is the *Requirements Metamodel Editor* (RMME), shown in Fig. 4, which provides users with facilities for describing requirement meta-models customized according to project's semantic needs (see Fig. 5). The second context is the *Requirements Model Editor*

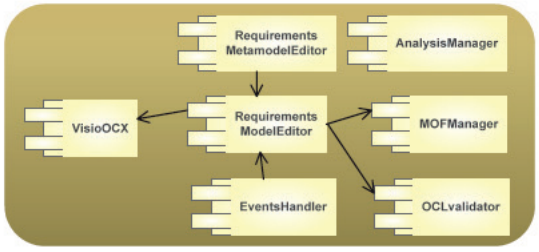


Fig. 4. Main elements of the Requirements Environment

(RME), also shown in Fig. 4, which automatically offers the user facilities to graphically specify models according to the active metamodel (see Fig. 8). These facilities are very useful to exploit MORPHEUS to support other proposals.

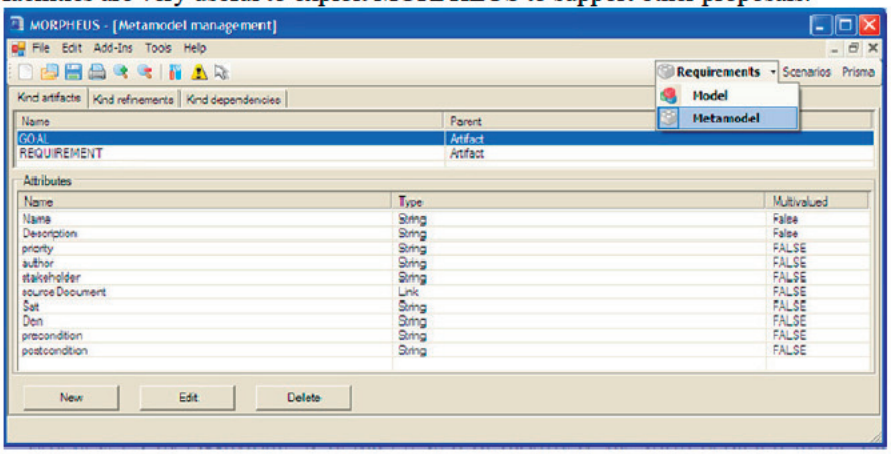


Fig. 5. Meta-Modelling work context (RMME) of the MORPHEUS Requirements Environment

It can be observed in Fig. 5 that the RMME allows the user to describe new meta-elements by extending the core metamodel described in Fig. 6, that is, new types of artefacts, dependencies, and refinements. This metamodel was identified and evaluated its applicability by analysing the existing proposals in requirements engineering [18]. For instance, Fig. 5 shows that the two meta-artifacts (goal, requirement) of the ATRIUM Goal Model were defined using the RMME. In order to fully describe the new meta-elements, the user can describe their meta-attributes and the OCL constraints he/she needs to check any property he/she deems appropriate. Fig. 7 shows how the meta-artifact *goal* was defined by

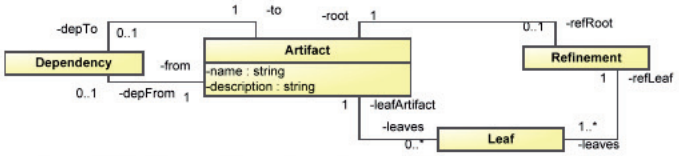


Fig. 6. Core-metamodel for the requirements environment

extending *artifact*; describing its meta-attributes *priority*, *author*, *stakeholder*, etc; and specifying two constraints to determine that the meta-attributes *stakeholder* and *author* cannot be null.

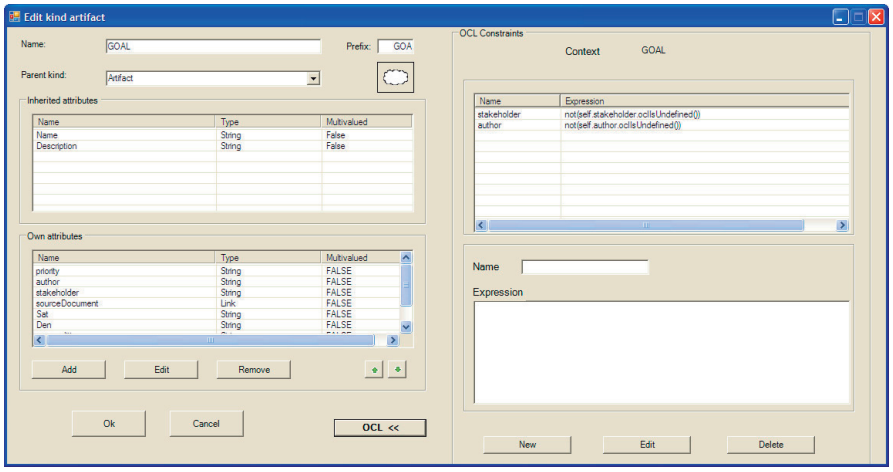


Fig. 7. Describing a new meta-artifact in MORPHEUS

It is worth noting that automatic support is provided by the environment for the evolution of the model, that is, as the metamodel is modified, the model is updated in an automatic way to support those changes, asking the user to confirm the necessary actions whenever a delete operation is performed on meta-elements or meta-atributes. This characteristic is quite helpful because the requirement model can be evolved as the expressiveness needs of the project do.

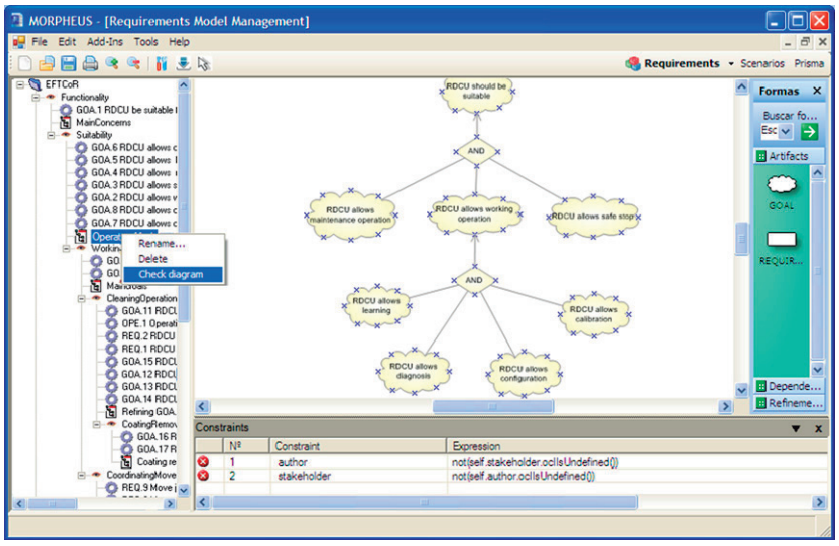


Fig. 8. Modelling work context (RME) of the MORPHEUS Requirements Environment

Once the metamodel has been defined the user can exploit it in the modelling context, RME, shown in Fig. 8. It uses VisioOCX to provide graphical support, as Fig. 4 shows, and has been structured in three main areas. On the right side, the *stencils* allow the user to gain access to the active metamodel. Only by dragging and dropping these meta-elements on the drawing surface in the centre of the environment, the user can specify the requirements model. He/she can modify or delete these elements by clicking just as usual in other graphical environments. For instance, some of the identified goals and requirements of the EFTCOR are described in the centre of the Fig. 8. On the left side of the RME, a browser allows the analyst to navigate throughout the model and modify it. As Fig. 4 illustrates, the *EventHandler* is in charge of manipulating the different events that arise when the user is working on the RME.

In addition, as Fig. 4 illustrates, the RME uses two components to provide support to OCL: *OCLvalidator* and *MOFManager*. The former is an engine to check OCL constraints that was integrated in MORPHEUS. The later was developed to allow us to manipulate metamodels and models in MOF [19] format. By exploiting these components the constraints defined at the metamodel can be automatically checked. For instance, when the active diagram was checked, two inconsistencies were found that are shown at the bottom of the Fig. 8.

However, the support of the tool would be quite limited if it only provides graphical notation. For this reason, the *Analysis Manager*, shown in Fig. 4, has been developed to allow the user to describe and apply those rules necessary to analyse its model. These rules are defined by describing how the meta-attributes of the meta-artifacts are going to be valuated depending on the meta-atributes of the meta-artifacts they are related by means of which meta-relationships. Once these rules are defined, the Analysis Manager exploits them by propagating the values from the leaves to the roots of the model [17]. This feature can be used for several issues such as, satisfaction propagation [17], change propagation, or analysis of architectural alternatives [15].

1.2 Scenario Environment

As presented in section 2, *Modelling Scenarios* is the next activity of ATRIUM. This activity is in charge of describing the scenario model. This model is exploited to realize the established requirements in the goal model by describing

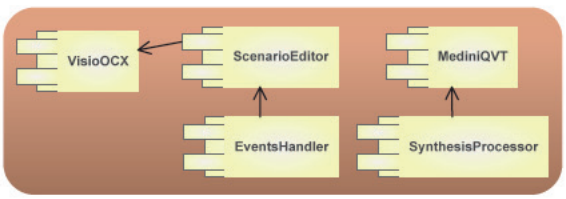


Fig. 9. Main elements of the Scenario Environment

partial views of the architecture, where only shallow-components, shallow-connectors and shallow-systems are described. In order to describe these scenarios, an extension of the UML2 Sequence Diagram has been carried out to provide the necessary expressiveness for modelling these architectural elements [15]. In order to provide support to this activity the *Scenario Editor* (SME), shown in Fig. 10, was developed. The *ScenariosEditor* uses the VisioOCX to provide the user with graphical support for modelling the Scenario Model. The *EventHandler* is in charge of managing all the events trigged by user actions. Fig. 10 illustrates how the SME has been designed. In a similar way to the RMME described in the previous section, it has been structured in three main areas. The *Model Explorer*, on the right, facilitates the navigation through the Scenario Model being defined in an easy an intuitive way and manages (creation, modification and deletion) the defined scenarios. It is pre-loaded with part of the information of the requirements model being defined. For this reason, the selected *operationalizations*, catalogued by their dimensions, are displayed. It facilitates to maintain the traceability between the Goal Model and the Scenario Model. Associated to each operationalization one or several scenarios can be specified to describe how the shallow architectural elements collaborate to realize that operationalization. In the middle of the environment is situated the *Graphical View* where the elements of the scenarios can be graphically specified. In this case, Fig. 10 depicts the scenario "OpenTool" that is realizing one of the operationalizations of the goal model. It can be observed how several architectural and environmental elements are collaborating by means of a sequence of messages. On the right side it can be seen the *Stencil* that makes available the different shapes to graphically describe the ATRIUM scenarios. The user only has to drag and drop on the *Graphical View* the necessary shapes. In addition, below the stencil a control allows the user to introduce the necessary properties for each element being defined.

Another component of the Scenario Environment is the *Synthesis processor* (see Fig. 11). It provides support to the third activity of ATRIUM *Synthesis and Transform* which is in charge of the generation of the proto-architecture. For its

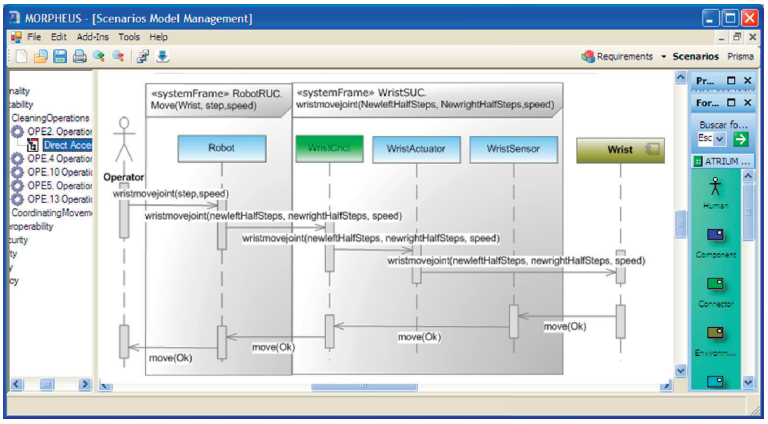


Fig. 10. What the Scenario Editor (SME) looks like

development, the alternative selected was the integration of one of the existing M2M transformation engines considering that it has to provide support to the QVT-Relations language. Specifically, a custom tool based in the *medini QVT* [11] transformation engine (licensed under the *Eclipse Public License*) was integrated as Fig. 11 illustrates. It accepts as inputs the metamodels and their corresponding models in XMI format to perform the transformation. This engine is invoked by the *Synthesis processor* which proceeds in several steps. First, it stores the Scenario Model being defined in XMI. Second, it provides the user with a graphical control to select the destination target architectural model, the QVT transformation to be used and the name of the proto-architecture to be generated. By default, PRISMA is the selected target architectural model because the QVT rules [15] for its generation have been defined. However, the user can define its own rules and architectural metamodel to synthesize the Scenario Model. Finally, the *Synthesis processor* performs the transformation by invoking the QVT engine. The result is an XMI file describing the proto-architecture.

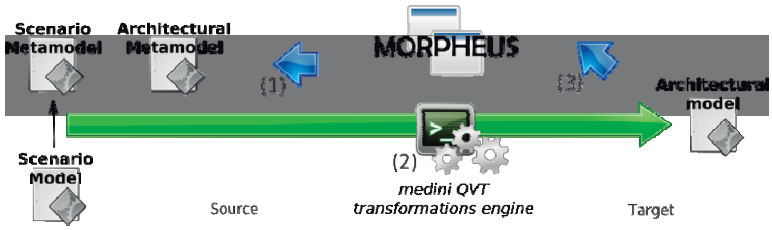


Fig. 11. Describing the Synthesis processor

1.3 Software Architecture Environment

As can be observed, both the Requirements Environment and the Scenario Environment provide support to the three activities of ATRIUM. However, as specified in section 2, a proto-architecture is obtained at the end of its application. This proto-architecture should be refined in a latter stage of development to provide a whole description of the system-to-be. With this aim the *Software Architecture Environment* [23] was developed. It makes available a whole graphical environment for the *PRISMA Architecture Description Language* [22] so that the proto-architecture obtained from the scenarios model can be refined.

As Fig. 13 depicts, this environment integrates VisioOCX for graphical support in a similar way to the previous ones. The *Architectural Model Editor* is the component that provides the graphical support, whose appearance can be seen in Fig. 12. It has three main areas: the stencil on the right where the PRISMA concepts are available to the user, the graphical view in the centre where the different architectural elements are described; and the model explorer on the right. It is worthy of note that this browser is structured in two levels following the

recommendation of the ADL [23]: definition level, where the PRISMA types are defined; and configuration level where the software architecture is configured.

As this environment should allow the user to refine the proto-architecture obtained from the synthesis of the scenario model, it provides her/him with facilities to load the generated proto-architecture if PRISMA was the selected target architectural model. In addition, it also provides an add-in that facilitates the generation of a textual PRISMA specification, which can be used to generate C# code by using the PRISMA framework.

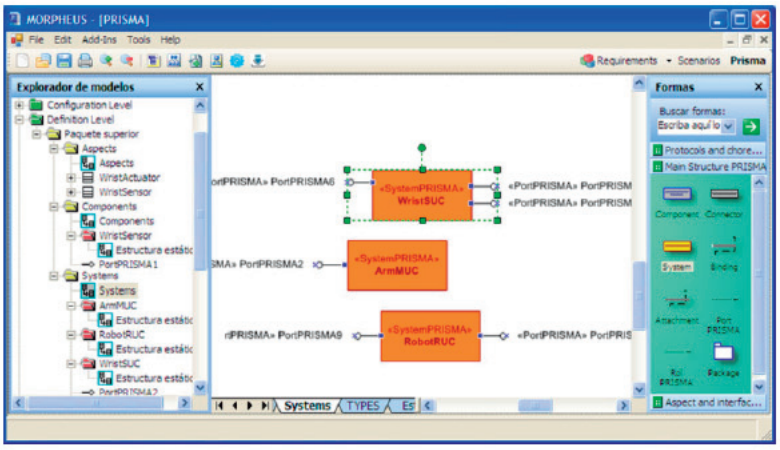


Fig. 12. What Architectural Editor looks like

4. Related works

Nowadays, MDD is an approach that is gaining more and more followers in the software development area, and lots of tools that support this trend have arisen. Nevertheless, none of the existing solutions can completely cover the capabilities of the MORPHEUS tool.

The Eclipse Modeling Framework (EMF) has become one of the most used frameworks to develop model-based applications. EMF provides a metamodelling language, called Ecore, that can be seen as an implementation of the Essential MOF language. Around EMF lots of related projects have grown that complement its modelling and metamodelling capabilities, such as OCL interpreters, model

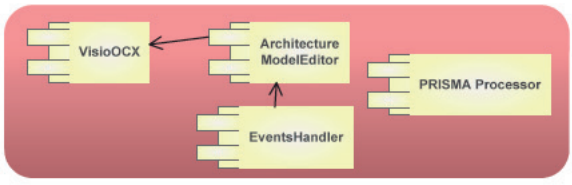


Fig. 13. Main elements of the Software Architecture Environment

transformation engines, or even tools able to automatically generate graphical editors, such as Graphical Modeling Framework (GMF [7]). The advantages are twofold: first they are usually quite mature tools, and second it is easy to interoperate with them by means of the XMI format. That is why the MORPHEUS tool has the *MOFManager* component: it allows us to reuse these tools as is the case of the OCL checker and the model transformations engine. Nevertheless, a solution completely based in EMF has also some important drawbacks. The main one is that, although it is not mandatory, this framework and its associated tools are fundamentally designed to deal with static models that do not change during run time. This factor makes frameworks like GMF completely useless for our purposes, because in MORPHEUS the requirements metamodel is populated with instances during its evolution and it is necessary to be able to synchronize them.

Other analyzed alternatives are the MS DSL Tools [3]. MS DSL tools are a powerful workbench that also provides modelling and metamodelling capabilities to automatically generate both code and graphical editors in Visual Studio. However, it exhibits the same weakness than the previous solution: it is basically designed to deal with models that do not evolve during time, so that, these models can only be modified during design time and not run time. Moreover, it lacks of the wide community that provides complementary tools to deal, check and analyze models, in comparison with the solution that is completely based on EMF. This disadvantage is also present in other tools, like the ones associated to Meta-CASE and Domain Specific Modelling techniques, such as MetaEdit+ [10].

5. Conclusions and further works

In this work a tool called MORPHEUS has been presented paying special attention to how it provides support to a MDD process, ATRIUM. It has been shown how each model can be described by using this tool and, specially, how traceability throughout its application is properly maintained. It is also worth noting the meta-modelling capabilities it has, providing automatic support to evolve the model as the metamodel is changed. The integration of an OCL checker is also interesting as it allows the user to evaluate the model using the properties he/she deems appropriate.

Several works constitute our future challenges. Although the tool is quite mature, we are considering the development of other functionalities, as for instance, a model checker of the software architecture or a report generator for the requirements environment. It is also among our priorities to deploy the tool in the next future as an open source project to be evaluated and used by the community.

Acknowledgments. This work is funded by the Dept. of Science and Technology (Spain) I+D+I, META project TIN2006-15175-C05-01 and by the UCLM, project MDDRehab TC20091111. This work is also supported by the FPU fellowship program from the Spanish government AP2006-00690.

References

1. Bézivin, J. (2004). *In search of a basic principle for model driven engineering*, Upgrade 5(2), pp. 21–24, 2004
2. Chung, L., Nixon, B. A., Yu, E., Mylopoulos J. (2000). *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing, Boston.
3. Cook, S., Jones, G., Kent, S., Cameron A. (2007) *Domain-specific development with Visual Studio DSL tools*, Addison Wesley Professional.
4. Czarnecki K., Helsen S. (2006) *Classification of Model Transformation Approaches*. *IBM Systems Journal*, 45(3), pp. 621-645.
5. Dardenne A., van Lamsweerde A., Fickas S., (1993) *Goal-directed Requirements Acquisition*, Science of Computer Programming, 20(1-2), pp. 3-50.
6. Eclipse Modeling Framework. http://www.eclipse.org/emf/
7. Eclipse Graphical Modeling Framework. http://www.eclipse.org/gmf/
8. GROWTH G3RD-CT-00794 (2003) *EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal*. European Project, 5th Framework Prog.
9. ISO/IEC JTC JTC1/SC7 N4098 (2008), *Software engineering-Software product Quality Requirements and Evaluation (SQuaRE) Quality model*.
10. Kelly S., Lyytinen K., Rossi M., METAEDIT+ *A fully configurable Multi-User and Multi-tool CASE and CAME Environment. Proc. of 8th International Conference on Advances Information System Engineering*, LNCS1080, Springer-Verlag, 1996, pp. 1-21.
11. Medini QVT, http://projects.ikv.de/qvt.
12. Montero F., Navarro E. (2007), *ATRIUM: Software Architecture Driven by Requirements*, *Proc. 14th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'09)*, IEEE Press, Jun. 2007, in press.
13. MORPHEUS (2009), http://www.dsi.uclm.es/personal/ElenaNavarro/research_atrium.htm
14. Navarro E., Cuesta C. E. (2008), *Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach*, *Proc. 2nd European Conference Software Architecture*, LNCS 5292, Springer Verlag, Sep. 2008, pp. 114-130.
15. Navarro E. (2007), *Architecture Traced from RequIrements applying a Unified Methodology*, PhD thesis, Computing Systems Department, UCLM.
16. Navarro E., Letelier P., Ramos I. (2007), *Requirements and Scenarios: playing Aspect Oriented Software Architectures*, *Proc. 6th IEEE/IFIP Conf. on Software Architecture*, IEEE Press, 2007, n. 23.
17. Navarro E., Letelier, P., Reolid, D., Ramos, I. (2007), *Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques*, *Proc. Information Systems and Development* (ISD'07), Springer US, Sep. 2007, pp. 167-179.
18. Navarro, E., Letelier, P., Mocholí, J.A., Ramos, I. *A Metamodeling Approach for Requirements Specification*. *Journal of Computer Information Systems*, 2006, 47(5): 67-77.
19. OMG (2006). Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01).
20. OMG (2005). Document ptc/05-11-01, QVT, MOF Query/ Views/Transformations.
21. OMG (2005). Software Process Engineering Metamodel (SPEM), ver. 1.1 formal/05-01-06.
22. Pérez, J., Ali, N., Carsí, J. Á., Ramos, I., (2006). *Designing Software Architectures with an Aspect-Oriented Architecture Description Language*, *Proc. 9th Int. Sym. on Component-Based Software Engineering* (CBSE 2006), Jun. 2006, pp. 123-138, Springer Berlin/Heidelberg.
23. Pérez, J., Navarro, E., Letelier, P. and Ramos, I. (2006). *A Modelling Proposal for Aspect-Oriented Software Architectures*, *Proc. 13th Annual IEEE Int. Conf. and Works. on the Engineering of Computer Based Systems* (ECBS'06), IEEE Press, Mar. 2006, pp. 32-41.
24. Selic, B. (2003). *The Pragmatics of Model-Driven Development.* IEEE Soft. 20(5), pp. 19-25.
25. Visio 2003 (2009), http://msdn microsoft.com/en-us/library/aa173161(office.11).aspx

The Journal of Systems and Software 83 (2010) 937–958

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Ambient-PRISMA: Ambients in mobile aspect-oriented software architecture

Nour Ali [a,*], Isidro Ramos [b,1], Carlos Solís [a,2]

[a] Lero – The Irish Software, Engineering Research Centre, University of Limerick, Limerick, Ireland
[b] Department of Information Systems and Computation, Universidad Politecnica de Valencia, Camino de Vera s/n, E-46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 30 November 2008
Received in revised form 23 November 2009
Accepted 8 December 2009
Available online 16 December 2009

Keywords:
Aspect-oriented software architectures
Mobility
Middleware
Distribution
Model driven engineering
Ambients

ABSTRACT

This work presents an approach called Ambient-PRISMA for modelling and developing distributed and mobile applications. Ambient-PRISMA enriches an aspect-oriented software architectural approach called PRISMA with the ambient concept from Ambient Calculus. Ambients are introduced in PRISMA as specialized kinds of connectors that offer mobility services to architectural elements (components and connectors) and are able to coordinate a boundary, which models the notion of location. Mobility of architectural elements is supported by reconfiguring the software architecture. This paper presents a metamodel that introduces ambients to design aspect-oriented software architectural models for mobile systems. The design of models is performed using an Aspect-Oriented Architecture Description Language. A middleware called Ambient-PRISMANET which maps the metamodel to .NET technology and supports the distributed runtime environment needed for executing mobile applications is also presented. In addition, a CASE Tool which allows users to specify the aspect-oriented architectural models in a graphical way and generate .NET code is provided. In this way, we explain how Ambient-PRISMA follows Model Driven Engineering. An example of an auction system is used throughout the article to illustrate the work.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

In the last few decades, the information society has undergone important changes which have increased the complexity of software development. New technologies have become part of our daily life and the Internet has been established as a framework for global knowledge. The software, the devices (PCs, laptops, PDAs, smart phones, etc) and people involved in current business processes are usually distributed, and mobile. As a result, the structure of software systems has become more complex due to the fact that these systems need to take into account new requirements such as adaptability, and security.

Software architecture is a discipline that focuses on the design and specification of overall system structure (Shaw and Garlan, 1996; Bass et al., 2003). It is considered to be the bridge between the requirements and implementation phases of the software life-cycle. The software architecture of a system describes its structure in terms of components (computational units), connectors (coordination units), and configurations (connection of components and connectors) (Medvidovic and Taylor, 2000). Reconfiguration is the change of the system structure. A kind of reconfiguration that can be found in distributed software systems is mobility. For example, Carzaniga et al. (1997) define code mobility as the capability to reconfigure dynamically, at runtime, the binding between the software components of the application and their physical location. Thus, code mobility causes the structure of a distributed system to change by creating and removing bindings (or connections) between software components and physical locations.

Aspect-Oriented Software Development (AOSD) (Filman et al., 2004) is also a technique that reduces complexity by increasing reusability, flexibility, and maintainability across the software development process. AOSD modularizes crosscutting concerns in separate entities called aspects (Kiczales et al., 2001). Distribution and mobility have been identified as crosscutting concerns; separating them in aspects increases their reusability, and decreases maintenance costs (Lobato et al., 2004; Lopes, 1997; Soares and Borba, 2002). For example, Lopes compared the code of nine case studies implemented in plain Java with the code obtained in the DJ aspect-oriented approach (Lopes, 1997). The results obtained were that in four of the case studies the number of lines achieved in plain Java were the same as the ones achieved in DJ, and five of the case studies the number of lines achieved in DJ were reduced in comparison with the plain Java code. The results demonstrate that distributed applications can benefit from AOSD as the lines of code can be reduced and the distribution concerns are well localized for maintenance tasks.

Research works which integrate AOSD and software architecture have been proposed (Chitchyan et al., 2005; Cuesta et al.,

* Corresponding author. Tel.: +35 3 61 233799.
E-mail addresses: Nour.Ali@lero.ie (N. Ali), iramos@dsic.upv.es (I. Ramos), Carlos.Solis@lero.ie (C. Solís).
[1] Tel.: +34 96 387 73 50.
[2] Tel.: +35 3 61 233799.

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.12.009

Ali, N., Ramos, I., Solís, C. Ambient-PRISMA: Ambients in Mobile Aspect-Oriented Software Architecture. Journal of Systems and Software 83. 2010.

2005). Introducing aspects in software architecture improves the modularization of crosscutting concerns that cannot be reflected in software architecture concepts (such as components, or connectors) (Batista et al., 2006)). This facilitates the evolution of software systems (Tarr et al., 1999)), and conserves the traceability of aspects across the software life-cycle (Baniassad et al., 2006).

Currently, one of the existing challenges is to allow code generation from software architecture. Model Driven Engineering (MDE) (Schmidt, 2006) (also known as Model Driven Development) is a software development approach based on transformations between models. MDE proposes the development of software by using models that describe a system in a technology-independent way. These models can be transformed into technology dependent models, which describe a system based on a specific technology. Two main approaches exist for supporting MDE through tools: the Model Driven Architecture (MDA) proposed by the Object Management Group (OMG) (Object Management Group, 2003), and the Software Factories proposed by Microsoft (Greenfield et al., 2004).

This paper presents an approach for developing distributed and mobile applications called Ambient-PRISMA. Ambient-PRISMA enriches an aspect-oriented software architecture approach called PRISMA with the ambient concept inspired from Ambient Calculus (AC) (Cardelli and Gordon, 1998; Cardelli, 1998). An ambient is a bounded place where computation happens such as a phone, a PC, a folder or a network. AC provides an explicit and abstract way for modelling locations, as well as a realistic mobility model that supports mobile computation (mobile code) and mobile computing (mobile devices) in a unique way.

AMBIENT-PRISMA introduces ambients as new kinds of architectural elements which define a bounded place where other architectural elements reside and are coordinated with the exterior of the boundary. Ambients can be hierarchically organized, conforming to a tree structure that is used to model distributed systems hierarchies. Architectural elements (including ambients) can move by entering and exiting ambients. Thus, mobility causes software architecture to reconfigure which involves removing mobile architectural elements from their originating ambients and adding them to their destination ambients, as well as creating and deleting connections among architectural elements. The functionality (behaviour) of an ambient is defined through mobility, coordination and distribution aspects.

Ambient-PRISMA follows MDE. It provides a metamodel for designing aspect-oriented software architectural models of mobile systems in a technology-independent way, a middleware called Ambient-PRISMANET, and a CASE Tool. The Ambient-PRISMANET middleware maps the metamodel to .NET technology and supports the distributed runtime environment needed to execute mobile applications. It also provides reconfiguration mechanisms needed for supporting mobility. The CASE Tool allows users to specify the aspect-oriented architectural models in a graphical way and generate .NET code of mobile applications.

The structure of the paper is as follows: Section 2 gives an overview of the PRISMA approach. Section 3 explains an Auction System example used to illustrate the work presented in this paper. Section 4 explains the characteristics of AMBIENT-PRISMA. Section 5 defines the AMBIENT-PRISMA metamodel basing on the PRISMA one. Section 6 shows how AMBIENT-PRISMA software architectural models can be defined. Section 7 introduces the AMBIENT-PRISMA-NET middleware, which permits the execution of AMBIENT-PRISMA models. Section 8 presents the AMBIENT-PRISMA Case tool that is used for modelling, and executing distributed and mobile applications by means of Model Driven Engineering techniques. Section 9 presents related work in aspect-oriented software architectures and mobility. Finally, conclusions and future work are presented in Section 10.

2. PRISMA

PRISMA (Pérez et al., 2008) is an approach that includes a metamodel (Pérez et al., 2005), an Aspect-Oriented Architecture Description Language (Pérez et al., 2006), and a tool for developing software systems from their aspect-oriented software architectures. PRISMA integrates the AOSD and the Component-Based Software Development (CBSD) (Szyperski, 2002) for defining software architectures. PRISMA uses AOSD to separate crosscutting concerns (safety, coordination, etc.) of architectures in aspects.

In PRISMA, there are two kinds of architectural elements: components and connectors. A component captures the logic of software systems and does not act as a coordinator. A connector acts as a coordinator among other architectural elements. An architectural element can be seen from an external and an internal view. In the external view (also known as black box view (Miles and Hamilton, 2006), see Fig. 1), an architectural element encapsulates its functionality and publishes a set of services that are offered to the rest of the architectural elements. These services are grouped into interfaces that are published through ports.

In the internal view (also known as white box view (Miles and Hamilton, 2006)), an architectural element is seen as a prism, where each side is an aspect. In this way, architectural elements are represented as a set of aspects and weaving relationships among them (see Fig. 1). An aspect defines the properties and behaviour of an architectural element from a concrete crosscutting concern. The behaviour defined in aspects specifies how and when services are executed. These services can be part of interfaces. In PRISMA, the difference between a component and a connector is that a component cannot have a coordination aspect, and a connector cannot have a functional aspect but must have a coordination

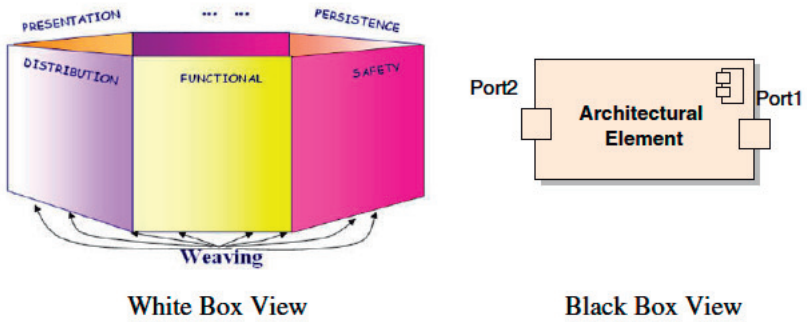


Fig. 1. A PRISMA architectural element.

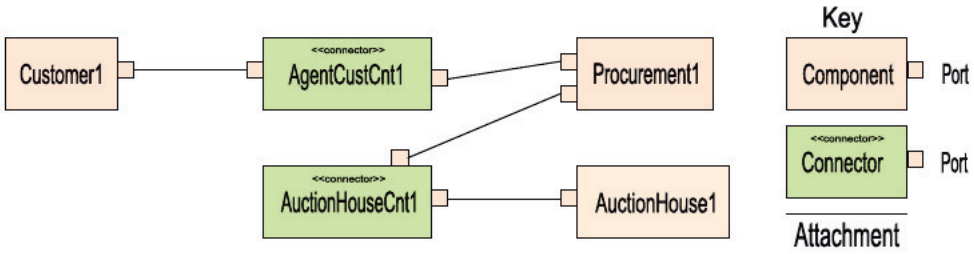


Fig. 2. Software architecture of the auction system example.

aspect. The aspects of an architectural element are synchronized through *weaving* relationships. A weaving relationship indicates that a service of an aspect can trigger a service in another aspect to execute.

The white box and the black box views are connected by means of interfaces. The same interfaces published through ports are used to specify the behaviour defined in aspects. Consequently, a service request that arrives to a port of an architectural element is processed by one of its aspects.

Architectural elements are reusable and unaware of each other. Components and connectors are connected in the software architecture through attachments. Each attachment is defined by connecting a component port with a connector port (see the lines in Fig. 2).

3. EXAMPLE: an auction system

The example used throughout the paper consists of an electronic Auction System which offers its products that can be bid and sold (This is a simplified version of a case study used in (Ali, 2008)). These kinds of auctions have been developed in works such as Sandholm and Huai (2000). The Auction System allows its customers to send mobile agents in order to search for interesting products and bid on their behalf. In this way, the customers can send the agents and disconnect from the network. This reduces network latency and traffic, and agents can respond quickly to changes in the auction.

The Customer gives descriptions of his/her interesting products to a Procurement agent and moves it to the site of the Auction House which offers products for bidding. In the auction site, the Procurement agent searches for products that fulfill the descriptions of the Customer. Once, the Procurement finds a product that fulfils certain characteristics it notifies the Customer. If the Customer confirms that it has found the appropriate product, the Procurement moves back home.

Fig. 2 shows the software architecture configuration of the example. The Customer1 and the Procurement1 are component instances that are coordinated through an *AgentCustCnct1* connector instance. The Procurement1 component instance is coordinated with the AuctionHouse1 component instance through the AuctionHouseCnct1 connector instance. For abbreviation, we will only use the terms component and connector instead of component and connector instances when we refer to the example.

4. Ambient-PRISMA overview

Ambient-PRISMA is the result of including distribution and mobility primitives in PRISMA. These distribution and mobility primitives are inspired from Ambient Calculus (AC) (Ali, 2008; Cardelli, 1998). AC introduces the concept of ambient which is a bounded place where computation happens. An ambient can contain other ambients that have running computations. Thus, an ambient can be anything with a boundary such as a laptop, a web page, a folder, etc. Ambients have capabilities for moving by crossing boundaries of either their parent ambient or their sibling ambients. The enter capability allows an ambient to enter a sibling ambient and the exit capability allows an ambient to exit its parent ambient.

In AC, the boundary of an ambient separates what is in the ambient from what is not. In PRISMA, software architectures are described by means of components or connectors, neither capture the notion of boundary or location that ambients are meant to provide. For instance, in the Auction System example, there is no primitive that represents where these components and connectors are located nor a primitive responsible for the way these move. The objective is to allow PRISMA architectural elements to be distributed across several locations as well as to allow them to move.

To introduce the ambient concept as an architectural element in PRISMA, the PRISMA primitives have to be enriched. An ambient has been introduced by extending the concept of connector. An ambient cannot be a component because it does not define the logic of the software system. An ambient locates architectural elements inside its boundary and coordinates them with elements outside it. This functionality can be achieved by defining an ambient with a coordination aspect.

Elements that are located in an ambient cannot be directly connected to elements of other ambients because boundaries exist. In addition, an architectural element that needs to move from one ambient to another has to be ensured that if it leaves, the destination ambient will receive it. In this way, an ambient needs coordination mechanisms in order to control what can enter or exit its boundary. However, in PRISMA connectors cannot locate and manage other architectural elements and their connections. Therefore, ambients are introduced in PRISMA as specialized kinds of connectors that are able to coordinate a boundary which models the notion of location and provides mobility support to other architectural elements.

Fig. 3 shows the graphical representation of an ambient. Ambient-PRISMA ambients have a boundary, an AOSD view, and a CBSD

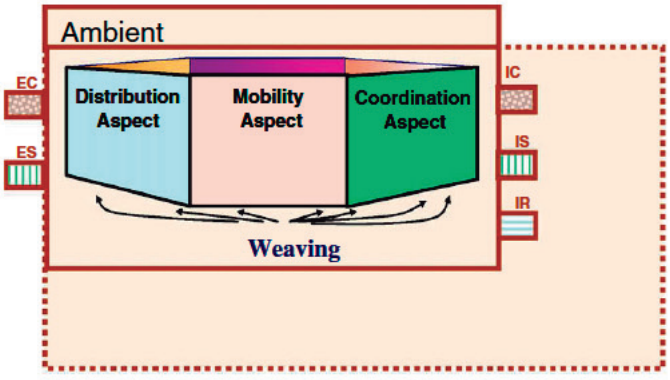


Fig. 3. The graphical notation of an Ambient-PRISMA ambient.

view. A boundary of an ambient represents a place where PRISMA components, connectors, and their attachments are located. The boundary separates what is in an ambient from what is out of it. As a result, mobility of an element is performed by crossing a boundary of an ambient and entering a boundary of another ambi ent. A mobile architectural element can only enter a sibling ambi ent or exit its parent ambient after its origin and destination ambients allow it to move. The boundary of an ambient is repre sented with a dotted line in order to indicate that the functionality of an ambient does not depend on the functionality of its internal architectural elements.

The AOSD (AOSD) view describes an ambient as a composition of different aspects that are weaved together (white box view). The designer can define different aspects for an ambient depending on the requirements of the system e.g., a security aspect when security properties are needed. However, an ambient has at least three aspects: a coordination aspect, a mobility aspect and a distri bution aspect. The mobility aspect of an ambient provides services for allowing architectural elements (including ambients) to move in and out of them. The coordination aspect receives calls from external architectural elements (distributed calls) and redirects them to corresponding architectural elements located in an ambi ent (internal architectural elements). It also receives calls from its internal architectural elements and redirects them to the exterior. The distribution aspect stores the name of the parent ambient in order to give awareness of the position of an ambient in the ambi ent hierarchy and allows internal architectural elements to consult routes of other architectural elements.

The CBSD view describes an ambient as a black box where it communicates with other architectural elements by using ports through which invocations of services are sent and received. An ambient is a connector which coordinates architectural elements located in its boundary with the exterior. Therefore, an ambient has ports which publish services for architectural elements located in its boundary (the prefix In is used to name these ports), and other ports for those located outside its boundary (the prefix E is used to name these ports).

The designer can define ambients with many ports. However, each ambient has five predefined ports for offering and requesting services: two for mobility called InCapabilitiesPort and ECapabili ties Port, two for coordination called EServicesPort and InService sPort, and one for distribution called InRoutePort. As shown in Fig. 3, the InCapabilitiesPort, and ECapabilitiesPort ports are desig nated by dots. The ECapabilitiesPort port, marked with *EC*, allows the ambient to offer and require the mobility services outside the ambient. The InCapabilitiesPort port, marked with *IC*, allows an ambient to offer the mobility services to architectural elements in side the ambient. The EServicesPort, and InServicesPort ports are designated by stripes. These ports allow architectural elements that are inside the ambient to offer and request services to and from architectural elements that are outside the ambient. There fore, the InServicesPort port, marked with *IS* in Fig. 3, is connected with internal architectural elements and the EServicesPort port, marked with *ES*, is connected with architectural elements outside the ambient. The InRoutePort port, marked with *IR*, allows internal architectural elements to consult routes of any architectural elements.

In AC, an ambient can be nested within other ambients. Ambi ents in Ambient PRISMA can be located inside the boundary of other ambients. This allows Ambient PRISMA software architec tures to describe topologies of locations where hierarchies of dis tributed and mobile systems can be modelled. An ambient hierarchy forms a tree structure which has a root ambient. In Ambient PRISMA, this root ambient must exist in each software architecture, and it is called *Root*.

For example Fig. 4 shows a hierarchy of ambients where the *Root* ambient locates two ambients called *ClientSite* and *AuctionSite*. The *ClientSite* ambient locates the *Customer1* component and the *AgentCustCnct* connector. The *AuctionSite* ambient locates the *Pro curement1* component, the *AuctionHouseCnct1* connector and the *AuctionHouse1* component.

An ambient controls what can pass in or out of it. For example in Fig. 4, when the *AgentCustCnct* connector sends a service request to the *Procurement1* component, the request has to be synchronized by the *ClientSite* ambient in order to allow it to be sent to the *Auc tionSite* ambient, and the *AuctionSite* ambient sends the service re quest to the *Procurement1* component. Also, if the *Procurement1* component needs to move from the *AuctionSite* ambient to the *Root* ambient, then the *AuctionSite* ambient has to coordinate the *Pro curment1* component with the *Root* ambient in order to allow the *Root* ambient to accept the *Procurement1* component. In this way, the *Procurement1* component crosses the boundary of the *Auction Site* ambient into the *Root* ambient.

Attachments connect ECapabilitiesPort (EC in Fig. 4) port of ambients to the InCapabilities (IC in Fig. 4) of their parent ambient. Attachments can also connect an EServicesPort port of an ambient

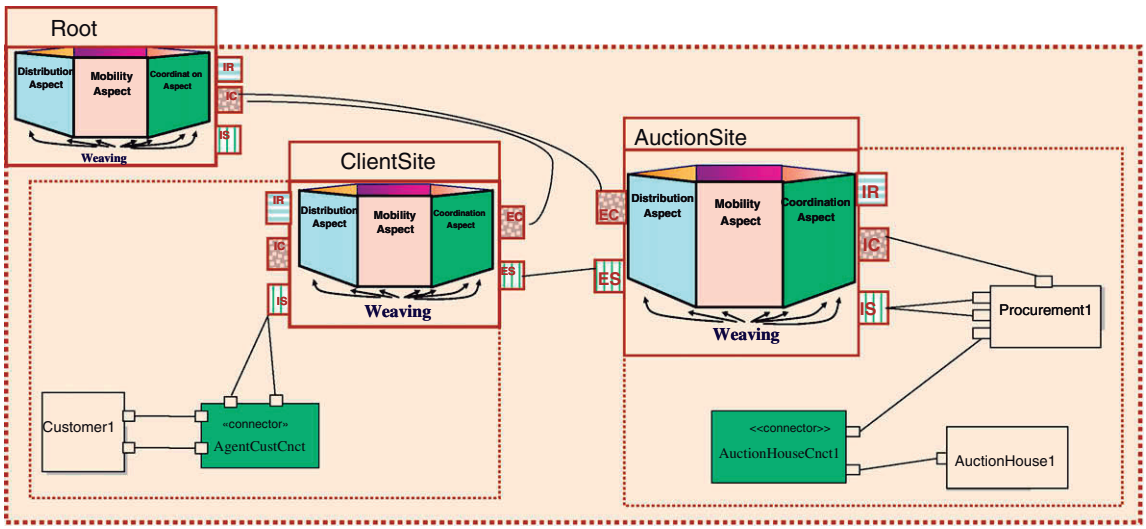


Fig. 4. A possible configuration of the auction mobile agent case study where the Procurement1 component is in the AuctionSite ambient.

to an EServicesPort port of its sibling ambient. For example, the ECapabilitiesPort port of the *AuctionSite* ambient is connected to the InCapabilities of the *Root* ambient (its parent ambient) through an attachment. In addition, the EServicesPort port of the *AuctionSite* ambient is connected to the EServicesPort port of the *ClientSite* ambient (its sibling).

Attachments also connect ports of architectural elements lo cated in an ambient to the InCapabilities port or the InRoutePort port of their parent ambient. For example in Fig. 4, the *Procure ment1* component has a port connected to the InCapabilitiesPort port of the *AuctionSite* ambient because the *Procurment1* compo nent is a mobile architectural element.

Elements of different ambients do not have direct attachments between each other. An attachment only connects elements of the same ambient. For example in Fig. 4, an attachment between a port of the *ClientSite* ambient and a port of the *AuctionSite* ambient is possible because the *ClientSite* and the *AuctionSite* are siblings (they share the same parent ambient). An attachment is also possible be tween the *Customer1* component and the *AgentCustCnct* connector because both of them are in the *ClientSite* ambient. However, the attachment which formed the communication channel in Fig. 2 to connect the *Procurement1* component to the *AgentCustCnct* con nector is not possible in Fig. 4. This channel is broken down into an attachment that connects the *InServicesPort* port of the *AuctionSite* ambient and a port of the *Procurement1*, an attachment that con nects the *EServicesPort* port of the *AuctionSite* ambient and the *ESer vicesPort* port of the *ClientSite* ambient, and an attachment that connects the *AgentCustCnctr* connector to the *InServicesPort* port of the *ClientSite* ambient.

When an architectural element moves from one ambient to an other, the attachments associated to an ambient are reconfigured in order to provide the mobile architectural element with the ser vices it needs. This reconfiguration is supported at runtime by the Ambient PRISMANET middleware (see Section 7.3). For example, when the *Procurement1* component in Fig. 4 moves from the *Auc tionSite* ambient to the *ClientSite* ambient, the attachments associ ated to the *Procurement1* are reconfigured (see Fig. 5): the attachments that connect the *Procurement1* to the InCapabilities and the InServicesPort port of the *AuctionSite* ambient are removed, the attachment that connected the *Procurement1* component to the *AuctionHouseCnct1* connector is also removed, attachments be tween the *Procurement1* and the InServicesPort and the InCapabil ities ports of the *ClientSite* are created, an attachment that connects the *AuctionHouseCnct1* connector and the InServicesPort of the *Auc tionSite* ambient is created, and two attachments that connect the *Procurement1* to the *AgentCustCnct* are created.

5. Ambient-PRISMA metamodel

Ambient PRISMA extends the PRISMA metamodel with con cepts and properties explained in Section 4. These concepts have been included by adding new metaclasses and relating them with the ones defined in PRISMA. The metaclasses and relationships de fine the structure and the information that is needed to describe Ambient PRISMA architectural models. In addition, the Ambient PRISMA metamodel includes OCL expressions (Warmer and Klep pe, 2003) in order to define constraints that must be satisfied by an Ambient PRISMA architectural model. In the following, the Ambient PRISMA concepts are presented.

5.1. Ambients

An ambient architectural element is defined in the metamodel as a specialized kind of connector (see Fig. 6) by the *Ambient* meta class which extends the *Connector* metaclass. The ambient inherits all the properties of a connector including the invariant called *coor dinationAspect* which specifies that a connector must import an as pect whose concern is *coordination* i.e., a Coordination aspect.

Moreover, all ambients have a unique Coordination Aspect (see Fig. 6). The Coordination Aspect of ambients is called ACoordina tion. As a result, the *Ambient* metaclass has an invariant (an OCL constraint) called *ambientCoordination*, which specifies that an ambient only has a Coordination Aspect and whose name is *ACoor dination*. The *mobilityAspect* invariant specifies that an ambient must import an aspect whose concern is mobility, and the name of this aspect is *MobilityAspect*. The *distAspAmb* invariant specifies that architectural elements including ambients must import an as pect whose concern is distribution and that this aspect must have an attribute called *location*.

In addition to the constraints shown in Fig. 6, an ambient should satisfy other OCL constraints concerning weaving and ports. In the following they are explained:

- The *weavingAmb* invariant specifies that an ambient needs to have at least a weaving between its *MobilityAspect* aspect and a distribution aspect.

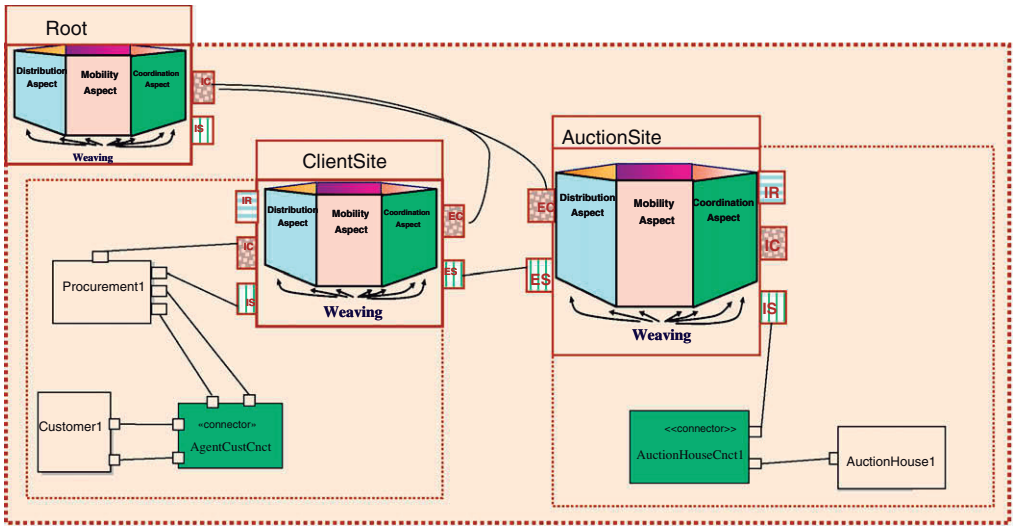


Fig. 5. Procurement1 component in the ClientSite ambient.

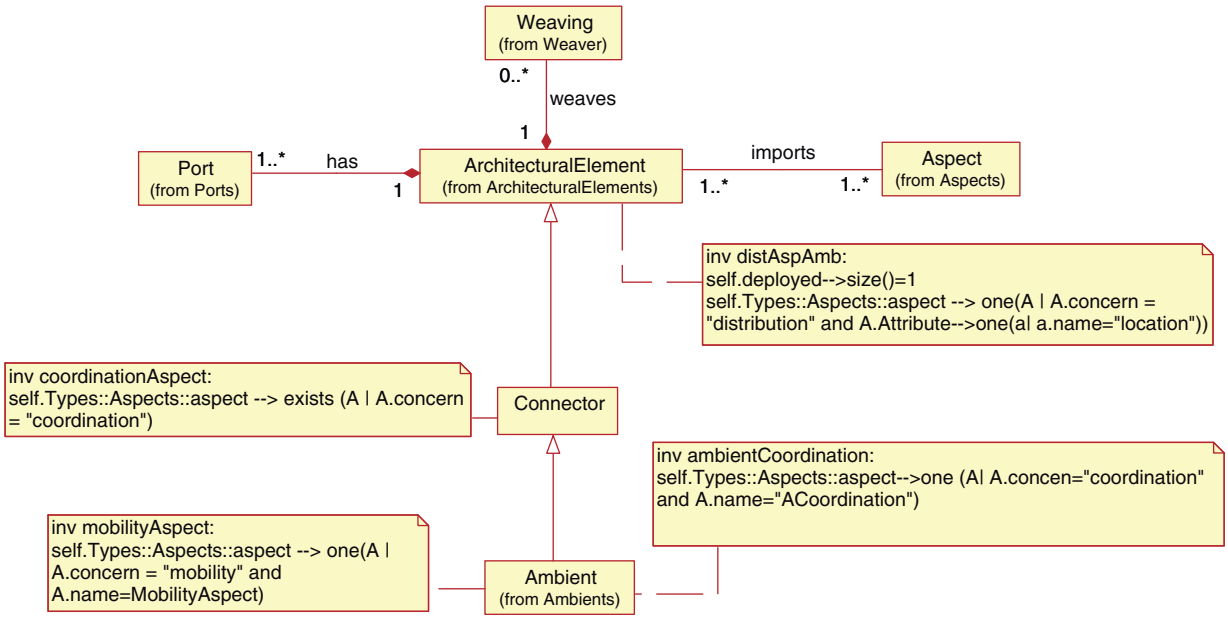


Fig. 6. An ambient in the Ambient-PRISMA metamodel.

- The *numPort* invariant specifies that an ambient must have at least five ports.
- The *ICport* invariant specifies that an ambient must have a port called InCapabilitiesPort.
- The *ECport* invariant specifies that an ambient must have a port called ECapabilitiesPort.
- The *ISport* invariant specifies that an ambient must have a port called InServicesPort.
- The *ESport* invariant specifies that an ambient must have a port called EServicesPort
- The *IRport* invariant specifies that an ambient must have a port called InRoutePort

5.2. Kinds of ambients

In Ambient PRISMA, we define three kinds of ambients:

- *Site Ambients:* They represent physical devices; that is, Site amb ients have a physical address. A PC or a PDA are examples of ambients that can be modelled as Sites. Site ambients can only contain Components, Connectors or Group ambients. For exam ple, the *ClientSite* and the *AuctionSite* of Fig. 4 are of type Site.
- *Group Ambients*: They represent a collection of architectural ele ments. They can be used for grouping a set of architectural ele ments that share specific security policies, or for moving groups of related architecture elements. A Group ambient can lodge other Group ambients, Components or Connectors. For example, *MobileAmb* in Fig. 17 is a Group Ambient that moves architec tural elements together.
- *Virtual Ambients*: They represent domains which can lodge other Virtual or Site ambients. Virtual ambients model networks or virtual spaces where sites are located. A Virtual ambient that consists of a collection of Sites represents the boundary of a net work (e.g. a LAN). The modelling of a Virtual ambient hierarchy can represent the hierarchy of LANs to form a Wide Area Net work (WAN). In Ambient PRISMA there is always a default Vir tual Ambient that represents the root of an ambient hierarchy called Root (see Fig. 4).

Fig. 7 shows that the three kinds of ambients inherit the prop erties of the *Ambient* metaclass. Each kind of ambient is repre sented by a metaclass and each one has a new service to create new kinds of ambients. Also, each metaclass has OCL invariants associated to it in order to differentiate each kind of ambient from the others.

The *Site* metaclass has the following OCL invariants:

- The *distSite* invariant specifies that the distribution aspect of a site ambient must have an attribute called *physicalLocation* and that its data type is *loc*. The *loc* data type is needed in order to model the different physical locations that form a specific dis tributed system.
- The *childSite* invariant specifies that a site ambient must not have a virtual ambient as one of its children.

The *Virtual* metaclass has the following OCL invariants:

- The *inmobileRoot* invariant specifies that the distribution aspect of an ambient called *Root* must have a constant attribute called *Location*. This invariant indicates that Root ambients are not mobile architectural elements (their location does not change).
- The *childVirtual* invariant specifies that a virtual ambient must only have site ambients or virtual ambients as children.

The *Group* metaclass has the following OCL invariants:

- The *childGroup* invariant specifies that a group ambient must not have a virtual ambient or a site ambient as a child.

5.3. Ambients in the software architecture

In this section, we will explain how an ambient is related to other software architectural concepts. The *PRISMAArchitecture* metaclass that represents a PRISMA software architecture has an aggregation relationship with the *Ambient* metaclass (see Fig. 8). In Ambient PRISMA, an architecture is defined by at least an ambi ent. Since an ambient is reusable, it can be used by many software architectures. In this way, an ambient is a first class entity of an Ambient PRISMA architectural model.

The *PRISMAArchitecture* metaclass has the following two con straints that ensure the correctness of models which follow the metamodel:

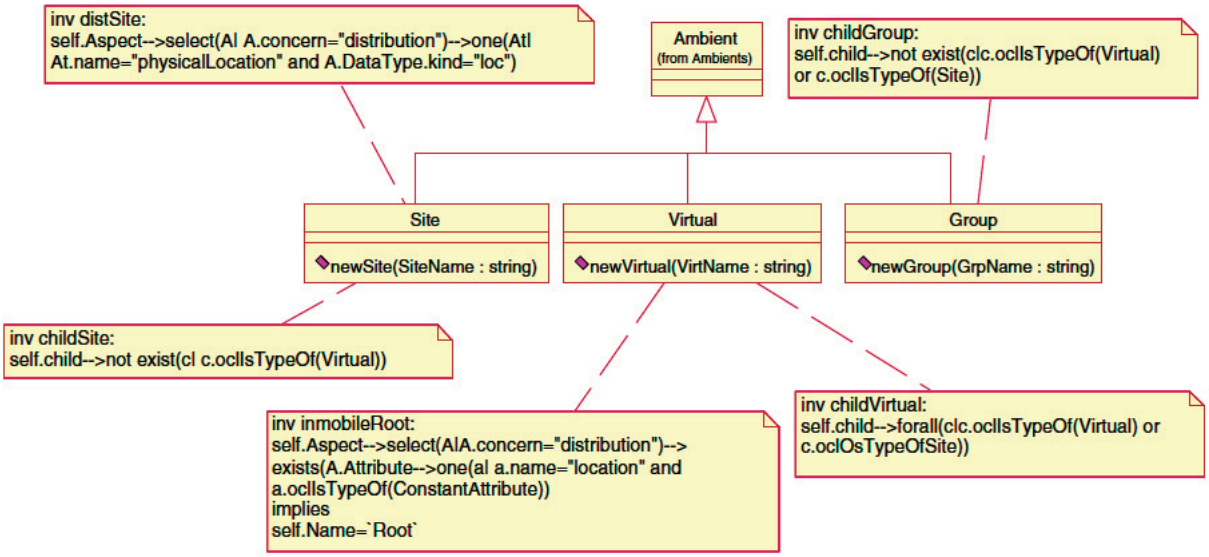


Fig. 7. Kinds of ambients.

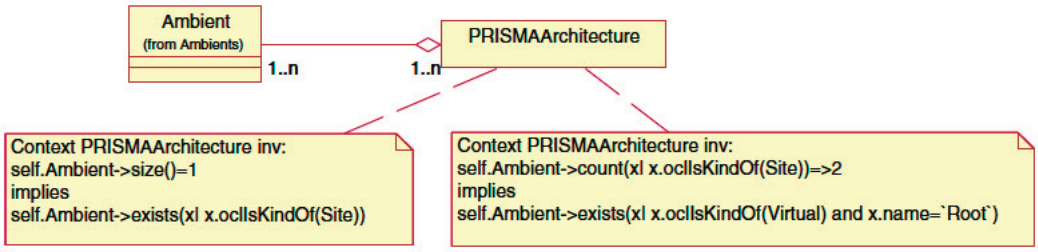


Fig. 8. The *AmbientPRISMAArchitecture* package of the Ambient-PRISMA metamodel.

- ≪If an architectural model only defines one ambient, then this ambient must be a Site ambient.≫
- ≪An architectural model that has two or more Site ambients, must have a Virtual ambient called Root.≫

Fig. 9 shows that ambients locate architectural elements (which can be components, connectors, or other ambients), and attachments. These attachments are those that connect an ambient with architectural elements located in it, and attachments that connect architectural elements of an ambient. As a result, the *Ambient* metaclass has two aggregation relationships: one with the *ArchitecturalElement* metaclass (this metaclass is the superclass of the component, and connector metaclasses which represent components and connectors, respectively), and one with the *Attachments* metaclass. The aggregation between the *Ambient* metaclass and the *ArchitecturalElement* metaclass has the *child* role and the *deployed* role. The *child* role indicates that an ambient can have from 0 to many architectural elements located in it. The *deployed* role indi-

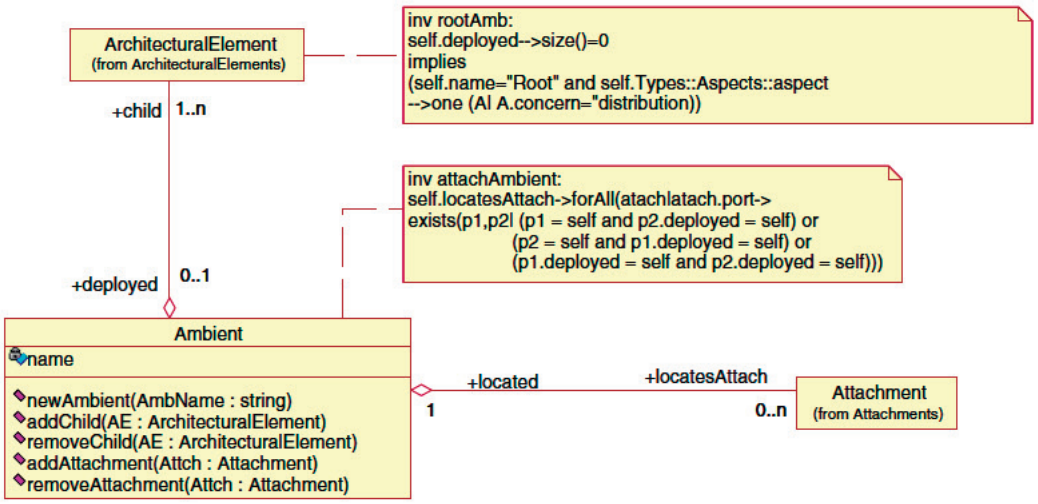


Fig. 9. Ambients locate architectural elements and attachments.

cates that an architectural element can be located in 0 or in only one ambient. The aggregation between the *Ambient* metaclass and the *Attachments* metaclass has the *located* role and the *locatesAttach* role. The *locatesAttach* role indicates that an ambient can have 0 or many attachments located in it. The *located* role indicates that an attachment can only be located in one ambient.

An OCL invariant is associated to the *ArchitecturalElement* metaclass:

- The *rootAmb* invariant specifies that the only architectural element that is not deployed (or located) in any ambient (the deployed role is equal to 0) is called the Root, and that Root must import an aspect whose concern is distribution.

The *Ambient* metaclass has an attribute called *name* which specifies the name of an ambient and five services:

- *newAmbient* service creates a new ambient by providing the name of the ambient as a parameter.
- *addChild* service adds a new architectural element in the boundary of an ambient.
- *removeChild* service removes an architectural element that is located in the boundary of an ambient.
- *addAttachment* service adds an attachment that connects an ambient to one of its children or an attachment between two children of the same ambient.
- *removeAttachments* removes an attachment located in the boundary of an ambient.

The *attachAmbient* invariant specifies that attachments located in an ambient are those that connect the ambient with the architectural elements located in its boundary or those that connect two architectural elements that are located in its boundary. Elements of different ambients cannot have attachments.

5.4. Attachments

The *Attachments* package of the PRISMA metamodel defines an attachment as a communication channel between a port of a component and a port of a connector. The *Attachments* package in Ambient-PRISMA is extended in order to include attachments that can connect ports of ambients to ports of components, connectors, or other ambients (see Fig. 10).

The association between the *Attachment* metaclass and the *Port* metaclass establishes that an attachment must connect two ports. The *Attachment* metaclass has three OCL constraints that determine which architectural elements ports can be attached. The OCL constraints specify the following:

- ≪One of the ports of an attachment must belong to a component and that the other must belong to a connector OR one of the ports must belong to an ambient and that the other port must belong to any architectural element.≫
- ≪An attachment cannot connect a port of a component to a port of another component.≫
- ≪The *InServicesPort* port of a virtual ambient can only be attached to a port of another Virtual ambient or a Site ambient.≫

The *Attachment* metaclass has an attribute called *name* for storing the name of an attachment. The *Attachment* metaclass also has four more attributes to specify how many instances of the attachment can be attached to the port of each architectural element instance that an attachment connects. The *card_min_port_A1* and *card_min_port_A2* attributes specify the minimum number of attachment instances that must be connected to one instance of A1 and A2 architectural elements through their ports, respectively. The *card_max_port_A1* and *card_max_port_A2* attributes specify the maximum number of attachment instances that must be connected to one instance of A1 and A2 architectural elements through their ports, respectively.

In Ambient-PRISMA, an architectural element is deployed in a single ambient. An ambient also has five predefined ports. As a result, the *Attachment* metaclass has three OCL invariants to predefine the values of the cardinality attributes. In the following, these are explained:

- The *intPortsAmbients* invariant specifies that if an attachment connects the *InServicesPort* port, the *InCapabilitiesPort* port, or the *InRoutePort* port of an ambient to a port of a child, the values of the cardinalities are cardmin_port_A1=0, cardmax_port_A1=n, cardmin_port_A2=0, and cardmax_port_A2=1.
- The *eSerPortAmbients* invariant specifies that if an attachment connects the *EServicesPort* port of an ambient to a port of an architectural element, the values of the cardinalities are cardmin_port_A1=0, cardmax_port_A1=n, cardmin_port_A2=0, and cardmax_port_A2=n.
- The *eCapPortAmbients* invariant specifies that if an attachment connects the *ECapabilitiesPort* port of an ambient to a port of an architectural element, the values of the cardinalities are cardmin_port_A1=0, cardmax_port_A1=1, cardmin_port_A2=0, and cardmax_port_A2=n.

6. Ambient-PRISMA architectural models

Architectural models must conform to the metamodel definition. Following the PRISMA approach (Pérez et al., 2008) architectural models are defined at two levels: at the type definition level and at the configuration level. At the type definition level interfaces, aspects and architectural elements types are defined. At the configuration level a topology of a software architecture is designed by instantiating architectural element types and their attachments. PRISMA provides an Aspect-Oriented Architecture Description Language (AOADL) (Pérez et al., 2006) which allows the definition of architectural models. Ambient-PRISMA uses this AOADL and extends it to include ambients.

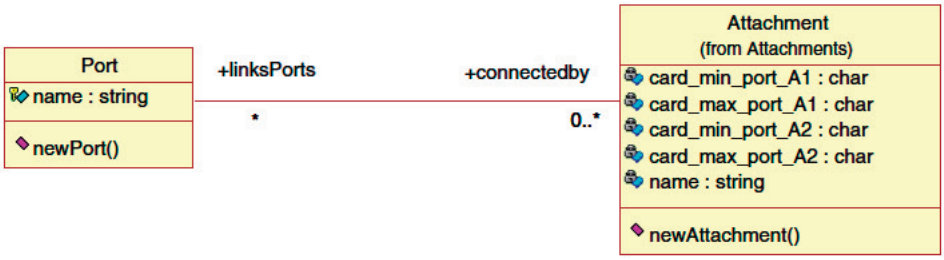


Fig. 10. The *Attachments* package of the Ambient-PRISMA metamodel.

In the following, we are going to explain how an ambient is defined at the type definition level. Afterwards, we will show how to define the configuration of an architecture with ambients.

6.1. An ambient type

An ambient at the type level has predefined aspects, weavings and ports. In the following, these are explained and then an example is specified:

6.1.1. Aspects

In PRISMA, aspects are not specified as part of an architectural element. Aspects are specified separately, and then architectural elements import the aspects required. The same happens with ambients. In the following, the specification of an ambient's aspects are explained.

6.1.1.1. ACoordination aspect. In Section 4, we discussed that an ambient needs to coordinate the architectural elements located in an ambient with architectural elements located outside an ambient. This is possible thanks to the Coordination Aspect of an ambient which is called *ACoordination*. The ACoordination aspect allows requested or provided services of architectural elements in an ambient to be redirected to the exterior of an ambient.

For example, in Fig. 5, the *Procurement1* component can communicate with the *Customer1* component through the *AgentCustCnctr* connector in order to send the description of a potential product that the Customer maybe interested in buying. Also, the *Customer1* component has to reply to the *Procurement1* component by telling it if he/she is interested or not in buying this product. As the *Customer1* and the *Procurement1* components are in different ambients, the messages that they interchange must pass through their ambients until reaching them. In this case, this is achieved through the *ACoordination* aspect of both the *ClientSite* ambient and the *AuctionSite* ambient.

Since all services of the internal and external elements of an ambient must pass through its *ACoordination* aspect, an ambient must control all the interfaces of its internal elements. Therefore, the *ACoordination* aspect uses the *ICall* interface which is specified in Fig. 11. This allows an ambient not to change its interfaces each time an architectural element exits or enters an ambient. A *call* service invocation uses its parameters to receive the signature of a service requested or provided by one of the architectural elements located in an ambient. The *call* service has three parameters: *Name* which stores the name of the original service, *ParamsList* which is a list that stores the parameters of the original service and *ParamType* which is a list that stores whether the parameters of the original service are *input* or *output*. *Input* parameters are those that are required for executing the service; whereas *output* parameters are those that are generated by the execution of the service.

The *ACoordination* aspect specifies the behaviour of the call service (see Fig. 12). It specifies that the *call* service is an in/out i.e. it is required and provided. The *ACoordination* coordination aspect is a generic aspect that is reused by all ambients.

6.1.1.2. MobilityAspect aspect. Since an ambient controls what moves in and out of its boundary, one of its responsibilities is to offer services that allow architectural elements inside its boundary to move. These mobility services are specified in the *MobilityAspect* aspect. Since the *MobilityAspect* gives a behaviour that is common

```
Interface ICall
   call(input Name: Service, input ParamsList[]: Parameter,
        input ParamType[]: ParameterKind);
End_Interface ICall
```

Fig. 11. Specification of the ICall interface.

```
Coordination Aspect ACoordination using ICall

Services
   ...
   in/out call(input Name: Service, input ParamsList[]: Parameter,
               input ParamType[]: ParameterKind);
```

Fig. 12. Partial specification of the ACoordination aspect.

Table 1
Description of the *ICapability* interface services.

Service	Description
startMovement	Given a name of an architectural element, the ambient prepares an architectural element to move by updating the attachments that connect the architectural element.to others
exit	Given a name of an architectural element, an architectural element is allowed to exit its parent ambient
enter	Given a name of an architectural element and a name of an ambient, an architectural element is provided to enter the ambient with the given name, if the ambient is a sibling of the architectural element
finishMovement	Given a name of an architectural element, an ambient creates the needed attachments in its boundary
changeLocation	An ambient returns its name to an architectural element that has just arrived to it
accept	Given a receptor ambient, an origin ambient, an architectural element, and the list of attachments connected to the architectural element, the receptor ambient returns true to the origin ambient if it accepts to receive the architectural element
copy	Given a name of an architectural element, a parent ambient creates a new instance with the same state as the architectural element and returns true when the copy has been created

to all ambients, this behaviour is defined in an aspect that is imported and reused by all ambients.

For example, the *Procurement1* component in Fig. 4 is a mobile component. In order to be able to move, it requests the mobility services that the *MobilityAspect* aspect of its parent ambient provide, in this case the *AuctionSite* ambient. The *MobilityAspect* of the *AuctionSite* asks the destination ambient of the Procurement1 component whether it accepts it or not. In this way, the *AuctionSite* acts as a coordinator between the *Procurement1* and its destination ambient.

The *MobilityAspect* aspect uses the *ICapability* interface (see Table 1) in order to specify the behaviour of the services which it publishes (see Fig. 13). Most of the services that the *MobilityAspect* aspect provides are transactional services that reconfigure the software architecture, i.e., services that change the configuration of the software architecture. In this way, if an error occurs during the mobility process, the state of the software architecture can be recovered to its previous state before mobility.

For example, the *MobiltyAspect* aspect specifies that the *exit* service is a transactional service (see Line 01, Fig. 13). The transaction checks whether the architectural element that needs to exit is a child of the ambient that imports the aspect or not. This is checked through the *isChild* service (see Line 03). If the requested architectural element is a child of the ambient, then the *ischild* service returns true through the *isChildOK* output parameter. If *ischild* returns true, then the transaction proceeds to execute *getParent* service (see Line 04). The *getParent* service returns the name of the parent ambient of the *MobilityAspect* ambient through the *Parent* output parameter. This is needed in order to indicate the destination of the exiting architectural element. Then, the moving process is performed through the *moving* service (see line 05). The *moving* service has two parameters: *Requested* and *Parent*. The *Requested* parameter indicates the name of the exiting (or moving) architectural element and the *Parent* parameter indicates the destination of the moving architectural element.

The *moving* service is a transactional service that consists of removing an element from the boundaries of an ambient: removing attachments between the element and the ambient, removing attachments between the element and other elements of the same ambient and creating attachments between the other elements that were connected to the element and the ambient.

6.1.1.3. Distribution aspects of ambients. A distribution aspect is responsible for specifying the location of an architectural element (all kinds of architectural elements have distribution aspects in Ambient-PRISMA not only ambients). The distribution aspect gives location-awareness to an ambient (or architectural elements) and allows it to take different decisions or strategies depending on its current location. An ambient is located in a hierarchy of ambients, therefore the distribution aspect has to specify its parent ambient. It also specifies the mobility strategies of an ambient, thus an ambient can be a mobile architectural element. The distribution aspect specifies whether an ambient is mobile or not. If an ambient is mobile, the aspect can also specify when and how the ambient (or architectural element) should move. Therefore, an ambient uses its distribution aspect for requesting mobility services from its current parent ambient, and therefore the ambient can change its position in an ambient hierarchy by changing its parent ambient.

Each distribution aspect stores the parent name of the ambient that imports it. Also, it allows other aspects and architectural elements of an ambient to consult locations and routes. Therefore, a distribution aspect uses the *IGetLocation* interface specified in Fig. 14 and the *IGetRoute* interface specified in Fig. 15.

Fig. 16 shows the specification of a distribution aspect called *ADist* that can be imported by an ambient. The *ADist* aspect uses both the *IGetLocation* and the *IGetRoute* interfaces (see line 01, Fig. 16). The *ADist* stores the name of the parent ambient of the ambient that imports it through the *location* attribute (see lines 02–04, Fig. 16). The location attribute is marked in bold because it is a keyword of Ambient-PRISMA. The *ADist* aspect specifies that the *getLocation* service and the *getRoute* service are of kind in/out i.e., first the aspect receives the request (*in*), then the *Valuation* assigns the value of the output parameter and finally the service sends the value of the output parameter (*out*) to the requester.

Although an ambient must have a distribution aspect, and must contain at least the specification shown in Fig. 16, a distribution aspect is not a generic aspect that is predefined. In this way, a distribution aspect is defined differently depending on the systems requirements. For example, the *ClientSite* and the *AuctionSite* are not mobile. Therefore, they can reuse the same distribution aspect. However, an ambient that needs to move, such as *MobileAmb* ambient shown in Fig. 17, cannot reuse the same distribution aspect as *ClientSite*. As a result, the distribution aspect imported by the *MobileAmb* ambient has to specify its mobility strategy.

6.1.2. Weavings

Aspects of an ambient can be weaved together. The services of two aspects can be synchronized by the weavings in order to define the overall behaviour of an ambient.

All ambients have a predefined weaving between the *mobility* and the distribution aspect of an ambient. This weaving is needed

```
Interface IGetLocation
 getLocation(output Location: Ambient);
End Interface IGetLocation
```

Fig. 14. Specification of the IGetLocation interface.

```
01 Mobility Aspect MobilityAspect using ICapability
   …
02 TRANSACTIONS in EXIT(input Requested: Ambient):
03 exit  = isChild! (input Requested, output isChildOK)→
          EXIT1;
04 EXIT1 = {isChildOK==true}
           getParent(output Parent)→EXIT2;
05 EXIT2 = moving! (Requested,Parent);

End_Mobility Aspect MobilityAspect
```

Fig. 13. Partial specification of the *MobilityAspect* aspect.

```
Interface IRoute
 getRoute(input Name: ArchitecturalElement, output Route[]: Ambient);
End_Interface IRoute
```

Fig. 15. Specification of the IRoute interface.

```
01 Distribution Aspect ADist using IGetLocation, IGetRoute
02 Attributes
03    Variable
04     location : Ambient NOT NULL;
  …
05 Services
  …
06  in/out getLocation( output Location: Ambient)
     Valuations
         [in getLocation(output Location)] Location := location;
07  in/out getRoute(input Name: ArchitecturalElement, output Route[]: Ambient)    Valuations
         [in getRoute(input Name, output Route)] Route := deriveRoute(Name);
  ….
End_Distribution Aspect ADist
```

Fig. 16. Partial specification of an aspect of kind distribution.

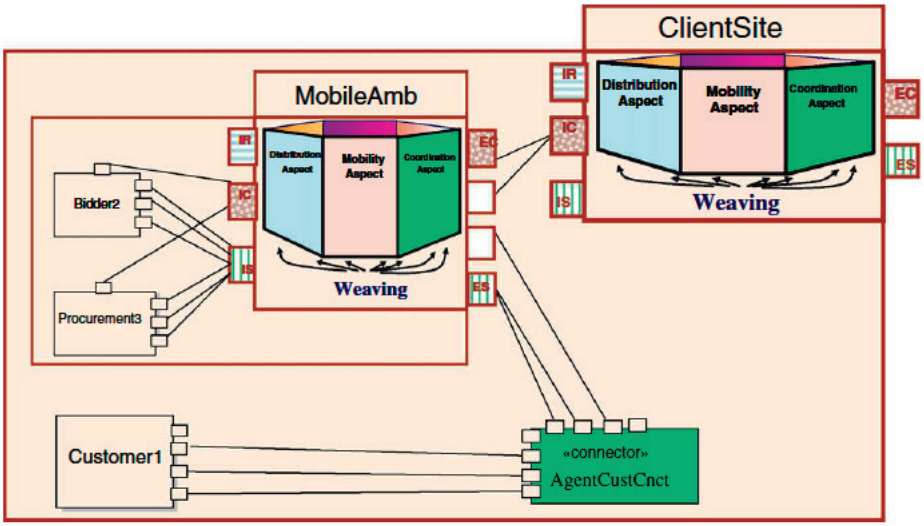


Fig. 17. A mobile ambient called MobileAmb.

because when an ambient serves the exit transaction (see Fig. 13), the ambient needs to know its parent ambient. This is due to the fact that when an architectural element invokes the exit service, the architectural element does not specify its destination ambient. Since an architectural element that exits an ambient always moves to the parent of its ambient, a weaving needs to trigger the *getLocation* service of the distribution aspect.

The predefined weaving of each ambient specifies that the *getParent* service of the *MobilityAspect* aspect triggers the *getLocation* service of a distribution aspect (see Fig. 18). The *getLocation* service (see line 06, Fig. 16) is executed instead of the *getParent* service (see Line 04, Fig. 13).

In addition, an ambient can have other weavings depending on the aspects that it imports. Although an ambient has three obligatory aspects, an ambient can also have more aspects. For example, an ambient can have a security aspect that constrains the messages that can be received and sent by the architectural elements inside its boundary. The security aspect can also define constraints on what type of architectural elements can be accepted in an ambient. In this way, a security aspect can be weaved to the *ACoordination* coordination aspect or to the *MobilityAspect* mobility aspect.

Weavings can allow ambients to import their generic aspects and adapting them to the software system needs. For example, a LAN ambient may need some security policies that are different from a PC ambient inside of the LAN. Therefore, both the LAN and the PC ambient import the same *ACoordination* aspect, but the *ACoordination* aspect is weaved with different security aspects.

6.1.3. Ports

Ports publish the services of interfaces. Since an ambient has predefined aspects that use interfaces, an ambient also has some predefined ports. Each ambient must have at least five ports: two

```
Weavings
     ADist.getLocation(Location) instead
                                    MobilityAspect.getParent(Parent);
```

Fig. 18. Specification of the predefined weaving between the *MobilityAspect* aspect and a distribution aspect called ADist of an ambient.

for the *MobilityAspect* aspect and two for the *ACoordination* aspect, and one for a distribution aspect. The *ICapabilities* interface services used by the *MobilityAspect* are requested/provided through the *InCapabilitiesPort* port and *ECapabilitiesPort* port (Marked with IC and EC, respectively in Figs. 4 and 17).

The *ICall* interface service which is used by the *ACoordination* aspect is requested/provided through the *InServicesPort* port and *EServicesPort* port. These ports allow architectural elements of an ambient to be in contact with the exterior boundary of an ambient. The port of an ambient that is marked with IS in Figs. 4 and 17, is the *InServicesPort* port and it is connected to internal architectural elements. The port of an ambient that is marked with ES in Figs. 4 and 17, is the EServicesPort port and it is connected to exterior architectural elements.

The *IRoute* interface service which is used by distribution aspects of an ambient is offered through InRoutePort port, marked with *IR* in Figs. 4 and 17. The InRoutePort port offers the *getRoute* service to internal architectural elements of an ambient.

Fig. 19 shows the specification of the ports section in an ambient specification. It can be observed that the *InCapabilitiesPort* port and the *ECapabilitiesPort* are of type *ICapability* interface i.e., they publish the services of the *ICapability* interface. The *EServicesPort* port and the *InServicesPort* port are of type *ICall* interface, and the *InRoutePort* port is of type *IRoute* interface.

```
Ports
     InCapabilitiesPort: ICapability …
     ECapabilitiesPort: ICapability …
     EServicesPort: ICall …
     InServicesPort: ICall …
     InRoutePort: IRoute …
….
 End_Ports
```

Fig. 19. Partial specification of ambients ports.

6.1.4. Example of an ambient: HostSite

Fig. 20 shows the specification of a site ambient called *HostSite*. The ambient imports the *MobilityAspect* aspect, the *ACoordination* aspect, and the *ADist* aspect. The predefined weaving and the five predefined ports of an ambient are specified. All the kinds of ambients are specified in the same way. The only difference is the reserved words that precede and end their specifications. These reserved words are *Virtual…End_Virtual*, *Site…End_Site*, and *Group…End_Group* for the specification of a virtual ambient, a site ambient and a group ambient, respectively.

It can be noticed, that the template of the ambient does not include the architectural elements that it locates or their attachments. It is assumed that ambients of an architectural model can locate all architectural elements and that all the possible attachments that connect architectural elements with ambients are created automatically. In this way, an ambient can be reused for different architectural models with different architectural elements.

6.2. An ambient-PRISMA configuration

In Ambient-PRISMA, the configuration of an architectural model is defined by instantiating ambients and assigning each architectural element instance in an ambient. In this way, the hierarchy of a distributed system can be modelled.

Fig. 21 shows the specification of the configuration called *AuctionSysConf* of the *AuctionSys* architectural model. The configuration consists of instantiating the loc data type, ambients, components, connectors and attachments. The loc data type is instantiated by giving the possible physical locations of the archi-

```
Ambient_Site type HostSite
 Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;
 Weavings
     ADist.getLocation(Location)instead
                                 MobilityAspect.getParent(Parent);
 End_Weavings

 Ports
     InCapabilitiesPort: ICapability …. ;
     ECapabilitiesPort: ICapability …;
     EServicesPort: ICall…;
     InServicesPort: ICall…;
     InRoutePort: IGetRoute …;
 End_Ports
End Ambient_Site type HostSite;
```

Fig. 20. HostSite ambient.

```
Architectural_Model_Configuration AuctionSysConf =

New AuctionSys
{
  01   IP1 = new loc(ip.of.host.1);
  02   IP2 = new loc(ip.of.host.2);
  03   IP3 = new loc(ip.of.host.3);

  04   ROOT = new Root() ;
  05   ClientSite = new HostSite(ROOT, IP1);
  06   AuctionSite = new HostSite(ROOT, IP2);
  07   LondonSaleRoomSite = new HostSite(Root, IP3);

  08   AuctionHouse1 = new AuctionHouse ("AuctionSite");
  09   Customer1 = new Customer("ClientSite");
  10   Procurement1 = new Procurement ("ClientSite", …);
  11   AgentCustCnct1 = new AgentCustCnct("ClientSite");
  12   AuctionHouseCnct1 = new AuctionHouseCnct("AuctionSite");

  13   AttchProc1Auc1 = new AttchProcAuc (Procurement1, PROCURAUCTPort,
                                          AuctionHouseCnct1, ProcurPortAuct);

  .....
```

Fig. 21. Specification of the AuctionSysConf configuration.

tecture (see Lines 01–03). Ambients are then instantiated. First, the *Root* ambient is instantiated. It can be noticed that the constructor of the *Root* ambient does not have any parameters (see Line 04). This is due to the fact that the *Root* ambient is not located in any ambient. Site ambients are instantiated by indicating in their constructor the name of their parent ambient (their location) and the physical address they represent (see Lines 05–07). For example, an ambient instance called *ClientSite* is created from the *HostSite* type ambient (defined in Fig. 20). The constructor of the *ClientSite* instance indicates that it is located in the *Root* ambient and that its physical location is *IP1* (see Line 05). Components and connectors are instantiated by indicating their locations in their constructors (Lines 08–12). In addition, the attachments relationships are also defined by connecting the instances created from the components and connectors types (Line 13).

It is important to clarify that the user does not have to define the attachments that attach components and connectors ports to ambients ports. The runtime environment creates the needed attachments transparently depending on which ambients the components, connectors and ambients are deployed in, i.e., depending on the hierarchy configuration of the ambients. For example in Fig. 21, the user only defines the *AttchProc1Auc1* attachment which connects the *Procurement1* component to the *AuctionHouseCnct1* connector, and depending where the *Procurment1* component and the *AuctionHouseCnct1* connector are located the attachments taking into account ambients are created (see Section 7.2).

7. Ambient-PRISMANET middleware

The implementation of an Ambient-PRISMA aspect-oriented software architecture is supported by Ambient-PRISMANET middleware. Ambient-PRISMANET maps the primitives of Ambient-PRISMA into .NET constructs in order to allow the execution of Ambient-PRISMA distributed and mobile software systems. The implementation of the distributed characteristics

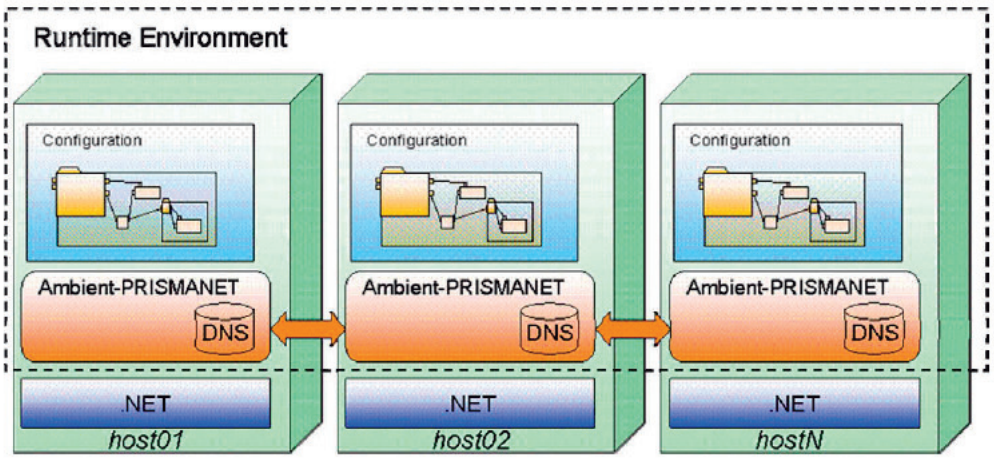


Fig. 22. Distributed runtime environment of Ambient-PRISMANET.

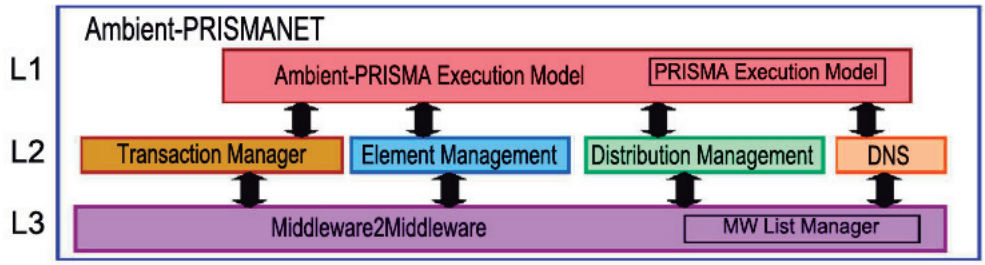


Fig. 23. Ambient-PRISMANET architecture.

have been performed using .NET Remoting technology (McLean et al., 2003) which provides mechanisms for accessing objects in a remote way as well as serializing them for providing mobility.

Ambient-PRISMANET provides a distributed runtime environment (see Fig. 22). An Ambient-PRISMANET instance, which forms part of the runtime environment, executes on each host. Each middleware instance manages architectural elements running on the host as well as collaborates with the rest of middlewares that form the distributed environment. The runtime environment has a distributed Domain Name Server (DNS) which all middlewares can access. In this way, a middleware can access any distributed architectural element.

Ambient-PRISMANET architecture consists of the following layers (see Fig. 23):

- *Ambient-PRISMA execution model*: This layer extends the PRISMA execution model (Pérez et al., 2005) in order to implement the primitives which are introduced by Ambient-PRISMA. Each Ambient-PRISMA metaclass and services (presented in Section 5) is implemented in a C# class. As a result, the implementation of a concrete Ambient-PRISMA application is achieved by extending these C# classes.
- *Transaction manager*: This layer is in charge of executing transactions in a correct way. It provides mechanisms that allow services of transactions to be offered by distributed architectural elements as well as mechanisms for recovering the state of the software architecture when a transaction fails.
- *Element management layer*: This layer is in charge of instantiating architectural elements (their aspects, ports, and weavings), and attachments, and storing each created instance in lists in order to be able to manage the instances.
- *Distribution management*: This layer is in charge of implementing the services needed for moving architectural elements from one middleware on a host to a new middleware on another host in a transparent way.
- *DNS layer*: This layer is in charge of resolving locations of architectural elements. It receives local requests in order to find the locations of distributed architectural elements and it receives distributed requests in order to consult whether an architectural element is executing on the host of the DNS layer or not.
- *Middleware2Middleware layer*: This layer is in charge of establishing communication among the different distributed middlewares that form part of a distributed environment. This layer acts as a proxy offering the services of the layers of a middleware to external middleware.
- *Middleware's list management layer*: This layer is in charge of maintaining a list of known middleware. This list is shared among the middleware which form part of a distributed environment.

```
[Serializable]
 public class Ambient: ConnectorBase
 {
 (...)
  public Ambient(string AmbientName): base(AmbientName)
{
(...)
 /**DEFINITION OF ASPECTS OF AN AMBIENT **/
 AddAspect(new ACoordinationAspect());
 AddAspect(new MobilityAspect());
(...)
 /** DEFINITION OF PORTS OF AN AMBIENT **/
 // Ports for generic communication
 InPorts.Add("ICallPort-InOut", "ICall", "INTERN");
 InPorts.Add("ICallPort-OutIn", "ICall", "EXTERN");
 // Ports for mobility
 InPorts.Add("ICapabilityPort-OutIn", "ICapability", "EXTERN");
 InPorts.Add("ICapabilityPort-InOut", "ICapability", "INTERN");
(...)
 }
```

Fig. 24. A segment of the `Ambient` class constructor.

Each of the layers presented in Fig. 23 has been developed because the .NET platform does not offer them. For more details on the implementation of each layer refer to Ali (2008). The Ambient-PRISMA Execution Model (Level L1) uses the layers of L2. The different layers at Level L2 also collaborate with each other. Layers in Level L2 use the Middleware2Middleware layer (L3) to communicate with other layers on other hosts.

In the following, how Ambient-PRISMANET implements ambients, distributed attachments, and mobility is explained. Refer to Ali (2008) for more details about Ambient-PRISMANET middleware.

7.1. Ambients

The Ambient-PRISMANET Execution Model layer contains the classes that are needed to map the Ambient-PRISMA concepts into .NET. The `Ambient` class has been implemented by extending the `ConnectorBase` class (see Fig. 24). The `Ambient` class inherits `ConectorBase` properties and methods and also defines its own. A characteristic of all ambients is that they have the predefined Mobility and Coordination aspects as well as their five predefined ports. Fig. 24 shows that the predefined aspects have been included by using the `AddAspect()` method in the `Ambient` constructor as well as the predefined ports by the `Add()` method.

Fig. 25 shows how the three ambients extend the Ambient class. Group ambients are implemented in the AmbientGroup class. The AmbientGroup class has a list for each kind of architectural element that it can contain: components (*componentsList*), connectors (*connectorsList*), and group ambients (*groupAmbientsList*). The methods of this class, mainly override the ones of the Ambient class. Elements that move in a Group ambient originate from either a Site ambient or another Group ambient. Also, the destination of elements that exit a Group ambient is either a Site ambient or a Group ambient. In this way, ambients of the Group ambient kind do not deal with serialization or deserialization since the mobility that is performed is always local to the current host.

Site Ambients are implemented in the `AmbientSite` class. For example, the *HostSite* ambient that is specified in Fig. 20 must inherit from the `AmbientSite` class in order to be executed. This kind of ambient can contain the same architectural elements as a Group ambient. As a result, the `AmbientSite` class extends the `AmbientGroup` class in order to inherit its lists and the methods that manage them. An important fact that distinguishes a Site ambient from other ambients is that it represents a device, and its physical location is passed to the Site ambient constructor. In addition, a Site ambient cannot change its location, and there is only one Site ambient per device.

In other words, a Site ambient can be considered to be a bridge between an ambient hierarchy and the middleware where other architectural elements execute. Each time an element exits a Site, the element moves into a new host and starts executing on another Ambient-PRISMANET middleware. However, this is transparent to the element that is moving, since the Site ambient is in charge of performing these changes.

Virtual Ambients are implemented in the `AmbientVirtual` class. For example, the Root ambient is instantiated from the `AmbientVirtual` class in order to be executed. These kinds of ambients are the parents of Site ambients, and can represent the Ambient-PRISMANET execution environment that is made up of the different middlewares executing on the distributed hosts.

A Virtual Ambient is the representative of a specific execution environment which can be connected to other execution environments. An instance of an *AmbientVirtual* class executing on a host represents a gateway for other *AmbientVirtual* instances that contain other Sites. In this way, an execution environment needs a Root that is represented in the host where an *AmbientVirtual* instance executes.

When architectural elements move from Site to Site, architectural elements need to exit a Site ambient and be accepted in an *AmbientVirtual* instance of their execution environment. The *AmbientVirtual* serializes architectural elements to Site ambients. However, elements that move among Sites cannot stay in an *AmbientVirtual* instance. If an architectural element enters a Virtual ambient and does not exit it, the mobility transaction fails and the element is sent back to its original ambient.

The `AmbientVirtual` class stores the name of Site ambients that it locates in a list called *siteAmbientsList* (see Fig. 25). Also, an element that is temporally transiting through a Virtual ambient is stored in the *tempFolder* variable. The methods of `AmbientVirtual` also override the methods of *Ambient* in a similar way to *AmbientGroup*. Most of the methods manage the *siteAmbientsList*

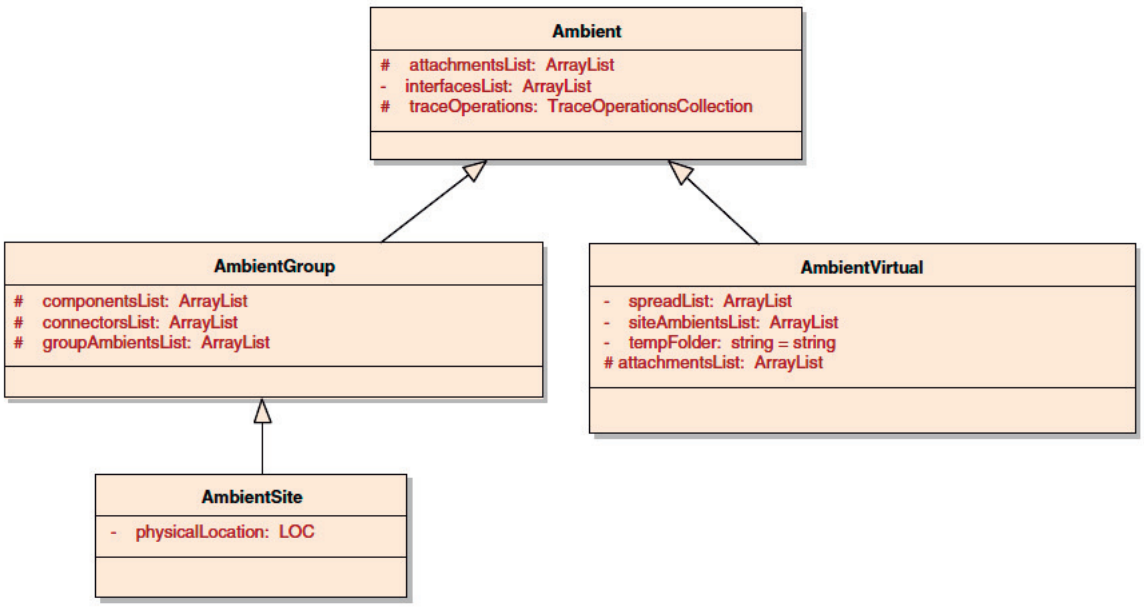


Fig. 25. The three kinds of ambients in Ambient-PRISMANET.

list. The *AddLocalAttachment* method of the `AmbientVirtual` class creates: attachments between the ports of architectural elements (that are temporally located in the Virtual ambient) and the Virtual ambient ports or attachments between the ports of Site ambients and the ones of the Virtual ambient. The attachments that connect two Site ambients are distributed. In this case, the location of the distributed ambients must be identified using the Ambient-PRISMA DNS and be connected by .NET Remoting.

7.2. Attachments

In Ambient-PRISMA, communication among architectural elements is performed by the attachments. As explained in Section 6.2, the user is not aware of the creation of attachments between architectural elements in different ambients. The ElementManagement layer is in charge of creating the attachments that communicate two distributed architectural element instances.

In Ambient-PRISMA, a distributed communication channel is not only formed by an attachment but by many attachments. An attachment can also be used by many communication channels. From now on, the term communication channel is going to be used to refer to a connection between two architectural elements that can be located in different ambients, and can be formed by many attachments. For example, the attachments shown in Fig. 4 are referred to as attachments and the *attachments* specified in Fig. 2 are *communication channels*. Thus, *attachments* can be added or removed when one of the instances they connect moves.

To implement an attachment in Ambient-PRISMANET, the Attachment class (see Fig. 26) has a collection that stores the names of the communication channels that use an attachment instance (communicationChannelsList). For example, in Fig. 9 the attachment between the *AuctionSite* and the *ClientSite* can be used by any communication channel that connects an instance located in *AuctionSite* and an instance located in *ClientSite*. As a result, the Attachment class has methods for consulting and modifying the communication channel collection. Also, the Attachment has a property called role to indicate whether an attachment instance connects two siblings or it connects a parent ambient and an instance located in it. This information is needed to configure attachments.

The initial configuration of the attachments is obtained as follows: Once the developer specifies the communication channels in the configuration as shown in Fig. 21, the ElementManagement layer of the Ambient-PRISMANET middleware uses the DNS to know on which host the instances that need to be connected are located. Then, a bottom-up search algorithm (Ali, 2008) is applied to obtain the route of each instance in a tree hierarchy of ambients (i.e., from the instance passing through the intermediate ambients until the Site ambient is reached). A bottom-up search is more efficient because an instance knows its direct parent ambient, whereas an ambient has many children. Once the route of both instances is obtained in an ordered list, the ambients common to both routes are identified. An attachment with sibling role is created between the next elements on both lists, and the remaining attachments are created with parent role. For example, to create the communication channel between the *Procurement1* component and the *AuctionHouseCnct1* connector (see Fig. 27): the route obtained for the *Procurement1* is Root\ClientSite\Procurement1, and the route obtained for the *AuctionHouseCnct1* is Root\AuctionSite\AuctionHouseCnct1. Ambient-PRISMANET creates an attachment between *AuctionSite* and *ClientSite* with a sibling role, an attachment between *Procurement1* and *ClientSite* with a parent role, and an attachment between *AuctionHouseCnct1* and *AuctionSite* with a parent role.

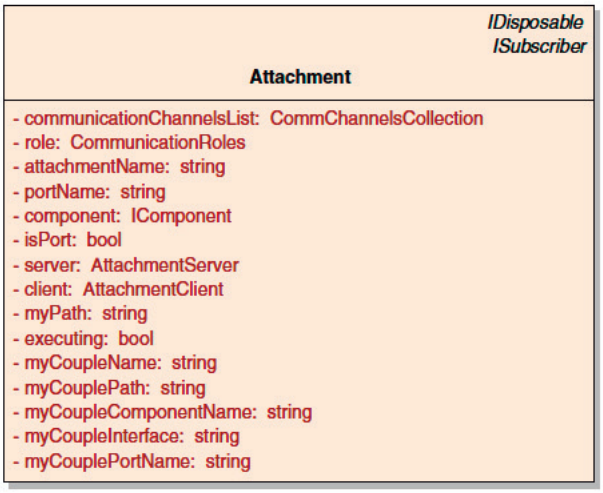


Fig. 26. The attachment class in Ambient-PRISMANET.

7.3. Mobility

Mobility in Ambient-PRISMA causes the reconfiguration of the software architecture. A mobile element is removed and added from one ambient to another preserving its state before moving in the new location (Pérez et al., 2005). In addition, the attachments that form a communication channel can be modified at execution time due to the fact that an architectural element they connect can move from one ambient to another. As a result, each time an architectural element moves the set of attachments that connect it to the rest of architectural elements change. In the following, we show the steps performed by the Ambient-PRISMANET middleware when *Procurement1* component of Fig. 5 moves from the *ClientSite* ambient to the *AuctionSite* ambient.

The *Procurement1* component invokes the *startMovement* service of the *ClientSite* ambient. The *startMovement* service permits the *ClientSite* ambient to create a list of structures that store the information of the current attachments of the *Procurement1* component. This structure is called *TraceOperation* (see Fig. 28). Let's call T01 the traceOperation list created in the *ClientSite*. Then, the *ClientSite* ambient removes all the attachments of the *Procurement1* component except for the attachment that connects the *Procurement1* component to its *InCapabilities* port.

Then, the *Procurement1* component requests the *exit* service. As a result, the *ClientSite* ambient sends to the *Root* ambient the T01 list through the accept service. The Root ambient receives the list and generates another list of *TraceOperations* structures called TO2, which stores the attachments that the Root has to create. The *Procurement1* component is removed from the *ClientSite* and added to the Root ambient.

Once the *Procurement1* component is in the *Root* ambient, the *ClientSite* modifies the attachments in its boundary that connected the *Procurement1* using TO1, and deletes the attachment that connected the *Procurement1* to the InCapabilitiesPort port of the *ClientSite* ambient. Finally, the *ClientSite* deletes the TO1. Fig. 29 shows the state of the configuration after the *Procurement1* component has exited from the *ClientSite* ambient. As it can be noticed, new attachments have been created in *ClientSite* that provide architectural elements connected to *Procurement1* to remain connected to it although it has moved.

Then, the *Procurement1* component requests the *enter(Procurement1, AuctionSite)* service to the *Root* ambient. In a similar way to the exit, the *Root* sends the TO2 list to the *AuctionSite* ambient, the *AuctionSite* ambient creates the TO3 list, and the *AuctionSite* creates an attachment between a port of *Procurement1* and its InCapabilitiesPort port. Then, the Root uses a service offered by

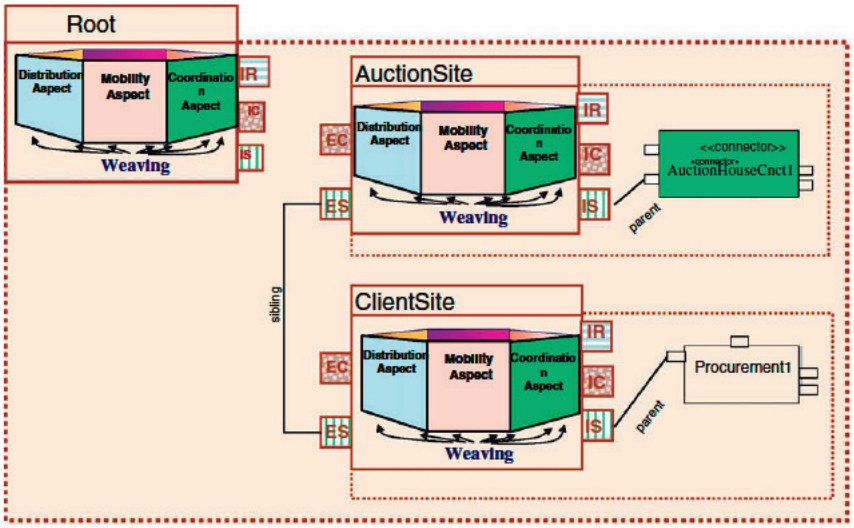


Fig. 27. Creation of attachments.

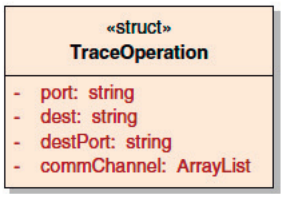


Fig. 28. TraceOperation structure.

the Distribution Management layer, which basically ensures that the *Procurement1* is moved to a consistent state i.e., when it finishes executing the services of its aspects, and serializes its state. The *Procurement1* enters the *AuctionSite* ambient. Afterwards, the *Root* checks whether it needs to make any attachment modifications in its boundary. In this case, the *Root* only deletes the attachment between the *Procurement1* and its InCapabilitiesPort port. Thus, the attachment between the *AuctionSite* ambient and the *ClientSite* ambient is used as part of the channel to connect the *Procurement1* component to the *AgentCustCnct* connector.

Finally, the *Procurement1* component requests the *finishMovement(Procurement1)* service from the *AuctionSite* ambient. As a result, the *AuctionSite* uses the TO3 to create the attachments needed for connecting Procurement1 (see Fig. 30).

8. Ambient-PRISMA case tool

The objective of the Case tool is to allow a user of the Ambient-PRISMA approach to model architectures using an intuitive and friendly graphical interface, verify that the models are correctly built, and automatically generate executable code.

Model Driven Engineering is a software development paradigm that is based on models that are transformed and used to generate software artefacts (Warmer and Kleppe, 2003). Ambient-PRISMA follows MDE in order to develop applications from its technology-independent model. Currently, there are two main approaches that apply MDE: the Model Driven Architecture (MDA) (Object Management Group, 2003), proposed by the Object Management Group (OMG), and the Software Factories approach, proposed by

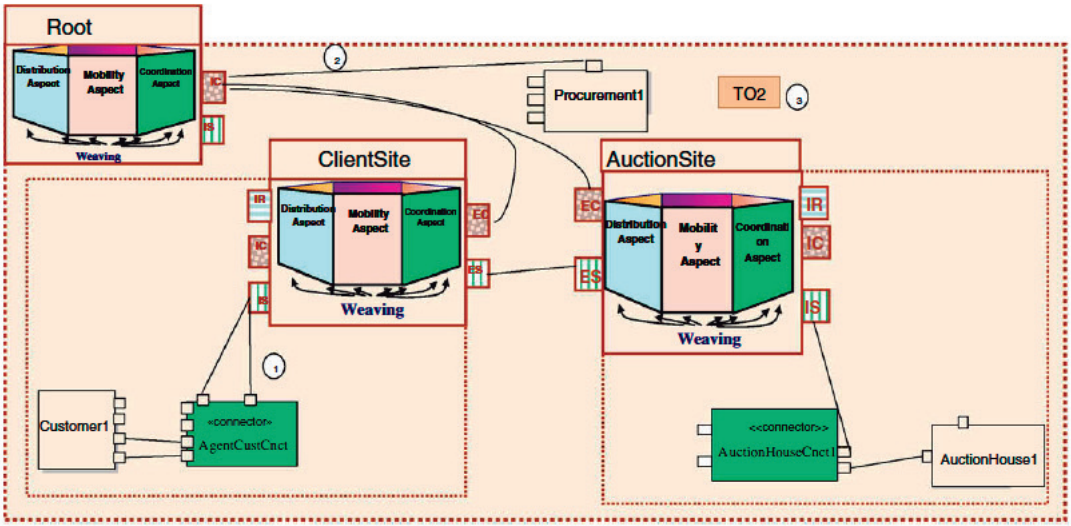


Fig. 29. Configuration after that Procurement1 has exited ClientSite.

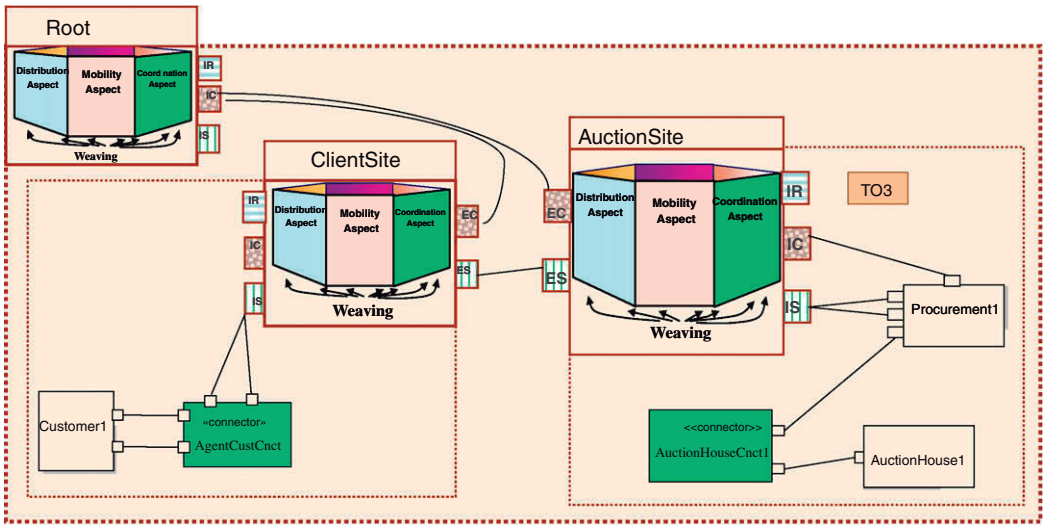


Fig. 30. Configuration obtained after Procurement1 requests the finishMovement service from AuctionSite.

Microsoft (Greenfield et al., 2004). In this work, we have used Microsoft's Domain Specific Language Tools (DSL Tools) (DSL, 2009) in order to implement the modelling tool of Ambient PRIS MA and the code generation templates.

8.1. Modelling

DSL Tools consists of the Domain Model Tool, and the Model Designer Tool, and code generation templates (DSL, 2009). The Domain Model tool provides a class diagram toolbox for allowing the user to define a domain model. The user represents concepts of its metamodel by using classes and relationships. The AMBI ENT PRISMA metamodel has been introduced using the Domain Model Tool (see Fig. 31).

The Model Designer Tool permits associating a graphical meta phor to each concept defined using the Domain Model Tool. DSL tools generates a specialized Modelling tool from the graphics de fined through the Model Designer Tool and the metamodel intro

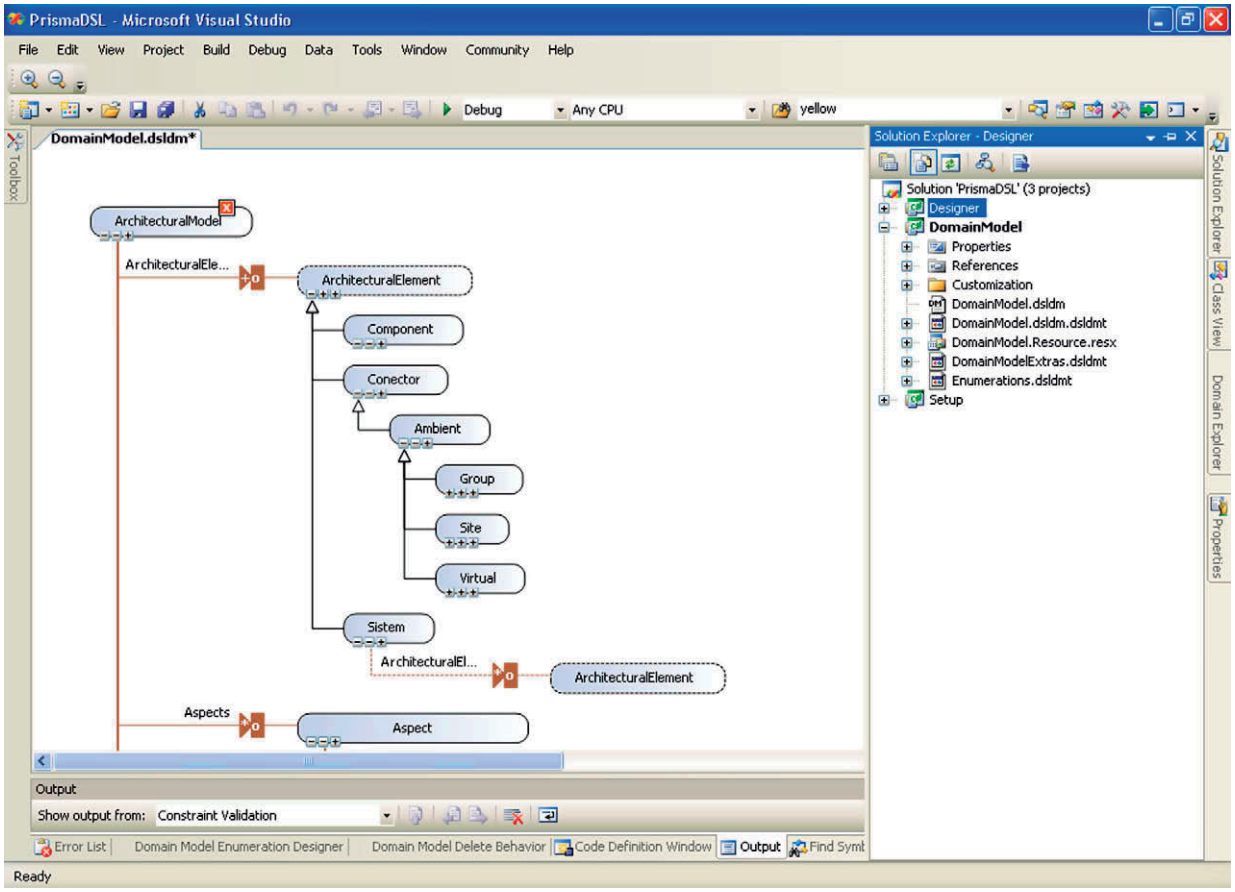


Fig. 31. Ambient-PRISMA metamodel in the DomainModel tool.

duced in the Domain Model. We have associated a graphical sym bol for each AMBIENT PRISMA metaclass and used DSL tools to cre ate our modelling tool (see Fig. 32). The user of the Ambient PRISMA Modelling tool drags and drops icons from the toolbox to the drawing sheet. While the user is building its model, constraints are checked.

The Modelling Tool allows the user to define ambients using interfaces, aspects, ports, and weavings. In addition, ambients can be related to other architectural elements of the software architecture. For example, Fig. 32 shows how a Group ambient called *MobileAmb* is defined by importing a distribution aspect which allows it to be mobile. The distribution aspect is called *Mobi leAmbDist* aspect and it uses an interface called *IMobility*. The Mobi leAmb ambient has a port which is connected through an attachment to a port of a connector called *AgentCustCnct*. It can be noticed that the predefined aspects of the MobileAmb ambient i.e., the *MobilityAspect* aspect and the *ACoordination* aspect do not have to be defined by the user.

The DSL tool includes templates which implement code gener ation patterns. These templates use the Ambient PRISMANET Exe cution Model to generate the applications implemented in C#.

8.2. Ambient PRISMA execution environment

Once the code is generated in C#, the middleware is launched and the user can define which hosts constitute the distributed environment. A panel where the user can add and delete hosts is provided (see Fig. 33a). In addition, it shows the hosts where other middlewares are executing. The host of the panel is the one that appears in bold.

Then, the configuration is uploaded (for example the configura tion specified in Section 6.2), and the middleware automatically creates the instances of the components, connectors, and ambients (and their corresponding aspects, etc), deploys them on their cor responding hosts and creates the needed attachments (as ex plained in Section 7.2). A generic user interface created for each running application can be used in order to execute services, query the value of attributes and validate the correct behaviour of each of the architectural elements that compose the architecture. For example, in Fig. 33b the user interface for the auction example is shown where the left hand contains the architectural elements. When a user clicks on the architectural element the panel shows its ports. When a port is chosen the services that this port pub lishes are shown on the bottom right hand. A user can select this service and invoke it to cause its behaviour to be executed.

9. Related work

Not many software architecture approaches have focused on developing distributed and mobile systems. One of the first ap proaches that have focused on the specification of software archi tectures for distributed systems is known as Darwin (Magee et al., 1995). Distributed Component based applications have been developed in the CORBA environment from Darwin specifications. However, Darwin does not support mobile systems development.

Some architecture approaches have modelled mobility in a technology independent way such as Community (Lopes et al., 2002) which is based on category theory, MobiS (Ciancarini et al., 1998) based on a multiple tuple space model, CHAM (Inverardi and Wolf, 1995) based on molecular solutions, Con Moto (Schafer, 2006) based on π calculus, and LAM model (Xu et al., 2003) based on Petri nets. Each of these approaches models locations and mobility. Community models locations by using a data type and a location is represented by a value of a variable. MobiS, and the LAM model use composite components for representing locations. Con Moto models locations by special kinds of components called physical components. Mobility of architectural elements is sup ported by reconfiguring the software architecture. On the other hand in Community, mobility is modeled as a change in a value. However, all of the mentioned approaches have not been used for developing and executing mobile applications.

Other software architecture approaches provide programming abstractions for supporting distributed applications. Java Aspect Components (JAC) (Pawlak et al., 2004) is a framework that pro vides an aspect oriented programming environment for distrib uted components. The JAC framework is based on the notion of containers that host software entities that can be both components and aspects. When an application is distributed, these containers become remotely accessible (either with RMI or CORBA). JAC also provides a library with some predefined aspects for distributed systems. However, it is dependent on Java and no work has been found on supporting mobility in JAC. FarGo (Holder et al., 1999) is a software architecture programming abstraction which allows components to move and their references to be automatically up dated. Ambient PRISMA follows MDE where it provides architec tural models to be specified independently of a specific

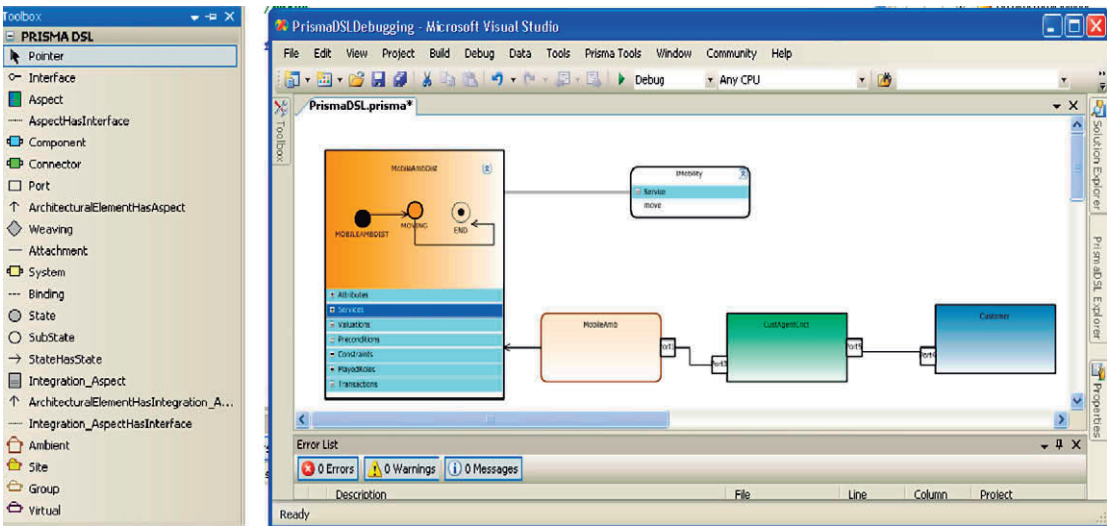


Fig. 32. Toolbox and the modelling tool.

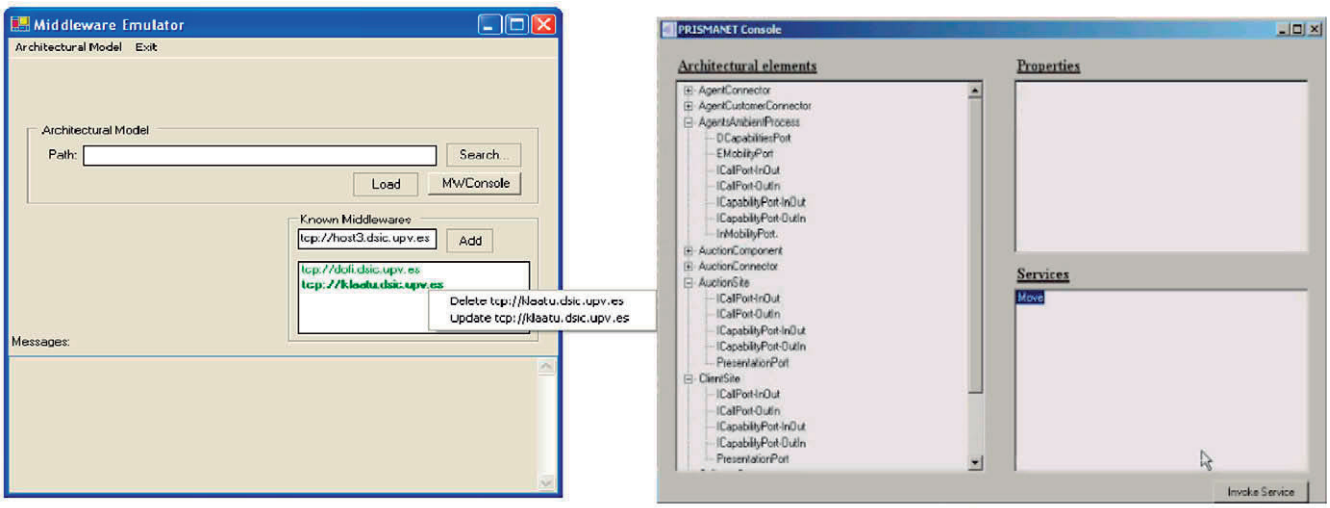


Fig. 33. Execution environment.

technology and provides a model compiler, which generates executable mobile applications.

C2 (Medvidovic et al., 2001) is an architectural style that supports both distribution and mobility. The style provides software connectors that are able to move components. It also has an implementation infrastructure called Prism MW (Mikic Rakic et al., 2008) to support this architectural style. However, there is no separation between coordination and distribution. Therefore, the components are the only architectural elements that are mobile, and the connectors are static. In our approach, all architectural elements can move. In addition, ambients can model hierarchal locations whereas in C2 locations are flat. Also, Ambient PRISMA combines CBSD and AOSD, achieving a better separation of concerns, such as separating the specification of the distribution concern from the mobility concern.

When we began to implement Ambient PRISMANET, we looked for implementations of AC. In (Cardelli, 1997), Cardelli proposes an algorithm for implementing AC in Java. The implementation describes a tree structure where each node of the tree is an ambient. The tree structure is modified through operations that implement enter and exit capabilities. However, the objective of this implementation is to show how the capabilities change the tree structure of ambients without considering distributed nodes.

To date, we have only found a single work that implements AC for supporting distributed applications. In Vasudeva and Joshi (2004), a scheme is proposed for supporting AC capabilities in distributed applications through an ambient runtime that sits above the .NET framework. This scheme focuses mainly on software mobility. However, Vasudeva and Joshi (2004) only proposes a scheme which is not implemented. In Ambient PRISMANET we have implemented AC primitives as well as software architecture concepts such as attachments.

10. Conclusions and future work

This paper has presented Ambient PRISMA, an approach which combines AOSD, software architecture, and MDE for developing mobile systems. Ambient PRISMA enriches the PRISMA approach with the concept of ambient inspired from AC. Ambient PRISMA extends the PRISMA metamodel, architectural model, middleware and case tool for supporting ambients.

Since Ambient PRISMA is inspired from AC it is characterized from other software architecture approaches in the following:

- Ambients can locate components, connectors, and other ambients.
- The hierarchy of distributed networks can be modelled in a realistic way.
- Mobility is modelled by navigating through the ambient hierarchy, i.e., by exiting and entering ambients. This is important because moving an element from an origin to a destination can be performed if only the mobile element exits and enters the intermediate ambients.

Ambient PRISMA is based on AOSD. The use of aspects provides Ambient PRISMA with a higher level of separation of concerns, which increases the reusability and adaptability of the mobile system. In addition, the use of aspects has allowed Ambient PRISMA to describe mobile behaviour which in most software architecture is lost. This has been important for allowing a more complete code generation.

Ambient PRISMA approach is characterized with the following:

- *Location:* In Ambient PRISMA, an explicit notion of location is provided by using the ambient concept. An ambient in Ambient PRISMA is a kind of connector that controls or coordinates a boundary where other architectural elements are located. In this way, an architectural element can only move to an ambient. As a result, space is represented as a tree hierarchy of ambients. In addition, ambients have been classified into three kinds: Virtual, Site and Group. This classification has been essential in order to represent network constrains and properties.
- *Mobility support*: In Ambient PRISMA, ambients provide mobility to other architectural elements. Each ambient provides the enter and the exit services to the architectural elements located in it. When one of these services is requested, a reconfiguration of the origin ambient and the destination ambient is provoked. Components, connectors and ambients can be mobile. The only architectural element which cannot be moved is the Root ambient.
- *Location awareness*: Architectural element instances use their distribution aspects in order to store the name of their parent ambient. As a result, architectural elements can take different decisions depending on their current locations.
- *Coordination*: In Ambient PRISMA, ambients have been incorporated as special kinds of connectors. This is due to the fact that an ambient needs coordination mechanisms for providing dis

tributed communication and mobility. In addition, simple con nectors can be used for other coordination purposes. Simple connectors can be connected to ambients and other kinds of connectors for forming more complex coordination behaviours.

- *Middleware support*: Ambient PRISMA primitives are currently supported on the .NET platform thanks to the Ambient PRISMA NET middleware. Ambient PRISMANET allows distributed and mobile applications to be executed.
- *Tool support*: The Ambient PRISMA Case Tool allows users to graphically model Ambient PRISMA software architectures, ver ify their correctness and execute a mobile software system.

Case studies such as the Auction System, and teleoperation sys tems (Ali et al., 2006; Ali et al., 2007) have been used in order to validate that the distribution and mobility primitives of Ambient PRISMA are expressive enough in order to create software architec tural models that can be compiled to obtain executable mobile applications. In these cases, we have noticed that although their business logic and architecture are different, their distribution behavior is similar. The distribution aspects could be reused not only in the same case study but also in both of them. In addition, ambients specification can be reused. One scenario presents diffi culties when tackled by our approach; this is when an architectural element needs to change distribution behaviors at runtime. For example, the number of enters and exits is determined at design time, as there is no way to specify them dynamically.

In the near future, Ambient PRISMA needs to be enriched with security primitives. Although ambients by definition are bounded places which control what can enter and exit them, our case stud ies have not considered modeling security scenarios. Security as pects can be specified by constraining what can enter and exit ambients.

The Case Tool also requires improvements. Better guidance for the user during the development process and extended functional ity should be provided such as taking into account resource con straints, latencies of networks, or guiding the user with mobility patterns which we have detected during the case studies develop ment. Currently, configurations in the Ambient PRISMA Case Tool have to be specified textually. This can be improved in order to per form them graphically.

Moreover, the Ambient PRISMANET middleware should be adapted for mobile computing where devices are mobile. In the current version of the Ambient PRISMA metamodel, there is no restriction for not allowing mobile sites. However, these devices will need a light weight version of Ambient PRISMANET which currently does not exist.

Acknowledgements

This work was supported by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, META project TIN2006 15175 C05 01. The authors acknowledge the undergraduate students who have implemented parts of the Ambient PRISMANET middleware, especially Carlos Millan Belda. The work performed in this article has been performed while Nour Ali and Carlos Solis were affiliated to the Polytechnic University of Valencia. Currently, they have re search fellows in Lero the Irish Software Engineering Research Centre, which is funded by the Science Foundation Ireland grant 03/CE2/I303_1. Finally and not least, the authors would like to acknowledge Kate Flynn for reviewing the language of this paper.

References

Ali, N., 2008. Ambient-PRISMA: ambients in aspect-oriented software architecture. Ph.D. Thesis, Polytechnic University of Valencia, February 2008.

Ali, N., Perez, J., Costa, C., Ramos, I., Carsi, J.A., 2006. Mobile ambients in aspect-oriented software architectures. In: IFIP Working Conference on Software Engineering Techniques – SET 2006, LNCS, Warsaw, Poland, October 17–20, 2006. ISSN:1571-5736.

Ali, N., Perez, J., Costa, C., Ramos, I., Carsi, J.A., 2007. Distributed replication in aspect-oriented software architectures using ambients. IEEE Latin America Transactions 5 (4). July (in Spanish).

Baniassad, E., Clements, P.C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B., 2006. Discovering early aspects. IEEE Software 23 (1), 61–70.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. Addison Wesley.

Batista, T., Chavez, C., Garcia, A., Rashid, A., Sant'Anna, C., Kulesza, U., Filho, F.C., 2006. Reflections on architectural connection: seven issues on aspects and ADLs. In: Proceedings of the 2006 International Workshop on Early Aspects At ICSE (Shanghai, China, May 21–21, 2006), EA '06. ACM, New York, NY, pp. 3–10.

Cardelli, L., Gordon, A.D., 1998. Mobile ambients, foundations of software science and computational structures. In: First International Conference, FOSSACS '98, LNCS 1378. Springer, pp. 140–155.

Cardelli, L., 1998. Abstractions for mobile computation. In: Vitek, J.C.J. (Ed.), Secure Internet Programming: Security Issues for Distributed and Mobile Objects, LNCS, vol. 1603. Springer-Verlag, pp. 51–94.

Cardelli, L., 1997. Mobile ambient synchronization. SRC Technical Note Report 1997-013, Digital SRC, 1997.

Carzaniga, A., Picco, G.P., Vigna, G., 1997. Designing distributed applications with mobile code paradigms. In: Proceedings of the 19th Conference on Software Engineering (ICSE'97). ACM Press, pp. 22–32.

Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A., 2005. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. Lancaster University, Lancaster, AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, 18 May 2005, pp 1–259. <http://www.comp.lancs.ac.uk:8080/c/portal/layout?p_l_id=1.94>.

Ciancarini, P., Mascolo, C., 1998. Software architecture and mobility. In: Perry, D., Magee, J. (Eds.), Proceedings of the 3rd International Software Architecture Workshop (ISAW-3), ACM SIGSOFT Software Engineering Notes, Orlando, FL, November 1998, pp. 21–24.

Cuesta, C., Romay, M.P., De La Fuente, P., Barrio-Solórzano, M., 2005. Architectural aspects of architectural aspects. In: Proceedings of the Second European Workshop on Software Architecture, LNCS 3527. Springer-Verlag, Pisa, June 2005.

Domain-Specific Language (DSL) Tools, 2009. <http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx>.

Filman, R.E., Elrad, T., Clarke, S., Aksit, M., 2004. Aspect-Oriented Software Development. Addison Wesley Professional.

Greenfield, J., Short, K., Cook, S., Kent, S., 2004. Software Factories. Wiley Publishing Inc..

Holder, O., Ben-Shaul, I., Gazit, H., 1999. Dynamic layout of distributed applications in FarGo. In: 21st International Conference on Software Engineering (ICSE 99), Los Angeles, California, United States, 1999.

Inverardi, P., Wolf, A.L., 1995. Formal specifications and analysis of software architectures using the chemical abstract machine model. IEEE Transactions on Software Engineering 21 (4), 100–114.

Kiczales, G., Hilsdale, E., Huguin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. An overview of aspect. In: Proceedings of the 15th European Conference on Object-Oriented Programming, LNCS 2072. Springer, Budapest, Hungary, June 18–22, 2001.

Lobato, C., Garcia, A., Romanovksy, A., Sant'Anna, C., Kulesza, U., Lucena, C., 2004. Mobility as an aspect: the AspectM framework. In: The 1st Brazilian Workshop on Aspect-Oriented Software Development – WASP'04, SBES'04, Brasília, Brazil, October 2004.

Lopes, C.V., 1997. D: a language framework for distributed programming. Ph.D. Thesis, College of Computer Science, Northeastern University, Boston, USA, 1997.

Lopes, A. Fiadeiro, J.L., Wermelinger, M., 2002. Architectural primitives for distribution and mobility. In: 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, pp. 41–50.

Magee, J., Dulay, N., Eisenbach, S., Krammer, J., 1995. Specifying distributed software architectures. In: 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, 1995, pp 137–153.

McLean, S., Naftel, J., Williams, K., 2003. Microsoft NET. Remoting. Springer.

Medvidovic, N., Rakic, M., 2001. Exploiting software architecture implementation infrastructure in facilitating component mobility. In: Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions of Software Engineering 1 (January), 26.

Mikic-Rakic, M., Malek, S., Medvidovic, N., 2008. Architecture-driven software mobility in support of QoS requirements. In: 1st Workshop of Software Architecture for Mobility (SAM 2008), Collocated with ICSE, 2008.

Object Management Group, 2003. Model driven architecture guide, 2003.

Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L., 2004. JAC: An Aspect-based Distributed Dynamic Framework Software: Practice and Experience, vol. 34. John Wiley & Sons (Issue 12, 15 Jun, pp. 1119–1148).

Pérez, J., Ali, N., Carsí, J.A., Ramos, I., 2005. Dynamic evolution in aspect-oriented architectural models. In: Second European Workshop on Software Architecture, LNCS 3527. Springer, Pisa, June 2005.

Pérez, J., Ali, N., Carsí, J.A., Ramos, I., 2006. Designing software architectures with an aspect-oriented architecture description language. In: 9th International Symposium on Component-Based Software Engineering (CBSE 2006), Mälardalen University, Västerås near Stockholm, Sweden, June 29th–1st July 2006.

Pérez, J., Ali, N., Carsı, J.A., Ramos, I., Álvarez, B., Sanchez, P., Pastor, J.A., 2008. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Information and Software Technology 50, 9–10 (August, 969–990).

Pérez, J., Ali, N., Costa, C., Carsí, J.A., Ramos, I., 2005. Executing aspect-oriented component-based software architectures on .NET technology. In: 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May–June 2005, pp. 97–108.

Russell Miles, Kim Hamilton, 2006. Learning UML 2.0, O'Reilly Media, Inc., April 25, 2006.

Sandholm, T., Huai, Q., 2000. Nomad: mobile agent system for an Internet-based auction house. Internet Computing, IEEE 4 (2), 80–86 (March–April).

Schafer, C., 2006. Modeling and analyzing mobile software architectures. In: Proceedings of the Third European Workshop on Software Architecture, (EWSA2006). Springer-Verlag.

Schmidt, D.C., 2006. Model-Driven Engineering, Computer, vol. 39(2). IEEE Computer Society, February 2006.

Shaw, M., Garlan, R., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall.

Soares, S., Borba, P., 2002. PaDA: a pattern for distribution aspects. In: Second Latin American Conference on Pattern Languages Programming – SugarLoafPLoP 2002, Itaipava, RJ, Brazil. ICMC, University of São Paulo Magazine, August 2002, pp. 87–99.

Szyperski, C., 2002. Component Software: Beyond Object Oriented Programming. ACM Press and Addison Wesley, New York, USA.

Tarr, P., Ossher, H., Harrison, W., Sutton, S.M., 1999. N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 21st international Conference on Software Engineering (Los Angeles, California, United States, May 16–22, 1999). ICSE '99. ACM, New York, NY, pp. 107–119.

Vasudeva, E., Joshi, Rushikesh K.A., 2004. In: Scheme for Implementing Ambient Calculus over an ARC Framework, Software Design and Architecture Workshop, IIIT Bangalore, December, 2004 (position paper).

Warmer, J., Kleppe, A., 2003. Object Constraint Language, The Getting Your Models Ready for MDA, second ed. Addison Wesley Professional Publishing.

Xu, D., Yin, J., Deng, Y., Ding, J., 2003. A formal architectural model for logical agent mobility. IEEE Transactions on Software Engineering 29 (1).

Nour Ali is a post doctoral researcher at Lero, Ireland. She has graduated in Computer Science from Bir-Zeit University-Palestine, and has obtained her Ph.D. at the Polytechnic University of Valencia, Spain. She has been a member of several research projects such as the Microsoft Research Cambridge Project: "PRISMA Model Compiler of aspect-oriented component-based software architectures" and others focusing on software architectures. Her research interests are software architectures, distributed and mobile systems, aspect-oriented software development and automatic code generation.

Isidro Ramos is a full professor since 1976. He has been teaching during 36 years in several universities. Currently, he is a lecturer at the Polytechnic University of Valencia and the leader of the Software Engineering and Information Systems research group of the same university. He has been a key researcher in 21 Research Projects, member and president of different Programme Committees of relevant conferences. He has also been the International Coordinator of the VII CYTED Subprogram on Applied Electronics and Computing, and the president of the University of Castilla la Mancha. He has received the Spanish Computer Science National Award 2006. He is currently working in MDE and in Software Architectures fields.

Carlos Solís is a Computer Engineer from the Instituto Tecnológico Autónomo de México. Currently, he is a postdoctoral researcher at Lero. He obtained his Ph.D. from the Polytechnic University of Valencia. He has worked as a Systems Analyst in Mexico's Central Bank, where he was specialized on financial systems. He has also been a consultant as an architect of J2EE systems for companies such as: Infonavit and GM. His research interests are Software Architectures, Hypermedia, and Business Processes.

Journal of Universal Computer Science, vol. 19, no. 10 (2013), 1433-1473
submitted 19/10/12, accepted 27/5/13, appeared 28/5/13 © J.UCS

Model-Driven Development of Aspect-Oriented Software Architectures

Jennifer Pérez
(Technical University of Madrid (UPM), Madrid, Spain
jenifer.perez@eui.upm.es)

Isidro Ramos, Jose A. Carsí
(Universidad Politécnica de Valencia, Valencia, Spain
iramos@dsic.upv.es, pcarsi@dsic.upv.es)

Cristóbal Costa-Soria
(Global Metanoia S.L., Paterna Technological Science Park (Valencia), Spain
ccosta@globalmetanoia.com)

Abstract: The Model-Driven Development (MDD) paradigm has become widely spread in the last few years due to being based on models instead of source code, and using automatic generation techniques to obtain the final software product. Until now, the most mature methodologies that have been proposed to develop software following MDD are mainly based on functional requirements by following the Object-Oriented Paradigm. Therefore, mature MDD methodologies are required for supporting the code generation from models that specify non-functional requirements. The Aspect-Oriented Software Development (AOSD) approach was created to provide explicit mechanisms for developing non-functional requirements through reusable elements called aspects. Aspect-Oriented Software Architectures (AOSA) emerged to deal with the design of both, functional requirements and non-functional requirements, which opened an important challenge in the software engineering field: the definition of a methodology for supporting the development of AOSAs following the MDD paradigm. This new methodology should allow the code generation from models which specify functional and non-functional requirements. This paper presents a mature approach, called PRISMA, which deals with this challenge. Therefore, this contribution takes a step forward in the area presenting in detail the PRISMA MDD process, which has been applied to generate the code of several real applications of the tele-operated robotics domain. PRISMA MDD approach provides complete support for the development of technology-independent AOSAs, which can be compiled from high-level, aspect-oriented architectural models into different technology platforms and languages following an MDD process. This contribution illustrates how to apply the PRISMA MDD approach through the modelling framework that has been developed to support it, and a case study of a tele-operated robot that has been completely developed using this approach. Finally, the results obtained from the application of PRISMA MDD process to develop applications of the tele-operation domain are analyzed in terms of code generation.

Keywords: Model-Driven Development MDD), Software Architecture, Aspect-Oriented Software Development (AOSD), Aspect-Oriented Software Architectures, Code generation
Categories: D.2.2, D.2.10, D.2.11, D.2.13

1 Introduction

The complexity of current software systems has increased the time and cost required in the development and maintenance processes of software. As a result, there is great interest in the software engineering area to fix this problem. Thus, to achieve the quality goals of software products and ensure market competitiveness, the Model-Driven Development (MDD) paradigm emerged.

MDD is a software development paradigm related to Model-Driven Engineering (MDE) [Schmidt, 06], which has become more and more widespread in the last few years [Beydeda, 05]. This is due to the fact that it improves the different stages of the software life cycle (requirements, analysis, software architectures, implementation, etc.) by automating their activities. It is mainly based on models for software development, techniques to improve reusability, and processes to support automation, traceability, and maintainability. It promotes a high level of abstraction of software artifacts (models instead of code) and the automation of most of programming tasks. The use of models to develop software provides solutions that are independent of technology, and whose source code can be obtained by means of automatic code generation techniques for different technologies and programming languages.

Until now, the most mature methodologies that have been proposed to develop software following MDD are based on the Object-Oriented Paradigm (OOP) [Meyer, 98]. So, only OO software systems can take advantage of the reduction of time to market and cost that MDD provides. OO modelling approaches are mainly focused on the specification of functional requirements. However, non-functional requirements are acquiring much relevance in the current software systems: such as safety, security or distribution in embedded systems, or persistence, graphical-user interface or privacy in information systems or social network services, etc. These non-functional requirements increment the complexity of software, and their implementation requires software modules being decoupled and modularized in order to be maintainable and reusable. Several software development approaches have been defined to deal with this need, such as Aspect-Oriented Software Development (AOSD) [Chitchyan, 05] and Software Architectures [Perry, 92]. Therefore, to provide the advantages of MDD to those software systems where non-functional requirements are critical, MDD needs to deal with the modelling and the code generation of these software development approaches, such as AOSD and Software Architectures. Aspect-Oriented Software Architectures (AOSAs) emerged to take advantage of both, the Software Architectures and AOSD approaches [Cuesta, 05], [Chitchyan, 05], etc. PRISMA is a model that follows the AOSA approach by integrating the Software Architectures and AOSD approaches [Pérez, 06a]. It is described by its meta-model, provides a formal Aspect-Oriented Architecture Description Language (AOADL) [Pérez, 06b] and defines a methodology for specifying its AOSAs [Pérez, 08a].

An important challenge in the software engineering field is the integration of the AOSA approach into the MDD paradigm to support the development and maintenance of complex software systems in an efficient way. In this paper, we deal with this challenge by the definition of the PRISMA MDD process for supporting the MDD of AOSAs.

From the PRISMA previous work, in this paper we take a step forward and we define a consolidated MDD process for AOSAs that wraps up all previous PRISMA

contributions. The construction of this AOSA MDD process is based on the results of the study that we have performed about "*The main set of desirable properties that any AOSA MDD approach should fulfil"*. These profitable results for the AOSA community are presented in this paper as the founding premises of the PRISMA MDD process.

This paper presents a novel contribution for PRISMA: its MDD process. The main goal of this PRISMA MDD process is to provide complete support for the development of technology-independent AOSAs, which could be compiled to different technology platforms and languages using automatic code generation techniques. To make feasible the PRISMA MDD process, the development of a tool for supporting the complete code generation for AOSAs was necessary. The PRISMA CASE framework has been developed to cope with this need. The paper illustrates how the PRISMA CASE framework supports each one of the steps of the PRISMA MDD process.

This PRISMA MDD process has been consolidated by means of its application to academic examples (such as: banking systems and auctions), and also by means of the generation of real applications from the robotic domain (such as: tele-operated robots and agriculture robots). From these applications of the PRISMA MDD process, we have obtained a set of code generation results, which analysis reveals that the complete automatic-code generation from AOSA models is feasible.

In summary, the novel contributions of this paper are: (i) the results of the study of the MDD support of AOSA approaches, (ii) the PRISMA MDD process, (iii) the complete PRISMA CASE framework, and (iv) the results of the application of the PRISMA MDD process. These contributions are structured in the paper as follows. The background about MDD, AOAS, and PRISMA, and the study about properties of AOSA MDD approaches are presented in Section 2. The robot *TeachMover* and the piece of its architecture that is used as an example throughout the paper are introduced in Section 3. The MDD process of PRISMA is explained in detail in Section 4. Section 5 discusses this contribution by analyzing related work. Section 6 describes and compares the experimental results obtained from the development of two applications using the PRISMA MDD process. Finally, conclusions and further work are presented in Section 6.

2 Background

2.1 Model-Driven Development (MDD)

Most mistakes that are made during software development come from the first stages of the software life cycle. Since these mistakes grow in an exponential way as the life-cycle progresses, it is necessary to focus on improving these first stages instead of postponing the solution for later stages. Most proposals that try to solve these problems improve software development by automating its first stages following MDD. Our contribution also fits these ideas and challenges focusing at the architectural level.

MDD is a software development paradigm of Model-Driven Engineering (MDE) [Beydeda, 05]. It is mainly based on models for software development, techniques to improve reusability, and processes to support automation, traceability, and maintainability. It promotes using models instead of code to automate software

development, guaranteeing solutions that are independent of the technology-platform, consistent and reusable. It is based on code generation techniques for generating the code of different technologies and languages from models (either models or metamodels (models of models)).

The OMG Meta-Object Facility (MOF) 2.0. [MOF, 12] specification defines an architecture to support meta-modeling, MDD and model management (definition of transformations, traceability links, mappings, etc., between models). It is a four-level architecture, whose main purpose is the management of model descriptions at different levels of abstraction. The upper layer (M3) defines the abstract language used to describe the entities of the M2 layer (metamodels) (see layer M3, Figure 1). The metamodel layer (M2) defines the structure and semantics of the models defined at the M1 layer (see layer M2, Figure 1). The M1 layer comprises the models that describe data, i.e. the application of the M0 layer (see layer M1, Figure 1). These models are described using the primitives and relationships described in the metamodel layer (M2). The lowest level is the information layer (M0), which comprises the final application to be described (see layer M0, Figure 1), i.e. the instances of the models that are defined at the M1 layer.

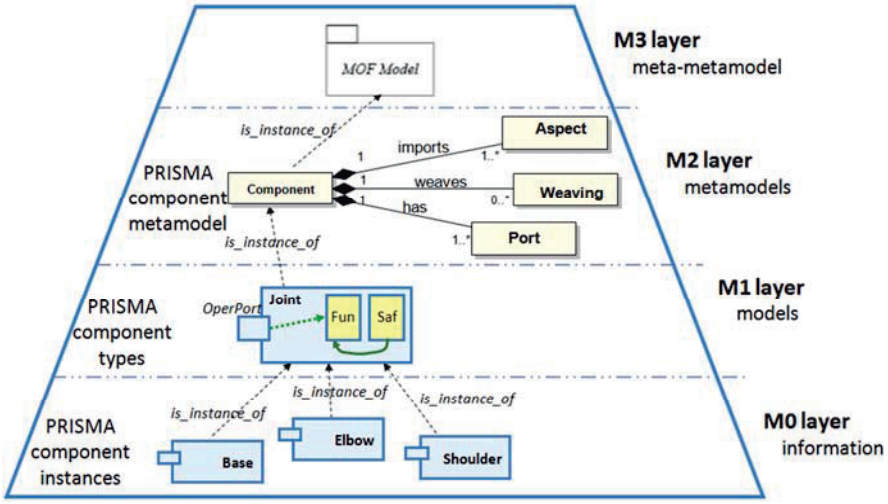


Figure 1: Meta-Object Facility (MOF) layers exemplified by PRISMA models

2.2 Aspect-Oriented Software Architectures in MDD

Software architectures and AOSD are the foundation of the AOSA approach. In this section, we provide an overview of AOSA by explaining the integration of Software Architectures and AOSD and their role in MDD, and the main properties that any AOSA MDD process should take into account for its construction.

2.2.1 Software Architectures

Software Architectures make software systems simpler and more understandable by hiding the low-level details. The works of Garlan [Garlan, 95] and Perry & Wolf

[Perry, 92] clearly define the role of software architectures in the software life cycle. The Software Architecture discipline bridges the gap between requirements and implementation stages of the software life cycle. The traceability between the resulting requirements of the requirements stage and developed code in the implementation stage is a key issue to prevent the failure of the software development process.

Software architectures comprise a wide-scope of tasks [Perry, 92]. This work is focused on the design of the structural viewpoint of Software Architectures, i.e. the description of Software Architectures. This structure is usually represented in terms of computational elements and the interactions among them using Architecture Description Languages (ADLs).

Software Architecture Descriptions are critical to trace the requirements to code in software development in general, and in MDD in particular. However, despite the fact that a wide variety of formal ADLs [Cuesta, 02], [Medvidovic, 00] have been proposed, there is a lot of work to be done to integrate formal ADLs into MDD.

2.2.2 Aspect-Oriented Software Development (AOSD)

Non-functional requirements have acquired an important relevance in current software systems. Non-functional requirements are usually common concerns of a domain system that are scattered in the software units that a system is composed of (classes/objects [Meyer, 98], components [Szyperski, 98], modules modules [Meyer, 03], etc.). These common concerns are crosscutting most software units of the system (*crosscutting concerns)*. These *crosscutting concerns* are tangled with the other concerns that also modify the same software unit affecting most functionalities of the system. Hence, the support of software modularity and *crosscutting-concerns* are essential challenges to be faced in software development. The *Separation of Concerns (SoC)* helps to address these challenges [Parnas, 72]. The *SoC* principle promotes dealing with the different concerns of a software system separately and context-unaware when possible. *Aspect-Oriented Programming (AOP)* [Kiczales, 97] has emerged as an innovative way of applying *SoC* in software development.

AOP proposes the separation of the *crosscutting-concerns* of software systems into separate entities, which are called *aspects.* This separation avoids having tangled concerns scattered among the software units and allows the reuse of the same aspect in different software units of the software system. AOSD has emerged not only to apply AO to the implementation stage, but also to apply it to every stage of the software life cycle. For this reason, the integration of AOSD development with the MDD process has to be taken into account.

There are some approaches that combine MDD and AOSD mechanisms such as [Aksit, 05], [Amaya, 05],[Kulkarni, 03], and [Simmonds, 05]. However, none of these AO approaches takes into account software architectures, and none of the original ADLs previously presented (see Section 2.2.1) explicitly distinguishes the conventional architectural elements from concerns, which crosscut multiple architectural elements of software architectures.

2.2.3 *The most relevant properties for MDD support of Aspect-Oriented Software Architectures*

The combination of AOSD and software architectures permits the separation of concerns of architectural components. So, software architectures can take advantage of AOSD benefits. This section analyzes the main set of desirable properties that any aspect-oriented architecture MDD approach should fulfil. These properties are mainly founded on the MDD principles [Schmidt, 06] and on the key guidelines about software development the literature sets out: high-level languages for programming [Parnas, 85], software partitioning by dividing into views [Perry, 92], [Kruchten, 95] or the size of code ("divide and conquer") [Dijkstra, 76] and software composition and decomposition [Parnas, 72] [Harrison, 02]. Hence, the premise, in which the study is based on, is the following: *"An aspect-oriented software architecture approach should completely support the development and maintenance processes of software following the MDD approach"*. And, the properties and the reasons for considering them as classification criteria of MDD aspect-oriented architectural approaches are described below:

- **Aspect-Oriented Model (AOM)**: This feature defines the kind of AOM that is integrated with the software architectural model, which may be mainly classified into asymmetric and symmetric [Harrison, 02]. This characteristic is important because the facilities of reuse and code generation can vary depending on which aspect-oriented model is used. An asymmetric model is based on a dominant decomposition, which is usually an OO-like functional decomposition. Models of this kind assume that aspects are non-functional concerns that crosscut the functional units of software (called *base code)*. However, in symmetric models, everything is considered as a concern, and there is no dominant decomposition. As a result, functionality is considered to be another concern, and concerns crosscut each other. Therefore, symmetric models are considered to be more flexible and abstract. However, since the most extended AOM is the asymmetric model of AspectJ [Kiczales, 01], asymmetric models are more widely used because they are easier to integrate in current software development approaches.
- **Architectural model**: This feature determines whether or not an architectural model provides connectors for modelling software architectures. The ones that have connectors provide features that improve the structure and maintenance of software architecture [Shaw, 94] . So, those architectural models that provide connectors make architectural elements more independent and facilitate their transformation in an MDD process, as well as their reusability and integration in other architectural models.
- **Formal ADL**: Another feature that is necessary to take into account when comparing architectural models is whether or not their corresponding ADLs are formal languages. The formal nature of an ADL is an indispensable property of architectural models if the purpose of the approach is to generate code without ambiguity, to verify properties, to validate behaviour, to trace the different levels of abstraction in a suitable way, and to evolve software architectures preserving the consistency of the system.
- **Graphical support**: The graphical specification of AOSAs greatly helps to avoid the complexity of using formal and technology independent ADLs. Graphical

support is achieved by defining the graphical metaphor for ADLs by means of a new language or by extending a well-known graphical language.

- **Definition of Aspects**: The most distinguishable feature of AO architectural models is how they integrate aspects and software architectures. There are two ways of doing this: (i) by simulating the notion of aspect by means of another architectural concept, or (ii) by defining a new concept for aspects in software architectures. The first way refines the architectural concepts by varying their original semantics; and the second one requires understanding a new concept to model software architectures. Therefore, those proposals that define aspects as a new concept with its proper semantics facilitate their transformation and unambiguous integration with any architectural model.
- **Definition of Weavings**: Weaving is a process that, in asymmetric AOM, consists of coordinating the *base code* and the *aspect code*. In symmetric AOM, weavings coordinate the different aspects. This is an important feature of AOMs and AO architectural models. The definition of weavings specifies where the weaving process between aspects and architectural elements is defined. If it is defined inside the aspect, the aspect is dependent on the context that the aspect is connected to. However, if it is defined outside the aspect, the behaviour of the aspect can be reused independently of where the aspects are connected to, and so, evolution and reuse are facilitated.
- **Aspect-Oriented Evolution**: The support of aspect evolution is an important feature that can improve the design-time and run-time evolution of software architectures. Thus, an approach that provides mechanisms for easily adding or removing aspects is a great advantage to perform transformations between models for their modification.
- **Purpose**: Since our premise of this analysis is the following: "*An aspect-oriented software architecture approach should completely support the development and maintenance processes of software following the MDD approach"*. The purpose of an approach is an essential feature to be able to compare the different AOSA approaches. There are AO architectural models that give complete support during the development process, others that analyze or evolve models, and still others that fulfil several purposes.
- **Technology**: This is an important feature that distinguishes the wide variety of aspect-oriented software architectural approaches that exist. If the purpose is to take advantage of MDD, AOSAs should be specified in an abstract way by means of formal and technology independent ADLs. As a result, a single specification can be transformed to different platforms and different programming languages. However, if the AOSA specification depends on a specific platform and/or programming language, its application and flexibility are considerably reduced.
- **Tool support**: A significant feature of a MDD AOSA approach is its support by means of a framework, which will guide the analyst during the development and maintenance processes. A framework can provide a wide variety of facilities such as modelling support, ADL generation, code generation, code execution, validation, verification, evolution, etc.
- **MDD support**: The previous criteria are essential to achieve the MDD support, which is essential for this contribution. The support that the MDD approach offers is significant in the development and maintenance processes. A complete MDD

support should consist in: (i) automatically generating the code from models, (ii) guiding the analyst through all the stages, and (iii) providing mechanisms to facilitate the tasks (some of these mechanisms are: verification techniques, reusability mechanisms, integration facilities, code generation mechanisms, etc.).

2.3 An Overview of PRISMA

2.3.1 The PRISMA Model

The PRISMA approach is based on the PRISMA model, its AOADL, its methodology, and its tool [Pérez, 06a], [Pérez, 06b], [Pérez, 08a]. PRISMA provides a model for the description of software architectures of complex and large systems. A PRISMA architectural element (components and connectors of an architecture design) can be seen from two different views: internal and external. In the external view (black box view), architectural elements encapsulate their functionality as black boxes and publish a set of services (*interface*) that they offer/ask to/from other architectural elements through their ports (see Figure 2.A). In the internal view (white box view), an architectural element is shown as a prism. Each side of the prism is an aspect that the architectural element imports (see Figure 2.B). In this way, architectural elements are represented as a set of aspects and the relationships among them (called *weavings*).

The notion of aspect arises to deal with crosscutting-concerns of software systems. This idea of crosscutting can vary depending on the nature of the model. The PRISMA model is a symmetrical AOM [Harrison, 02] because it does not consider functionality as a base code (i.e. different to aspects), and it does not constrain aspects to specify non-functional requirements (see property Aspect-Oriented Model (AOM), Section 2.2.3). In PRISMA, functionality is also specified as an aspect by providing a homogeneous treatment to functional and non-functional requirements. Aspects have been introduced in the PRISMA AOADL as a new concept rather than simulating the aspect using other architectural terms (components, connectors, views, etc.). This is due to the fact that a component can specify state and behaviour about different concerns, whereas an aspect is focused in a single concern. As a result, PRISMA preserves the meaning of the concepts of component and aspect, keeping them as first-order citizens.

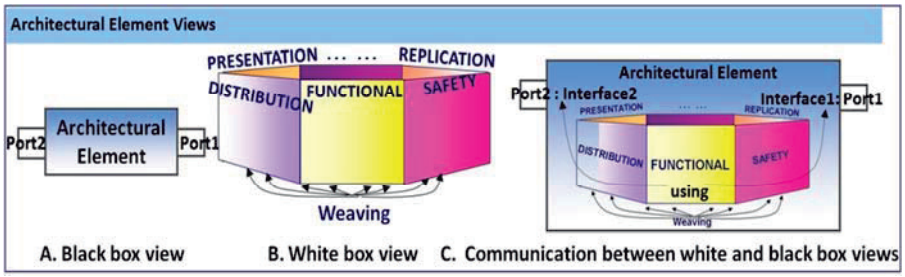


Figure 2: Views of an architectural element

With regard to AOSD, PRISMA aspects are artefacts of software architectures and represent a specific behaviour of a concern (safety, coordination, persistence, security, etc.) that crosscuts the software architecture. The same aspect can be imported by each one of the architectural elements (components and connectors) that need to take into account this behaviour. As a result, an architectural element is defined by a set of aspects that describe it from different concerns of the architecture. The communications between the white box and black box views is possible by means of interfaces, which are associated with ports and are used by aspects (see Figure 2.C). Consequently, a request for a service that arrives at a port of an architectural element is processed by an aspect that implements the same service that is provided by this port.

In PRISMA, a weaving indicates that the execution of an aspect service can trigger the execution of services in other aspects. In PRISMA, to preserve the independence of the aspect specification from other aspects and weavings, weavings are specified outside aspects and inside architectural elements (see Figure 2.C). They are specified using the weaving operators *after, before, instead, afterif(boolean_condition), beforeif(boolean_condition) and insteadif(boolean_condition),* and following the pattern: *aspect1.service1* ***weaving_operator*** *aspect2.service1*. These patterns means that the service2 of the aspect2 will be executed *after, before, instead* the *service1* of the *aspect1*, and in the case of the conditional weaving operators, the *service2* of the *aspect2* will be executed only *if the boolean_condition* is true. Aspects are reusable and independent of the context of application and weavings weave the different aspects that form an architectural element. This way of specifying weavings achieves not only the reusability of the aspects in different architectural elements, but also the flexibility of specifying different behaviours of an architectural element by importing the same aspects, and defining different weavings. Figure 3 illustrates an example of how PRISMA architectural elements import aspects, i.e. it shows the reusability facilities of PRISMA. This simple example shows how the same aspect, for example the aspect *FUNCTIONAL*, is imported by two different architectural elements, *Component1* and *Component3*, and how an architectural element, for example the connector *Connector1*, imports several aspects: *COORDINATION* and *SAFETY.*

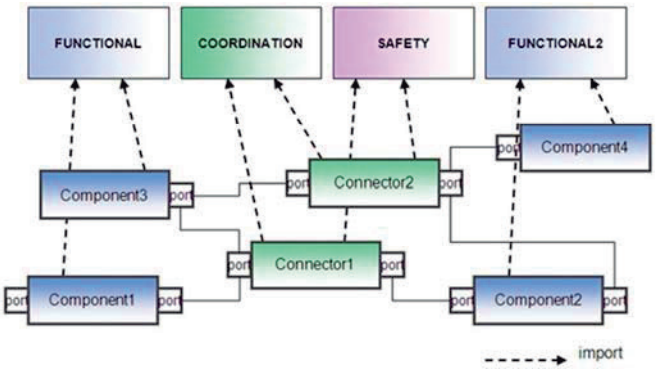


Figure 3: Crosscutting-concerns in PRISMA architectures

With regard to software architectures, PRISMA has three kinds of architectural elements: components, connectors, and Systems. A *component* is an architectural element that captures the functionality of software systems and does not act as a coordinator among other architectural elements, whereas a *connector* is an architectural element that coordinates the interactions among other architectural elements. Connectors provide the separation of component interactions. Connectors do not have the references of the components that they connect and vice versa. Thus, architectural elements are reusable and unaware of each other. *Attachments* are the channels that enable the communication between components and connectors through their ports. *Systems* are complex PRISMA components, which include a set of architectural elements (connectors, components and other Systems) that are correctly attached. *Bindings* establish the connection among the ports of a System (i.e. a complex component) and the ports of the architectural elements that this System contains.

The PRISMA model has an AOADL to support it. The PRISMA AODL is a formal and technology-independent language (see property Formal ADL Section 2.2.3). It is important to emphasize that most ADLs only permit the specification of the skeleton of architectures and the services that are sent/received among their different architectural elements. The PRISMA AOADL has greater expressive power and can specify more features and requirements using aspects [Pérez, 06b]: it can specify state and behaviour for each aspect.

2.3.2 PRISMA in MOF

The PRISMA MDD approach fits the four-level architecture of MOF. In this way, MOF allows us to clearly present the differences between PRISMA *types* and *instances* and their corresponding *models*. The PRISMA metamodel is defined at the layer M2 (see Section 2.1), and the **PRISMA type models** are specified at the layer M1 e.g. the model of a joint of the tele-operated robot software architecture (see layer M1Figure 1). So, the PRISMA model of the joint of the tele-operated robot is compliant with the PRISMA metamodel. Finally, the specific instances of an architecture configuration are placed in the layer M0. They are called in PRISMA, **PRISMA configuration models**. For example, Figure 1 illustrates how the *Elbow* is an instance of the component *Joint* of the layer M1, which means that the *Elbow* behaves as the *Joint* describes. PRISMA reusable types and patterns are specified at layer M1 and architectural configurations are defined at the layer M0 by instantiating the types and patterns of the layer M1.

In this work, we present an MDD approach by making this MOF architecture of PRISMA models feasible (see Figure 1). To implement the approach, it is important to keep in mind that MDD and automatic code generation avoids the mistakes of correspondence between the semantics of the model and the application code, but it does not prevent the modelling mistakes that the user makes. To prevent them, help must be provided to guide the user during the modelling process. This guidance mechanism is provided using verification techniques. Verification can be performed in the modelling stage to detect modelling mistakes. During the modelling stage, the verification techniques allows us to know whether or not a model satisfies the constraints that its metamodel defines (the model conforms to its metamodel). If a

model satisfies every constraint of its metamodel, then the model is conformed and verified.

3 The Tele-Operation Domain

The PRISMA MDD approach has been validated by designing academic examples such as, banking systems, auctions, etc., and also, the generation of real applications from the robotic domain such as: tele-operated robots, agriculture robots, etc. In this section, we describe one of these real applications, which will be used throughout the rest of the paper to illustrate our contribution. It is a robotic tele-operated system, which is a family of robots for hull maintenance operations called EFTCoR (Environmental Friendly and Cost-effective Technology for Coating Removal) [EFTCoR, 02], [Fernández, 05]. This work was feasible thanks to the close collaboration[1] with the DSIE (System Division and Electronic Engineering) Group of the Polytechnic University of Cartagena of Spain, who was involved in the design of this EFTCoR Robot to be run in the cleaning tasks of the shipyard of Cartagena City.



Figure 4: Tele-Operated Robots: A) EFTCoR Primary Positioning System (arm joint and joint of tracks), B) manual cleaning by an operator, C) The TeachMover Robot

Tele-operated robots are software intensive systems that are used to perform tasks that human operators cannot carry out due to the dangerous nature of the tasks or the hostile nature of the working environment (see Figure 4. B). The EFTCoR is a robotic platform with strong non-functional requirements that cleans the hulls of ships in a way that reduces the environmental pollution. Since the EFTCoR is a family of robots that are very large (big dimensions) and very heavy (high tonnage) (see Figure 4. A), a complete development of a small-scale robot was done before developing the software architecture of EFTCoR. Specifically, we validated our proposal by

[1] This close collaboration was through the national project DYNAMICA.

developing the *TeachMover* robot [TeachMover, 12] (see Figure 4. C). This robot was specially designed for the purpose of simulating the behaviour of large and heavy industrial robots that cannot be manipulated in laboratories. The *TeachMover* is simpler than EFTCoR, but it has the same architectural features at a small scale to test the software of industrial robots before its deployment. All these features allow the *TeachMover* robot to simulate the movements of most of the industrial tele-operated robots.

The *TeachMover* is formed by a set of joints that permit the movement of the robot, which are commanded by an operator from a computer. These joints are: *Base, Shoulder, Elbow, and Wrist*. In addition, it has a Tool to perform different tasks: catch, drop, push, pull, etc. (see Figure 4.C). In this case, the *Tool* is a gripper (in other cases it can be a brush to paint, a hose to clean, etc.), whose open and close actions allow the robot to pick up and deposit objects. Therefore, it allows the robot to move objects from an initial position to a final one. There are required safety constraints of the robot movements to be checked and to make sure that its movements are safe for itself and the environment that surrounds it.

From the different components that a tele-operated robot is composed of, and the *TeachMover* in particular, we are going to focus on those that implements the joints of the robot [Pérez, 08a]. The architecture of a tele-operated robot joint is defined by a System (complex component) called *Joint*, which is composed of architectural elements that interact with the hardware joints of the robot. Specifically, the *Joint* is a System composed of two components and a connector and their corresponding connections (see Figure 5). The component *Actuator* is in charge of (i) communicating with the hardware joint, when commands are sent to the hardware joint of the robot to be performed and (ii) notifying the joint System when the commands have been performed successfully. The component *WrapAspSys* encapsulates the behaviour and the state related to the software joint, such as the position of the joint and its movements. And the connector *CnctJoint* coordinates the interaction between the *Actuator* and the *WrapAspSys*.

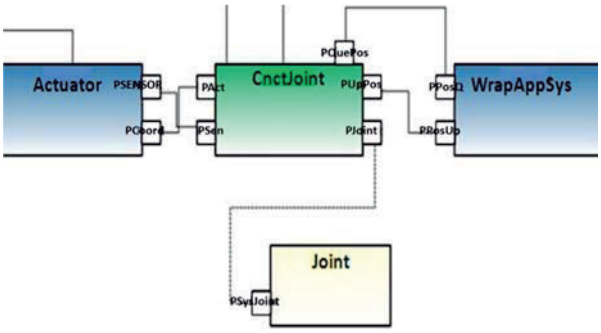


Figure 5: The joint architecture of a tele-operated robot

4 PRISMA: MDD Support for Developing Aspect-Oriented Software Architectures

This section describes how PRISMA provides complete support for the development of technology-independent, aspect-oriented software architectures following MDD.

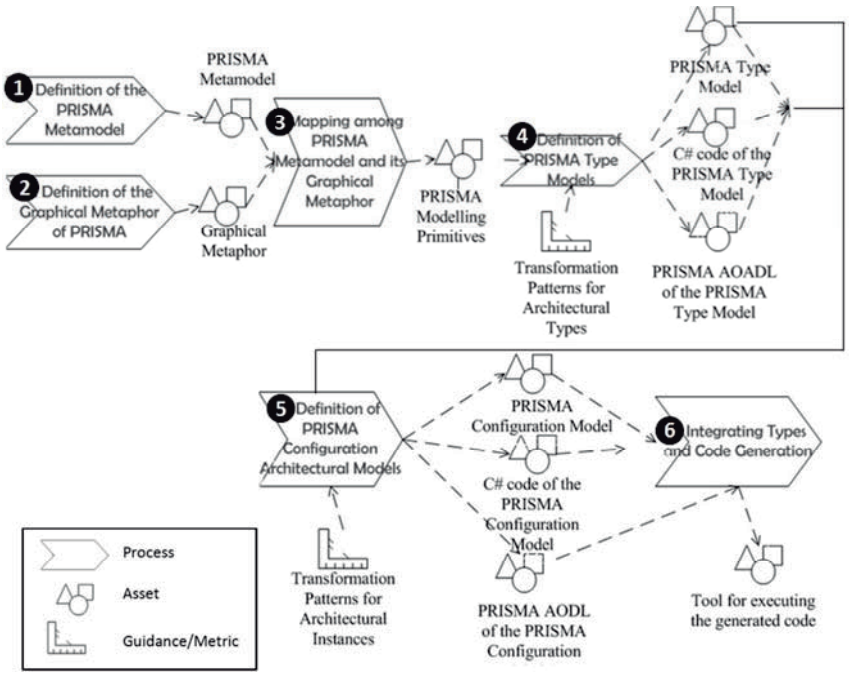


Figure 6: MDD Process from the PRISMA Metamodel to Applications

The PRISMA MDD process is not constrained to the definition of a specific number of levels of abstraction or techniques, because it can vary depending on the needs of each software system. PRISMA follows the MDD approach by enabling software architects to define AOSA models, which allow the complete generation of the final code of AOSAs. The tasks of the software architect are facilitated thanks to the fact that: (i) the level of abstraction provided by models is higher than the provided by programming languages, and (ii) the code is automatically generated from models. PRISMA CASE is the tool that makes the MDD software development of PRISMA feasible. It has been constructed using the Domain-Specific Languages Tools (DSL Tools) of the Microsoft Visual Studio framework [Cook, 07]. From PRISMA AOSA models, PRISMA CASE supports the modelling of PRISMA AOSA models and the generation of both PRISMA AOADL specifications and C# code, which is executable on .NET technology [NET,12]. The PRISMA CASE is composed of the following parts: the specification of the PRISMA metamodel, a graphical modelling tool, a model compiler, the PRISMANET middleware and a generic Graphical User Interface (GUI) to execute the generated code [Pérez, 06a]. The

PRISMA model compiler implements the C# code and AOADL generation patterns of the MDD process of PRISMA. When the model compiler is executed, the patterns are applied to transform the PRISMA architectural models to code, which is executed in the PRISMANET middleware. Finally and PRISMA CASE provides a generic GUI to assist the user in checking the behaviour of the architecture. Thereby, the C# PRISMA software architectures can be executed. In the paper, from now on, to illustrate the contribution, we are going to focus on the C# code generation, i.e. the execution of PRISMA models on .NET platform. And, we are going to use the PRISMA CASE tool and the example of a TeachMover joint to illustrate the PRISMA MDD approach. Figure 6 presents the complete PRISMA MDD process based on MOF, i.e. the different levels of refinement that models undergo during the development process. This process is explained in detail in the following sections.

4.1 From the PRISMA Metamodel to PRISMA Type Models

The PRISMA metamodel defines the PRISMA model and establishes its properties in a precise way. So, PRISMA type models are defined conforming to the PRISMA Metamodel. This definition is described in the following subsections.

4.1.1 Definition of the PRISMA Metamodel

The PRISMA metamodel is defined by a set of inter-related metaclasses (a class of classes) and constraints (see process 1, Figure 6). These metaclasses contain a set of properties and services for each concept considered in the model. Metaclasses, their properties and their relationships define the structure and the information that is necessary to describe PRISMA AOSAs. In addition, the PRISMA metamodel defines the constraints that cannot be specified using the structure or the information of the metamodel [Pérez, 06a]. The structure, information and constraints of the PRISMA metamodel must be satisfied by PRISMA type models in order to ensure that they are correct. One of the most representative packages, that the PRISMA Metamodel is composed of, is the package *Architectural Element* (see Figure 7). This package is going to be used for illustrating the contribution of this paper.

A PRISMA architectural element is defined by the metaclass *ArchitecturalElement* (which is modelled as a UML class). It is an abstract metaclass that specifies the commonalities of the three kinds of PRISMA architectural elements (Components, Connectors and Systems (see Section 2.3.1). To define that an architectural element has ports and weavings, the metaclass *ArchitecturalElement* has two aggregation relationships (the UML Class Diagram aggregation), *has* and *weaves,* with the metaclassess *Port* and *Weaving*, respectively. In addition, the metaclass *ArchitecturalElementen* has one association relationship *imports* with the metaclass *Aspect* to denote that an architectural element imports a set of aspects. Next, these three relationships (*has, weaves* and *imports*) are explained in detail taking into account the semantics that the cardinalities define: (i) An architectural element has at least one port and a port can only be defined as part of an architectural element (see the aggregation *has* in Figure 7); (ii) An architectural element imports at least one aspect and an aspect can be imported by one or more architectural elements of the software system (see the association *imports* in Figure 7); and (iii) An architectural element can include a set of weavings to synchronize its aspects. These Weavings are

related to the architectural element; in fact they can only be defined as part of an architectural element (see the aggregation *weaves* in Figure 7). Also, the metaclass *ArchitecturalElement* has attributes, services and constraints to completely define its properties.

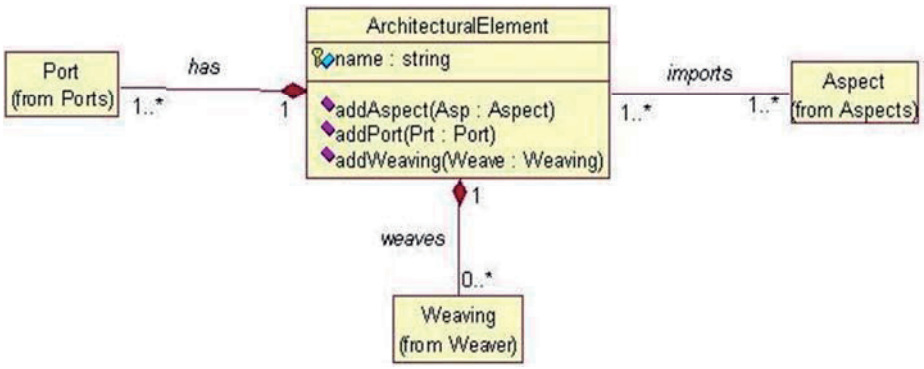


Figure 7: The package ArchitecturalElements of the PRISMA metamodel

With regard to the verification of the PRISMA metamodel, we have distinguished between two kinds of verification: verification rules that must always be satisfied (*hard constraints*), and verification rules that must be satisfied once the model has been completely finished (*weak constraints*).

The hard constraints and weak constraints of the MDD process of PRISMA have been included during the development of PRISMA CASE thanks to the partial C# classes that DSL Tools provide. In DSL Tools, each metaclass of the metamodel is implemented as a partial C# class, which can be extended and customized to add additional behaviour that cannot be included in the metamodel (such as verification rules) [Cook, 07]. In PRISMA CASE, the extension facilities of the partial C# classes have been used for including the needed verification constraints. Thus, since each time that a metaclass is used for modelling a concept in a PRISMA model, its Partial C# class is executed, the constraints of the PRISMA metamodel are verified during the modelling process.

4.1.2 Graphical representation and mapping with PRISMA metamodel concepts

The PRISMA metamodel provides the primitives to model AOSAs. However, to use these primitives in a modeling context, it is necessary to define a graphical representation for using them. PRISMA CASE provides a graphical language to model PRISMA software architectures in an intuitive and friendly way to facilitate the architect tasks (see process 2, Figure 6). The graphical representation that supports the main modelling concepts of PRISMA have been already presented in the paper (see Figures 2, 3, 5 and 10).

Once the graphical representations are defined, each one is associated with its corresponding metamodel concept (see process 3, Figure 6). The PRISMA CASE is generated from the PRISMA metamodel, its graphical representations, and its partial

C# classes. It is composed of a toolbox, a drawing sheet, a model explorer, a window of properties, and a PRISMA menu (see Figure 8).

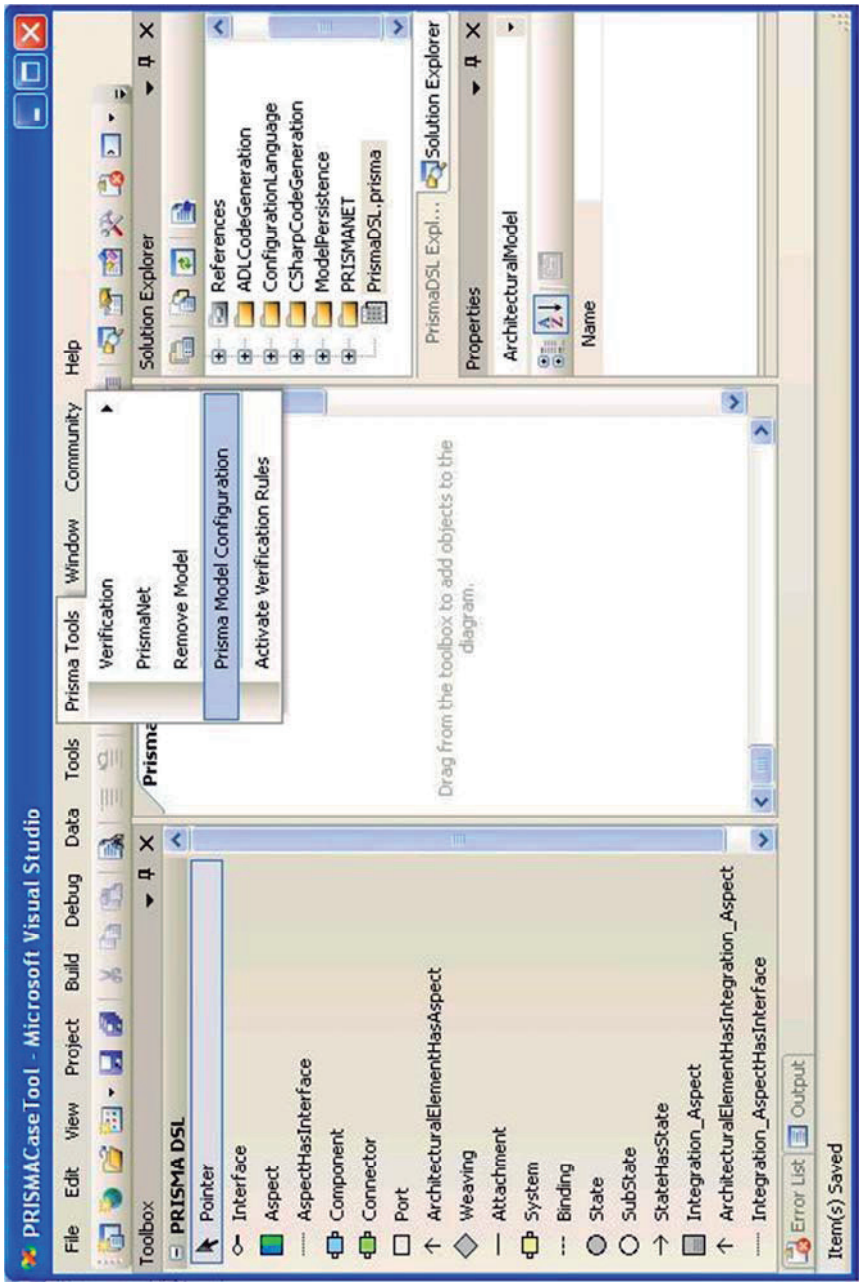


Figure 8: PRISMA CASE Tool: Toolbox, Drawing Sheet, PRISMA Tools Menu, and Solution Explorer and Properties Windows.

4.1.3 Definition of PRISMA type models

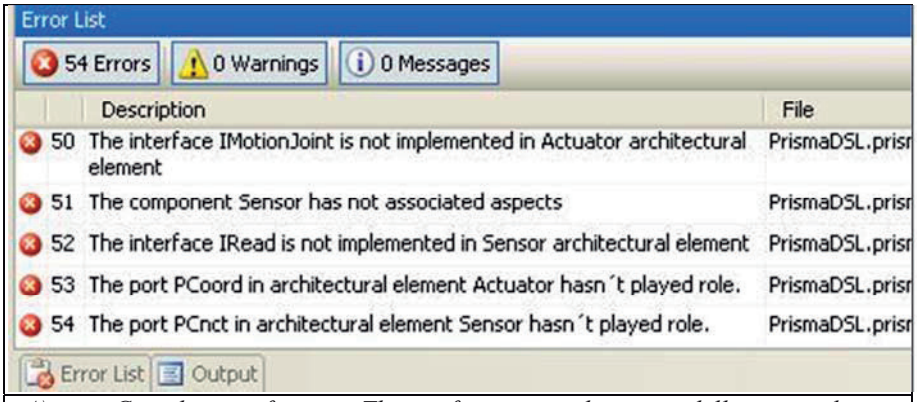
Every PRISMA type model must be defined conforming to the PRISMA metamodel (see process 4, Figure 6). A PRISMA type model is a generic system architecture (banking system, tele-operated robot, electronic auction, etc.) that can be reused for different specific systems. This step is easily developed thanks to the fact that PRISMA type models are described using the concepts that are defined in the metamodel as modelling primitives provided by PRISMA CASE. As a result, the user is able to model PRISMA architectural models and to make sure that these models satisfy the PRISMA metamodel.

The PRISMA MDD process assists the architect by providing mechanisms for the **verification of models**. The verification of models allows the detection of modelling mistakes, and keeps them from spreading throughout the rest of the stages. This is essential in the PRISMA MDD process in order to avoid code generation from incorrect models. Therefore, when a model reaches the last generation step (see Figure 6), it is guaranteed the correctness of the model and the proper performance of the code generation. The verification of architectural models consists of checking that PRISMA type architectural models satisfy the properties and constraints that are defined in the PRISMA metamodel. Specifically, the verification consists of checking that the models satisfy the following properties: (i) the types of a model contain all the information that their metaclasses establish, (ii) the relationships of the model only connect the types which connection is allowed, (iii) the number of types or the relationships between these types is correct, and (iv) the constraints of the metamodel are satisfied. This verification process must always be applied to the modelling process of PRISMA architectural models and must guide the software architect throughout the process. In the modelling tool, the verification of constrains is different depending on their kind. The **weak constraints** act as warnings during the modelling process of **PRISMA type models** (see Figure 9.A), whereas **hard constraints** are verified while the user is modelling.

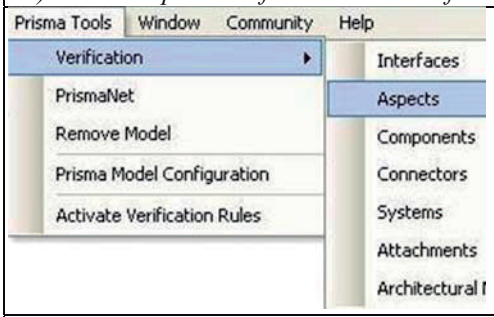
Weak constraints can be violated during the modelling process, but they must be rectified during the modelling process because all of them must be satisfied once the architectural model is finished. For example: in PRISMA, an architectural element must import at least one aspect, and must have at least one port (see Figure 7), but it is possible to define an architectural element without establishing its ports and/or aspect, and to establish them later.

Weak constraints provide more flexibility to the modelling process. The fact that there are weak constraints that are not satisfied means that the modelling process has not finished. However, there may be parts of the architectural model that are finished and the architect may want to verify them. As a result, there are two kinds of verifications that are supported by the PRISMA MDD process: **Partial Verification** and **Complete Verification**. The **Partial Verification** consists of applying only those constraints that affect the elements, concepts or parts of the model that have been selected by the architect for verification (see Figure 9. B and Figure 9.C). This kind of verification allows the architect to define a model, and then verify the model in an incremental way, as well as to verify elements of the model for their later storage in repositories and/or reuse in other models. **Complete Verification** is the verification that is applied to the complete architectural model (see Figure 9.A). As a result, complete verification consists of verifying all the constraints that must satisfy a

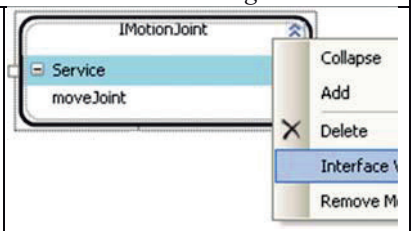
model. In PRISMA, this process implies that all the restrictions of the PRISMA metamodel are checked.



A) Complete verification: The verification result as a modelling error list



B) Partial verification: a specific type of element: Interfaces, Aspects, Components, etc.



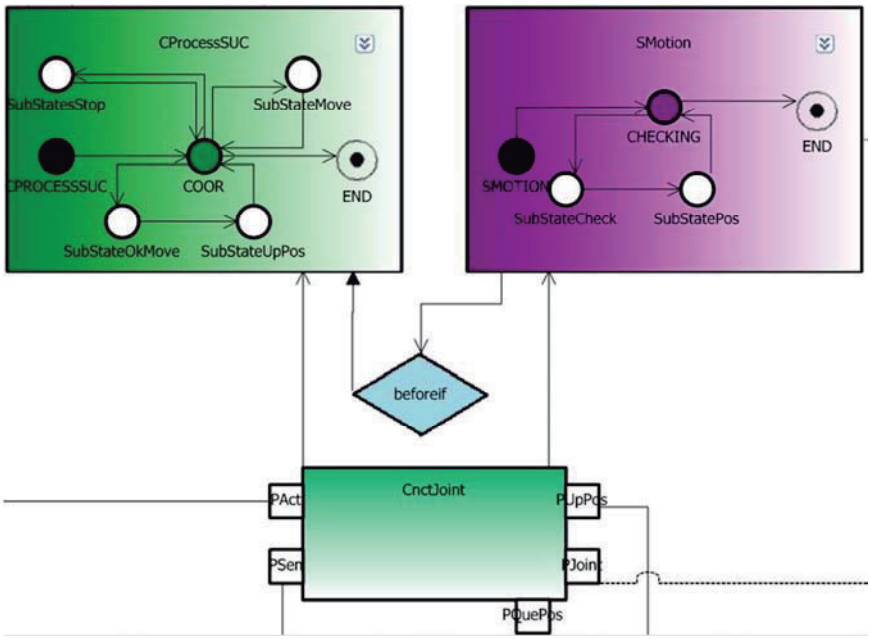
C) *Partial verification: only one element of the architectural model, the interface* IMotionJoint

Figure 9: Partial and Complete Verification of PRISMA Type Modelling Tool

Hard constraints are very close to the graphical metaphor and must always be satisfied without taking into account the modelling process situation. An example of hard constraint is the requirement that a component cannot import a coordination aspect. This is due to the definition of Component: "A component is an architectural element that captures the functionality of software systems and does not act as a coordinator among other architectural elements"(see Section 2.3.1). Since a component is not a coordinator, it never imports a coordination aspect. This hard constraint is materialized in the modelling tool as follows: If an architect associates a coordination aspect with a component, the resulting model would violate the PRISMA model. So, PRISMA CASE does not allow drawing this connection.

In our example, the result of this stage is a PRISMA type model for a generic joint of a tele-operated robot that can be reused for designing different joints of the same robot or joints of different robots. In this PRISMA type model, both components, *Actuator* and *WrapAspSyst,* are coordinated through a connector

CnctJoint (see Figure 5). Each component imports its functionality through aspects, and the connector *CnctJoint* imports its behaviour through a coordination and a safety aspects, *CoordJoint* and *SMotion* respectively (see the appendix A). These two aspects, *CoordJoint* and *SMotion,* are weaved (see Section 2.3.1) to ensure that a joint is moved only after the connector safety constraints are satisfied assuring that a movement is safe. Figure 10 illustrates this weaving, which specifies that the invocation of the *moveJoint* service of *CProcessSUC* implies that the *DANGEROUSCHECKING* service of *SMotion* will be executed *beforeif* the *moveJoint* service of *CProcessSUC*. The weaving condition also establishes that the execution of *moveJoint* must only be performed if the parameter *Secure* of *DANGEROUSCHECKING* returns *true* (see code in Figure 10).




```
Weavings
  Smotion.DANGEROUSCHECKING
    (FTransStepsToAngle(NewSteps), Speed, Secure)
  beforeif (Secure = true)
  CoordJoint.moveJoint(NewSteps, Speed);
End_Weavings;
```

Figure 10: Weaving Definition among Aspects

Once the architect has finished modelling a PRISMA type model, the architect can proceed to generate the C# code corresponding to this model (see process 4, Figure 6). **PRISMA type models** are inputs of the transformations that must be executed to automatically generate part of the C# code of the application. To execute C# PRISMA AOSAs, the .NET platform-specific model for PRISMA has been hard-

wired in its PRISMANET middleware. The PRISMANET middleware is a software layer that sits above the .NET platform and allows the execution of PRISMA applications by offering the aspect-oriented functionalities that .NET does not directly provide. Thereby, PRISMANET implements the PRISMA model by extending the .NET technology with the integration of aspects.

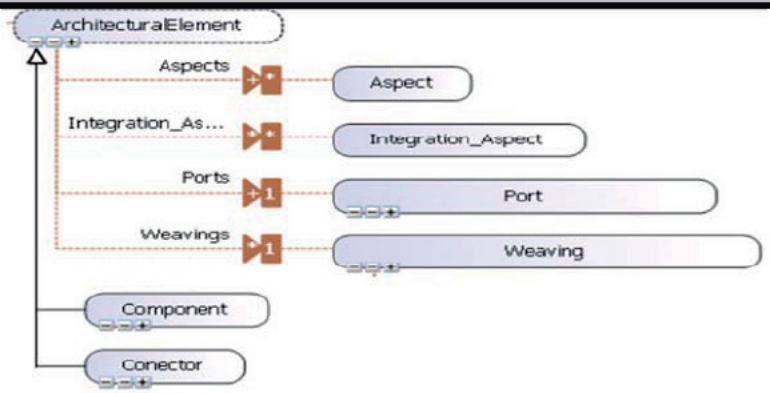
The transformation from models to code is performed using *the code generation patterns for types*. These patterns constitute a catalogue of thirty eight code generation patterns, structured as follows: the patterns PCS1- PCS19 are for generating C# and the patterns PADL1- PADL18 are for generating AOADL (see pattern PC15, Table 1). The patterns PADL1- PADL18 have the mappings between the PRISMA metamodel and AOADL. And the patterns PCS1- PCS19 have the mappings between the PRISMA metamodel and the .NET platform-specific model, i.e., the C# classes that allow instances to be executed on PRISMANET (see the example of the component *Actuator,* Table 1). Since each type that is defined in a PRISMA type model conforms to a metamodel concept, the execution of these mappings applied to the type generates its corresponding C# code. This transformation process (see Figure 6) also contains the mappings between the metamodel and the graphical metaphor because it has been defined in the process 3 of the MDD process (see Section 4.1.2). Hence, the patterns are applied to the graphical models by transitivity: if a concept *gx* of the graphical model corresponds with a concept of the metamodel *mx,* and the concept of the metamodel *mx* corresponds with a concept of the . NET platform-specific model *.NETx,* and a concept of the AOADL *lx,* then the concept *gx* corresponds with the concepts *.NETx* and *lx.* This transformation is possible thanks to the facility of DSL Tools "Transformation Templates", which supports the implementation of code generation patterns [Cook, 07]. The implementation of patterns consists in substituting the parameters of the patterns (see section *template*, Table 1) by the elements that the architect had modelled (see section *graphical representation,* Table 1). However, it is important to keep in mind that the result of this generation consists of reusable C# classes that still are not directly executable: they are architectural element types that must be instantiated to configure a specific system.

Pattern PCS15 presents the transformation template for architectural elements and shows an example of the pattern execution, the component *Actuator* (see section *result of the pattern execution*, Table 1). The pattern generates a component implemented as a serializable C# class (see Table 1, Pattern PCS15, Section 4.3, line 4) as the transformation template establishes (see Table 1, Pattern PCS15, Section 3.2, line 7). This class is *serializable* to enable its mobility in distributed versions of PRISMA CASE. Since the architectural element is a component and it is named *Actuator,* a public class called *Actuator* is generated as a subclass of the *ComponentBase* class of PRISMANET (see Table 1, Pattern PCS15, Section 4.3, line 5), as the template establishes for the generation of components (see Table 1, Pattern PCS15, Section 3.2, lines 6 and 8). This .NET class *ComponentBase* implements the .NET specific component behaviour of the PRISMA model. Then, the constructor of the class is created (see Table 1, Pattern PCS15, Section 4.3, line 6), as the template establishes (see Table 1, Pattern PCS15, Section 3.2, lines 12-13). Finally, the set of ports and aspects that make up a component are included by invoking the constructors of the *port* and *aspect* PRISMANET classes (see Table 1, Pattern PCS15, Section 3.2,

line 14). Both classes implement the port and aspect elements of the PRISMA model generating the ports and required aspects (see Table 1, Pattern PCS15, Section 4.3, lines 7-11).

Pattern PCS15: Simple Architectural Elements

1. PRISMA metamodel in DSL Tools



2.Graphical Representation



3.Transformation

3.1.Description

This pattern details how to generate the C# code from a simple architectural element. Specifically, it only generates the structure of the architectural model. The internal code of this structure, that is, ports, aspects and weavings are generated by other patterns related to this pattern.

3.2.Template

```
...
1 using System;  using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types;
2 using PRISMA.Components; using PRISMA.Middleware;
3 namespace <#=this.Model.Name#>
4 {
5 <#foreach (ArchitecturalElement architecturalElement in this.Model.ArchitecturalElements){
6  if (architecturalElement is Component || architecturalElement is Connector)  {#>
7  [Serializable]
8  public class <#=architecturalElement.Name#> : ComponentBase
9  <#    if (architecturalElement is Connector) {#>
10    , IConnector
11 <# } #>
12  { public <#=architecturalElement.Name#>
13    (string name<#=ArchitecturalElementArguments(architecturalElement)#> ) : base(name) {
14<#/* Aspects */ /* Weavings */ /* Ports */ #>
15  } }
16 <#
17 }/* endif (architecturalElement is Component || architecturalElement is Connector)*/...
```

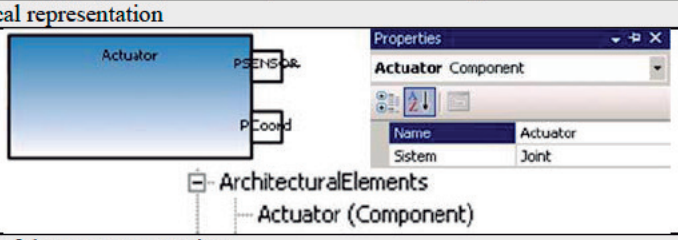
Pattern PCS15: Simple Architectural Elements

4. Case Study

4.1.Description

This pattern is illustrated using the component *Actuator* of the TeachMover case study. The representation of the *Actuator* in the PRISMA model and the C# code generated from this model by applying this pattern are presented as following.

4.2.Graphical representation



4.3. Result of the pattern execution

```
...
1 using System.Reflection; using PRISMA; using PRISMA.Aspects;
2 using PRISMA.Aspects.Types;using PRISMA.Components;using PRISMA.Middleware;
3 namespace RobotJoint
4 {[Serializable]
5   public class Actuator : ComponentBase{
6    public Actuator(string name ) : base(name){
7      /* Aspects */  AddAspect(new RS232 ()); /* Weavings */
8      /* Ports */InPorts.Add ("Pcoord", "IMotionJoint", INTMOVE);
9                 OutPorts.Add ("Pcoord", "IMotionJoint", INTMOVE);
10                 InPorts.Add ("PSENSOR", "IMotionJoint", OUTMOVE);
11                 OutPorts.Add ("PSENSOR", "IMotionJoint", OUTMOVE);
12               }}}...
```

Related Patterns

Pattern PCS16: Aspects importation, **Pattern PCS17:** Weavings and **Pattern PCS18**: Ports.

Table 1: PRISMA Code Generation pattern PCS15: Simple Architectural Elements

4.2 From PRISMA Type Models to PRISMA Configuration Models

The definition of PRISMA configuration models from PRISMA type models comprises processes 5 and 6, illustrated in Figure 6. Next they are described.

4.2.1 Definition of PRISMA Configuration Models

Every **configuration model** must conform to a **PRISMA type model**. A specific configuration instantiates a generic architecture to specify a particular system. An example is the *TeachMover* robot or a specific joint of a robot. This step is easily developed thanks to the fact that PRISMA configurations are defined using the concepts that are defined in **its PRISMA type model** as modelling primitives. They are provided by PRISMA CASE, which automatically generates a domain-specific graphical modelling tool to configure specific software architectures from the type models. This is possible by generating a new DSL Tools project from the output of the PRISMA type model (see Figure 11). The generation of the domain-specific graphical modelling tool is launched from the PRISMA Type Modelling Tool (see Figure 8, menu PRISMA Tools).

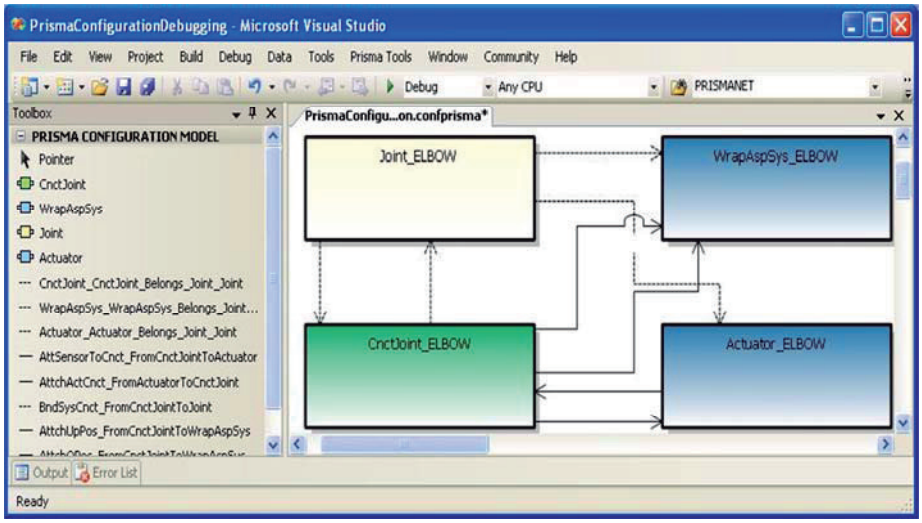


Figure 11: The architecture configuration of the joint Elbow

Figure 11 illustrates a domain-specific graphical modelling tool generated from the Type model of the *TeachMover*. Note that the elements of the toolbox are the components *Actuator* and *WrapAspSyst,* the connector *CnctJoint* and the connections (attachments and bindings) that were previously defined in the PRISMA type model of a Joint for Tele-Operatated Robots (see Figure 5 and Appendix A). In our example we have configured **the configuration model** of the *Elbow* of the *TeachMover* robot by dragging and dropping the primitives on the drawing sheet.

It is important to keep in mind that the **verification of architecture configurations** is also really important. It consists of checking that a configuration of instances satisfies the architectural model that it is an instance of (i.e., interconnections and compositions among instances are compliant with the interaction and composition patterns of the architectural model). In these verifications it is also considered the difference between **weak constraints** and **hard constraints** to support partial and complete verification as it is provided in PRISMA type models (see Section 4.1.3).

4.2.2 Integrating Types and Instances by means of Code Generation

Once the architect has finished the PRISMA configuration model, can proceed to generate the C# application corresponding to the **configuration model** (see process 5, Figure 6). The configuration model, together with the type model the code generated for these type model, are the inputs of the transformations that must be executed to automatically generate the final code of the application.

The transformation from models to code is performed using other patterns for architectural instances. In the case of C#, these patterns implement the mappings between a PRISMA type model (in our example, the Joint of a tele-operated robot) and the .NET platform-specific model (i.e., the C# primitives that allow PRISMA

AOSAs to be executed on PRISMANET). Since each instance that is defined in a PRISMA configuration model is an instance of a type of its PRISMA mode (e.g. the System instance *Elbow* is a System type *Joint* (see Figure 11), the mappings of the PRISMA model concept with PRISMANET and with the AOADL are executed generating its corresponding C# code. In this transformation step, the code obtained from process 5 of Figure 6 is completed by generating the C# instances. These instances can be launched and executed on PRISMANET. In order to do this, the PRISMA CASE executes the middleware PRISMANET and instantiates the defined configuration.

As a result of this execution, a generic GUI is launched to interact with the architecture by invoking its services and checking the value of its attributes (see process 6, Figure 6). The main purpose of the generic GUI is to assist the user in checking the behaviour of the architecture without having to worry about aesthetic details and without forcing the user to define a GUI in order to obtain a result. However, it is important to mention that the use of this interface is not mandatory. In other words, if the users prefer to define their own specialized GUIs, they can do so.

5 Related Work

The combination of AOSD and software architectures has created two new challenges: (i) how to define the concept of aspect at the architectural level? and (ii) how to integrate aspects and architectural elements in a suitable way? This section analyzes the most relevant approaches that deal with these two questions and how they satisfy the main set of desirable properties that any aspect-oriented architecture MDD approach should fulfil (see Section 2.2.3). This section also discusses our contribution taken into account the state of the art of AOSAs, paying special attention to their role in the software development process and their MDD support.

5.1 Analysis of the main Aspect-Oriented Software Architectural approaches

Several approaches have emerged to integrate aspects and software architectures, as [Fiadeiro, 04] promoted, either by extending original ADLs with aspects or by creating new ADLs from scratch. AOSA approaches can introduce aspects in their ADLs in different ways: as a component (AspectLEDA [Navasa, 05], JAC [Pawlak, 04], Jiazzi [McDirmid, 03]); as a connector among components (CAM/DAOP [Pinto, 03]); as a view of the architecture (AVA [Katara, 03]), etc. On the other hand, there are approaches such as FuseJ [Suvée, 06], [Suvée, 05] that promote the idea that there are no aspects because aspects can be modelled using components such as FuseJ, AspectLEDA, JAC, Jiazzi and others. However, as Kizcales explains in his work [Kiczales, 97], the semantics of these concepts is different; i.e., components and aspects are not the same thing. This is due to the fact that aspects can be modelled as components, but components cannot usually be modelled as aspects since a component generally implements properties of more than one concern. Thanks to the introduction of AO techniques to all the stages of the software life cycle, it is possible to take advantage of the notion of aspect from the beginning of the system definition by specifying the aspects that are found in the requirements specifications. As a result, these aspects can be used throughout the rest of the development process as well as in

the maintenance process. However, approaches such as AspectLEDA and TranSAT [Barais, 03], [Barais, 04] require having an initial architectural specification of the system to introduce aspects. In these two approaches, aspects can only emerge as new requirements of the system that are hooked to the base architecture, i.e. aspects are only used for evolution during the maintenance process and not during the development process. TranSAT is an approach for managing the evolution of software architecture specifications using AOP principles. The fact that TranSAT concerns are only technical and not more generic is another drawback. Thus, TranSAT software architectures are defined using a pure compositional ADL, and aspects only appear as an extension or evolution mechanism of software architectures.

In the AVA approach, the complete view of the software architecture is lost because the use of the view notion is required to define aspects in its software architectures. AVA uses concerns as viewpoints to obtain those views that are the aspects of the software architecture. However, the use of architectural views from an AO point of view should be considered as an additional mechanism for analyzing software architectural features, instead of limiting the use of aspects to solely defining views. In fact, in software development not only is it possible to define views using aspects, but it is also possible to use common criteria such as roles, different stakeholders, etc. As a result, this approach loses flexibility and expressiveness to define views.

As Shaw presents in her work [Shaw, 94] , the specification of software systems with complex coordination protocols is too difficult without the connector architectural element. This is because the connector provides separation of component interactions, thereby achieving a higher level of abstraction, modularity, and a greater architectural view of the system. For this reason, connectors should be first-class citizens of an original ADL. However, there are many ADLs that do not provide the notion of connector. From these ADLs, many AOADLs such as CAM/DAOP, AspectLeda, Jiazzi, AOCE [Grundy, 99] or JAC have been defined. CAM/DAOP introduces aspects as connectors, JAC models aspects as components, and the approaches of AspectLEDA, Jiazzi and AOCE implement weavings as connectors in order to coordinate their “component aspects” (aspects are modelled as components and are called component aspects). Therefore, they use the connector as their new primitive to model aspects or weavings. It also introduces the weaving process inside their “component aspects” losing the reusability feature of aspects, since component aspects are dependent on the component aspects that they are connected to. In addition, ADLs should be formal languages because it is the only way to reason about the obtained software architectures, validate properties and generate code. However, we lose the advantages of using an extended language such as XML or Java, as in the CAM/DAOP and FuseJ proposals, respectively.

It is important to mention that a few of these approaches that combine AOSD and software architectures provide support for complete development of software; i.e. generating code from AOSA models. Therefore, they do not provide complete MDD support for developing AOSAs through a tool. The PCS approach is supported by the ConcernBase tool [Kande, 03]. This tool supports MDD in a partial way. It provides mechanisms for modelling software systems, and it also allows the translation of UML models to the SADL language [Moriconi, 97]. Technology independence is a clear advantage of this approach. However, at the same time, it is a drawback of PCS

because it does not provide support to translate its models to a programming language and they cannot be executed on a specific technological platform. The CAM/DAOP approach provides the DAOP platform [Pinto, 05]. It has been implemented in Java, and it provides a middleware to support the execution of aspects, components, and the dynamic weaving between them over the Java technology. The platform and the DAOP-ADL specifications are integrated because the input of the DAOP platform is the XML document that contains the specification of the architectural model in XML. The XML document contains the information needed to instantiate components and aspects. For this reason, when the document is loaded by the DAOP platform, the instantiation of components and aspects starts taking into account the instantiation information defined in the document. The work of Fuentes et al. [Fuentes, 03] of CAM/DAOP is a first step to support MDD in the DAOP platform; however, complete support using code generation techniques for the development is not provided. It uses MDA to show the different views of the models that are specified in the platform. The TranSAT framework consists of a tool called SafArchie Studio [Barais, 03]. This tool offers several views of the evolution process depending on the kind of user. It only supports a tool to analyze the evolution of the software architecture and does not develop the aspect-oriented software architecture application. The MADE tool has been developed to support the AVA approach [Hammouda, 04]. It also shows the different views of the architecture, but it does not provide complete MDD support. AOCE has a tool to support its methodology, which extends the JViews [Grundy, 00] tool to support aspects. This tool is called JComposer [Grundy, 98]. JASCO provides two different tools. One of the JASCO tools transforms a Java bean into a JAsCo bean, and the other one integrates JAsCo with the PacoSuite [Wydaeghe, 01], which allows component models to be modelled at a high abstraction level and also allows one or more JAsCo connectors to be generated from its models. However, the JComposer and the pair of tools that support the JAsCo approach are dependent on technology. They are, in fact, implementation frameworks.

5.2 Comparison of the main Aspect-Oriented Software Architectural approaches

Two comparison tables[2] have been defined from the criteria established in Section 2.2.3 and the approaches studied in Section 5.1 to analyze and compare them (see Table 2 and Table 3). After the analysis and comparison of different approaches of AOSA, it may be concluded that, at the architectural level, these proposals usually extend ADLs without connectors and mainly follow an asymmetric model by considering functionality as architectural components. They always introduce the notion of aspect by using original architectural concepts instead of providing the suitable semantics for aspects. Furthermore, despite the fact that there has been a lot of work done, these proposals are only focused on a specific purpose: the analysis, evolution or development of software architectures. And, they do not pursue several purposes simultaneously to provide complete development and maintenance support.

[2] Blank cells indicate that no information was available.

	Aspect-oriented model	Architectural model	Definition of Aspects	Definition of Weavings	ADL
PCS	Multidimensional and symmetric	Without connectors	Aspects like connectors	Inside aspects	SADL: Formal compositional ADL
CAM/ DAOP	Asymmetric	Without connectors	Aspects like connectors	Outside aspects using communication between interfaces	DAOP–ADL: Not formal, based on XML
Superimposition	Asymmetric: Two levels: aspects and architectures		Java Classes inside a superimposition layer	Inside aspects	
TRANSAT	Asymmetric. Only technical aspects	Without connectors	Aspects like components. Aspect components	Outside aspects. Using adapters or weavers ≅ connectors	SafArchie component model
ASAAM	Asymmetric	Not fixed	Scenarios	Outside Aspects	Not fixed
AVA	Asymmetric	Not fixed	Aspects as views	Outside Aspects	Not fixed
Aspect LEDA	Asymmetric: Two levels: Aspects and architectures	Without connectors	Aspects as components	Outside aspects using coordinators ≅ connectors	Leda: Formal Compositional ADL
AOCE	Asymmetric	Without connectors	Aspects as components	Outside aspects using aspect managers ≅ connectors	
Component Views	Asymmetric		Not aspects. Concerns as viewpoints for defining architectural views		
Aspectual Components	Asymmetric: Two levels: Aspects and object-oriented applications		Aspects as components: Aspectual components	Outside aspects with connectors	
Caesar	Asymmetric		Aspect Collaboration Interface (ACI)	Separation of ACI modules into implementation and interaction of aspects	
JASCO	Asymmetric		Aspects	Hooks and connectors	
FUSEJ		With Connectors	Without Aspects: Components	Connectors	
JAC	Asymmetric		Aspects as components: aspectcomponents	Inside aspects	
JIAZZI	Asymmetric: Two levels: Aspects and object-oriented applications		Units	Linking units	

Table 2: First comparison of aspect-oriented software architecture approaches

	Aspect-Oriented Evolution	Purpose	Technology	Graphical support	Tool support
PCS		Development of AO Software Architecture	Independent	UML profile: Aspect is a stereotype of a UML class	ConcernBase tool: modelling support, ADL generation from UML, no code generation, no execution
CAM/DAOP	Dynamic weaving but not adding and removing aspects at run-time	Development of AO Software Architecture	Independent	UML profile	DAOP platform: Java Technology, modelling support, DAOP middleware for code execution
Superimposition		Programming aspect-oriented Java applications and verifying properties of aspect-oriented superimposition	Dependent on Java technology		
TRANSAT		Only evolution support, the initial aspect-oriented specification is not supported.	Independent	UML profile	SafArchie Studio. Extension of ArgoUML
ASAAM		Analysis of Software Architectures	Independent	UML profile: scenarios	ASAAM-T
AVA		Development of AO Software Architecture	Independent	UML profile: aspect is an stereotype of a UML package that contains an extension of component diagram	MADE tool: modelling support
Aspect LEDA		Development of AO Software Architecture	Independent		
AOCE	Dynamic weaving	Development of AO Software Architecture	Dependent on JViews		JComposer: An extension of the JViews tool
Component Views		Analysis of software architectures	Independent	UML profile	
Aspectual Components		Programming aspect-oriented Java applications	Dependent on Java technology		
Caesar		Programming aspect-oriented Caesar applications	Dependent on Caesar programming language		Programming framework

	Aspect-Oriented Evolution	Purpose	Technology	Graphical support	Tool support
JASCO	Dynamic weaving and support for adding and removing aspects at run-time	Programming aspect-oriented application	Dependent on Java or .NET technology		Programming framework
FUSEJ		Programming aspect-oriented applications onto Java Beans	Dependent on the Java Beans component model		
JAC		Programming aspect-oriented Java applications	Dependent on Java technology		
JIAZZI		Programming aspect-oriented Java applications	Dependent on Java technology		

Table 3: First comparison of aspect-oriented software architecture approaches

Therefore, it may be concluded that an approach of AOSAs for symmetric AOMs and ADLs with connectors that follows the MDD paradigm should be defined in such a way that the starting premise would be fulfilled. This approach should include: (i) a suitable semantics for the aspect concept; (ii) a graphical modelling metaphor, (iii) technological support in order to execute the aspect-oriented architectural models that have been defined independently of technology, (iv) guided support throughout the development and maintenance processes of software following MDD: Reusability, Verification, Code generation, Maintenance, Evolution, etc. The PRISMA MDD approach was defined to fulfil these needs by completing the Tables 2 and 3 as follows:

	Aspect-oriented model	Architectural model	Definition of Aspects	Definition of Weavings	ADL
PRISMA	Symmetric	With Connectors	Aspects	Outside aspects and inside architectural elements	Formal AOADL

	Aspect-Oriented Evolution	Purpose	Technology	Graphical support	Tool support	MDD support
PRISMA	Dynamic weaving and support for adding and removing aspects	Development and Maintenance of aspect-oriented software architectures	Technology independent	Graphical AOADL	PRISMA CASE	Complete MDD support

6 Experimental Results: PRISMA as an step forward in the MDD process of Aspect-Oriented Software Architectures

This PRISMA MDD process has been applied to different kind of applications to validate its feasibility. It has been used for the design and code generation of academic examples such as, banking systems, auctions, etc. But also, it has been put into practice for the generation of real applications that are deployed in the industry. Specifically, PRISMA has been put into practice in the robotic domain. In particular, the PRISMA MDD process has been applied to the development of the robots

TeachMover [Pérez,08a] and *Agrobot* [Costa-Soria, 11], as a proof-of-concept for this MDD process.

6.1 Validation Goal and Variables of measurement

The validation of our approach has been focused on the code generation capability of the PRISMA MDD process. The analysis of this capability has been driven by following question: *Is the PRISMA MDD process able to automatically generate the executable C# code of applications from Aspect-Oriented Software Architectural models?*.This main question has been refined into two more concrete questions:

Q1) *Is the code generation complete or partial?*, and

Q2) *Is the code generation ready to be executed or it requires any update for its execution?*.

These two questions aims to find out the degree of automation of the process and allow us to define the response variables for quantitatively measuring the PRISMA MDD process. These variables are the following: Percentage of Generation (%G), Percentage of Manually Programmed Code(%MPC), Percentage of Updated Code (%UC), Number of Generated Lines of Code (NGLC), Number of Manually Programmed Lines of Code (NMPLC), Number of Updated Lines of Code (NULC), Number of Generated Classes (NGC), Number of Manually Programmed Classes (NMPC), Number of Updated Classes (NUC), Number of Generated Interfaces (NGI), Number of Manually Programmed Interfaces (NMPI), and Number of Updated Interfaces (NUI) . In addition, other variables in terms of aspect-oriented software architectures concepts has been collected: Number of Lines of Code for Interfaces (NLCI), Number of Lines of Code for Aspects (NLCA), Number of Lines of Code for Architectural Elements (NLCAE), Number of Lines of Code for Components (NLCC). Number of Lines of Code for Conectors (NLCCN), and Number of Lines of Code for Systems (NLCS).

6.2 Description of the Robotic Applications.

In this paper, we have used part of the architecture of the tele-operated robot *TeachMover* to illustrate the PRISMA MDD process. The detailed description of the functionality of this tele-operated robot is described in Section 3. The complete architecture of the robot TeachMover consists of 16 architectural elements (5 Systems, 6 components and 5 connectors), and 16 aspects (6 functional aspects, 5 safety aspects and 5 coordination aspects) [Pérez, 08a].

In addition, the PRISMA MDD approach has been applied to the code generation of the agriculture robot *Agrobot* [Costa-Soria, 11]. The Agrobot is conceived as an autonomous, small-sized robot, which objective is to patrol –at periodical intervals- a small field or delimited area, looking for pests or disease attacks over a set of growing crops. When a threat is detected, a pesticide is applied to the field, as a first counter-attack measure, and an alarm is sent to the manager to take further specialized actions. The use of small agricultural robots (small-sized, light-weight, and autonomous) [Blackmore, 06] is being encouraged in the Spanish agriculture sector to reduce the high labour involved and the production costs of plague control. The *Agrobot* architecture is hierarchically defined as a composition of Systems. The top-level architecture consists of 12 Systems and 10 connectors. From these systems, the MDD

process has been applied to one of them, the System which allowed us to validate its behaviour without being geographically in an agriculture field: the Vision System [Costa-Soria, 2011]. The Vision System captures and pre-filters images from the environment, which are used by other Systems of the Agrobot to look for crop diseases and/or guide the movement. This System has been completely modelled using the PRISMA CASE tool, and its code has been automatically generated. This System consists of 8 architectural elements (1System, 4 components, and 3 connectors), 8 aspects (4 functional aspects, 3 coordination aspects, and 1 presentation aspect) and 3 interfaces.

6.3 Analysis of the Experimental Results

The PRISMA MDD process has been applied to the development of both robots. The results obtained from the measurement of the variables identified in Section 6.1 are described in Tables 4 and 5.

	%G	%MPC	%UC	NGC	NMPC	NUC	NGI	NMPI	NUI
TeachMover	94,3	5,7	0	32	2	0	19	0	0
Agrobot	68,1	20,09	10,90	16	2	4	3	0	0

Table 4: Result of the robot TeachMover and Agrobot in percentages, nº classes and nº interfaces

	NGLC	NMPLC	NULC	NLCI	NLCA	NLCAE	NLCC	NLCCN	NLCS
TeachMover	5686	345	0	146	5215	325	99	150	65
Agrobot	1125	345	180	28	1388	234	109	84	41

Table 5: Results of the robot TeachMover and Agrobot in Lines of Code

The results presented in Tables 2 and 3 reveals a high percentage of automatic-code generation from PRISMA AOSA models: 94,3% for the *TeachMover* and 68,1% for the *Agrobot*, which correspond with 5686 and 1125 Generated Lines of Code (NGLC), respectively (see Tables 4 and 5). The percentage of lines of code manually programmed (NMPLC) of both applications correspond to the 345 lines of code that implement the Generic Graphical User Interface (GUI) of PRISMA (see Tables 4 and 5). This GUI allows the execution of any PRISMA application, and both robots are executed using this GUI. As a result, both robots have the same number of lines manually programmed, and the same Manually Programmed Classes (NMPC) (see Tables 2 and 3). Since this number is constant, the percentage of generation varies depending on the extension of the system. In this case, as the TeachMover is more extensive than Agrobot, the percentage of automatic generation is higher.

On the other hand, the number of classes (see Tables 2 and 3) illustrates that the PRISMA MDD process generates one class for each concept modelled, e.g. the 32 generated classes (NGC) of the TeachMover (see **Fehler! Verweisquelle konnte nicht gefunden werden.**) correspond to the 16 architectural elements and 16 aspects of its PRISMA aspect-oriented architectural model.

With regard the number of lines of code that have been generated, most of them are related to the body of aspects, which define state (attributes), services (methods), and protocols (the valid transitions among services).

As a result of the modelling and code generation process of the Vision System of the Agrobot, the experiment showed that some behaviour could not be modelled and must be directly introduced on the generated code. This was the case for low-level behaviour: the code responsible for interacting with the image capturing device, and the algorithms for processing and filtering the images captured. In this case, the approach was to replace the body of the services automatically generated with the specific, low-level code. This resulted in 180 lines of code updated (see NULC in Table 5), which in total represented only the 10,9% of the total amount of code (see %UC in Table 3). This is significantly better that developing all the code from the beginning.

These code generation results reveal that the complete automatic-code generation from AOSA models is feasible. In this contribution we provide an approach that takes a step further in the automatic code generation from aspect-oriented software architectures. The PRISMA MDD process is able to generate the code of the business logic, but is not able to automatically generate its GUI. This establishes the next step to improve this MDD process and its modelling framework.

6.4 Evaluation of validity and limitation

This approach has been validated by automatically generating the code of two real applications from their aspect-oriented architectural models. However, it is difficult to generalize the results because, as any experiment, there is a set of factors that affects the results of the response variables. In this case, the main factors that have been identified are: 1) the project size, 2) the architecture complexity, and 3) the concerns modelled.

The experimentation field recommends to intentionally vary these factors to mitigate the variations that could impact the results and facilitate the generalization of these results [Juristo, 01]. Following this guideline, we have applied the PRISMA MDD process to two applications with different size and complexity. However, to completely mitigate these results, it is necessary to apply the process to other domains, in which other concerns should be required, such as persistence, graphical-user interface or privacy in information systems or social network services.

7 Conclusions and Future Work

In this paper, the PRISMA MDD process of PRISMA is presented as an important advance in the automatic generation of code from AOSAs. This contribution describes this process and each one of their generation steps to serve as a first guidance for the MDD of AOSAs. This process also defines two steps of transformation that are supported by generation patterns that allow the generation of AOADL specification and C# code. Hence, this contribution provides a pattern template to describe the transformations of code generation. Moreover, the need to support model verification throughout the MDD process is established by defining not only the kind of verifications (**partial, incremental, complete)**, but also the kind of

constraints (**hard constraints** and **weak constraints)** to consider. It is important to emphasize that the refinement from the **types model** to the **configuration model** of PRISMA MDD process provides a **domain-specific model** to configure PRISMA software architectures. This domain-specific model is important because it reduces the gap between the user and his/her knowledge about modelling during the MDD process.

This PRISMA MDD process has been materialized into the PRISMA CASE framework, which supports each one of the steps that the process establishes. PRISMA CASE has allowed us to apply the process to the *TeachMover* and *Agrobot* robots. In this paper, we present the code generation results of both applications and the analysis reveals that the complete automatic-code generation from AOSA models is feasible with the presented MDD process,

PRISMA is a new approach that opens a perfect setting for further research. All the parts that the PRISMA approach is composed of can be extended in order to face new challenges. This MDD process can be enriched by introducing more layers of refinement or defining new transformations for other languages, models or platforms and/or developing abstract middleware that would hide the differences between the different platforms. Another extension to this MDD process is the incorporation of COTs throughout the process. Yet another task is to create a repository with a query language and metadata description of the architectural elements and aspects to improve reusability even more. Finally, it is necessary to evaluate PRISMA using applications of other domains, which could allow us to generalize the obtained results..

Acknowledgements

The work reported here has been partially sponsored by the Spanish MEC projects (DSDM TIN2008-00889-E and MULTIPLE TIN2009-13838), and MICINN (INNOSEP TIN2009-13849)

References

[Aksit, 05] Aksit, M., Systematic analysis of crosscutting concerns in the model-driven architecture design approach. Symposium How Adapatable is MDA?, 2005.

[Amaya, 05] Amaya, P. A., González, C. F., & Murillo J. M., MDA and separation of aspects: An approach based on multiple views and subject oriented design. AOM, AOSD, Chicago, USA, 2005.

[Barais, 04] Barais, O., Cariou, E., Duchien, L., Pessemier, N., & Seinturier, L., Transat: A framework for the specification of software architecture evolution. The 1st Int. Workshop on Coordination and Adaptation Techniques for Software Entities, Oslo, 2004.

[Barais, 03] Barais, O., Duchien, L., & Pawlak, R, Separation of Concerns in Software Modeling: A Framework for Software Architecture Transformation. IASTED Int. Conf. on Software Engineering Applications, ACTA Press, pp. 663-668, Los Angeles, CA, USA, 2003

[Beydeda, 05] Beydeda, S., Book, M., & Gruhn V., Model-Driven Software Development, Springer, 2005.

[Blackmore, 06] B.S. Blackmore, H.W. Griepentrog, S. Fountas. Autonomous Systems for European Agriculture. In proc. of Automation Technology for Off-Road Equipment (ATOE). Bonn, Germany, 2006.

[Chitchyan, 05] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., et al.: Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. AOSD-Europe Deliverable D11, 2005.

[Cook, 07] Cook S., Jones G., Kent S., Cameron Wills A., Domain-Specific Development with Visual Studio DSL Tools, ISBN-10: 0-321-39820-3, Addison-Wesley, 2007.

[Costa-Soria, 11] Costa-Soria, Cristobal: Dynamic Evolution and Reconfiguration of Software Architectures through Aspects. PhD Thesis.Universidad Politécnica de Valencia, June 2011.

[Cuesta, 05] Cuesta, C., Romay, M.P., De La Fuente, P., & Barrio-Solórzano, M., Architectural Aspects of Architectural Aspects. 2nd European Workshop on Software Architecture, LNCS 3527, Pisa, 2005.

[Cuesta, 02] Cuesta C.E.. Dynamic Software Architecture based on Reflection. PhD. Thesis, Dpt. of Computer Science, University of Valladolid, 2002. (In Spanish)

[Dijkstra, 76] Dijkstra, E., *A Discipline of Programming*. Prentice-Hall, 1976.

[EFTCoR, 02] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794, 2002.

[Fernández, 05] Fernández C., Pastor J.A., Sánchez P., Álvarez B., Iborra A., Co-operative Robots for Hull Blasting in European Shiprepair Industry. IEEE Robotics and Automation Magazine (RAM), September 2005.

[Fiadeiro, 04] Fiadeiro, J.L., & Lopes, A, CommUnity on the Move: Architectures for Distribution and Mobility. FMCO 2003, LNCS3188, pp. 177–196. Springer Heidelberg, 2004.

[Fuentes, 03] Fuentes. L., Pinto. M., & Vallecillo A. How MDA can help designing component- and aspect-based applications. EDOC, pp. 124-135, 2003.

[Garlan, 95] Garlan, D., Perry D., Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, vol. 21 no. 4, April 1995.

[Grundy, 00] Grundy J.. Multi-perspective specification, design and implementation of software components using aspects. Int. Journal of Software Engineering and Knowledge Engineering, vol. 20, 2000.

[Grundy, 99] Grundy, J. Aspect-Oriented Requirements Engineering for Component-based Software Systems. The 4th IEEE Int. Symp. on RE, 1999.

[Grundy, 98] Grundy, J.C., Mugridge, W.B., & Hosking, J.G.. Static and dynamic visualisation of component-based software architectures. The 10th Int. Conf. on Software Engineering and Knowledge Engineering, KSI Press, San Francisco, California, USA, 1998

[Hammouda, 04] Hammouda, I., Koskinen, J., Pussinen, M., Katara, M., & Mikkonen, T., Adaptable Concern-Based Framework Specialization in UML. Automated Software Engineering, pp. 78-87, Linz, Austria, 2004.

[Harrison, 02] Harrison, W., Ossher, H., & Tarr, P., Asymmetically vs Symmetrically Organized Paradigms for Software Composition. IBM T.R. RC22685 (W0212-147) Thomas J. Watson Research Center, IBM, 2002.

[Juristo, 01] Juristo N., Moreno A.M., Basics of software engineering experimentation, Kluwer, 978-0-7923-7990-4, pp.1-395, 2001.

[Kande, 03] Kande M., A concern-oriented approach to software architecture. PhD. Thesis, Lausanne, Switzerland: Swiss Federal Institute of Technology (EPFL), 2003.

[Katara, 03] Katara, M, & Katz, S., Architectural Views of Aspects. The Int. Conf. on AOSD, ACM Press, 2003.

[Kiczales, 01] Kiczales, G., Hilsdale, E., Huguin, J., Kersten, M., Palm, J., & Griswold W.G., An Overview of AspectJ. The 15th European Conf. on Object-Oriented Programming, LNCS 2072, Budapest, Hungary, 2001.

[Kiczales, 97] Kiczales, G., Lamping, J., Mendekar, A., & Maeda, C., Aspect-Oriented Programming. The 11th European Conf. on Object-Oriented Programming, LNCS-1241, Jyväskylä, Finland, 1997.

[Kruchten, 95] Kruchten P., The 4+1 View Model of Architecture. IEEE Software, Vol. 12, no. 6, November, 1995.

[Kulkarni, 03] Kulkarni, V., & Reddy S., Separation of concerns in model-driven development, IEEE software 20(5), 2003.

[McDirmid, 03] McDirmid, S., & Hsieh, W.C.. Aspect-Oriented Programming with Jiazzi. The 2nd Int,. Conf. on Aspect-Oriented Software Development (AOSD), Boston, Massachusetts, pp. 70-79, 2003.

[Medvidovic, 00] Medvidovic, N., & Taylor, R.N., A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering,Vol. 26, No. 1, 2000.

[Meyer, 03] Meyer B., The Grand Challenge of Trusted Components. International Conference on Software Engineering (ICSE), IEEE Computer Press, Portland, Oregon, May 2003.

[Meyer, 98] Meyer, Bertrand, Object-Oriented Software Construction, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[MOF, 12] Object Management Group (OMG), "Meta-Object Facility (MOF) Specification 2.0 TR formal-06-01-01," http://www.omg.org/spec/MOF/2.0/PDF/.

[Moriconi, 97] Moriconi, M., & Riemenschneider, R. A., Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI Int., 1997.

[Navasa, 05] Navasa, A., Pérez, M.A., & Murillo, J.M., Aspect Modelling at Architecture Design. Second European Workshop on Software Architecture, LNCS 3527, Springer, Pisa, 2005.

[NET,12] .NET Technology, Microsoft Corporation, http://www.microsoft.com/net

[Parnas, 85] Parnas, D. L. 1985. Software aspects of strategic defense systems. Commun. ACM 28, 12 (Dec. 1985), 1326-1335. DOI= http://doi.acm.org/10.1145/214956.214961

[Parnas, 72] Parnas D. L., On the Criteria to be used in Decomposing Systems into Modules. Communications of the ACM, Vol 15, No.12, pp. 1053–1058, December 1972.

[Pawlak, 04] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F. , & Martelli, L, JAC: an Aspect-Based Distributed Dynamic Framework. Software Practice and Experience, V. 34, pp.1119-1148, 2004.

[Pérez, 08a] Pérez, J., Ali, N., Carsí, J.A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J.A., Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Information & Software Technology 50(9-10): 969-990, 2008.

[Pérez, 06a] Pérez, J. PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Polytechnic University of Valencia, 2006.

[Pérez, 06b] Pérez, J., Ali, N., Carsí, J.A., & Ramos, I., Designing Software Architectures with an Aspect-Oriented Architecture Description Language. The 9th Int. Symp. on Component-Based Software Engineering, LNCS 4063, Västerås, Sweden, 2006.

[Perry, 92] Perry, D., & Wolf, A., Foundations for the Study of Software Architecture. ACM Software Engineering Notes, Vol. 17, No 4, p 40-52, 1992.

[Pinto, 05] Pinto, M., Fuentes, L., Troya, J.M., A Dynamic Component and Aspect Platform, The Computer Journal Vol. 48, No. 4, pp. 401-420, 2005.

[Pinto, 03] Pinto, M., Fuentes, L, & Troya, J.M., DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. The Int.Conf. on Generative Programming and Component Engineering, LNCS 2830, Springer, 2003.

[Schmidt, 06] Schmidt D.C., Model-Driven Engineering, IEEE computer Society, 2006.

[Shaw, 94] Shaw, M.,B, Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. Workshop on Studies of Software Design, 1994.

[Simmonds, 05] Simmonds, D. et al, , An aspect oriented model driven framework. 9th IEEE Int. EDOC Enterprise Computing Conf. pp. 119-130, 2005

[Suvée, 06] Suvée, D., De Fraine B., & Vanderperren, W., A symmetric and Unified Approach Towards Combining Aspect-Oriented and Component Based Software Development. 9th Int. Symp. on Component-Based Software Engineering, LNCS 4063, Västerås, Sweden, 2006.

[Suvée, 05] Suvée, D., Vanderperren, W., Wagelaar, D., & Jonckers, V., There Are No Aspects. ENTCS, Special Issue on Software Composition, Vol. 114, pp. 153-174, 2005.

[Szyperski, 98] Szyperski C., Component software: beyond object-oriented programming. ACM Press and Addison Wesley, New York, USA (1998).

[TeachMover, 12] The TeachMover Robot, http://www.microbotzone.com/TeachMover/TeachMover/tabid/3648/Default.aspx

[Wydaeghe, 01] Wydaeghe, B., Vanderperren, W., Visual Component Composition Using Composition Pattems. Tools, Santa Barbara, California, 2001.

Appendix A: Model-Driven Development of Aspect-Oriented Software Architectures

In this appendix, we show in detail the complete architecture of the running example that has been used in the paper for illustrating the PRISMA MDD process. This is the PRISMA type model for a generic joint of a tele-operated robot that can be reused for designing different joints of the same robot or joints of different robots.

Figure 13 illustrates how the components, *Actuator* and *WrapAspSyst,* are coordinated through a connector *CnctJoint* (see Figure 12), how each component imports their functionality through aspects (see Figure 17, **Fehler! Verweisquelle konnte nicht gefunden werden.**, and **Fehler! Verweisquelle konnte nicht gefunden werden.**), and how the connector *CnctJoint* imports its behaviour through a coordination and a safety aspect, *CoordJoint* and *SMotion* respectively, to define its behaviour (see Figure 16). In addition, aspects import their corresponding interfaces, which are published through the architectural element ports that import these aspects (see Figure 14 and Figure 15). Finally, it is important to mention the need for a weaving emerges due to the fact that a joint is moved only after the connector safety constraints are satisfied assuring that a movement is safe (see Figure 16).

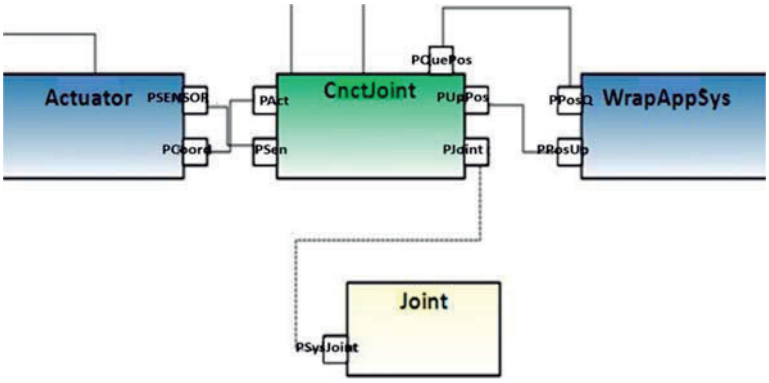


Figure 12: The joint architecture of a tele-operated robot

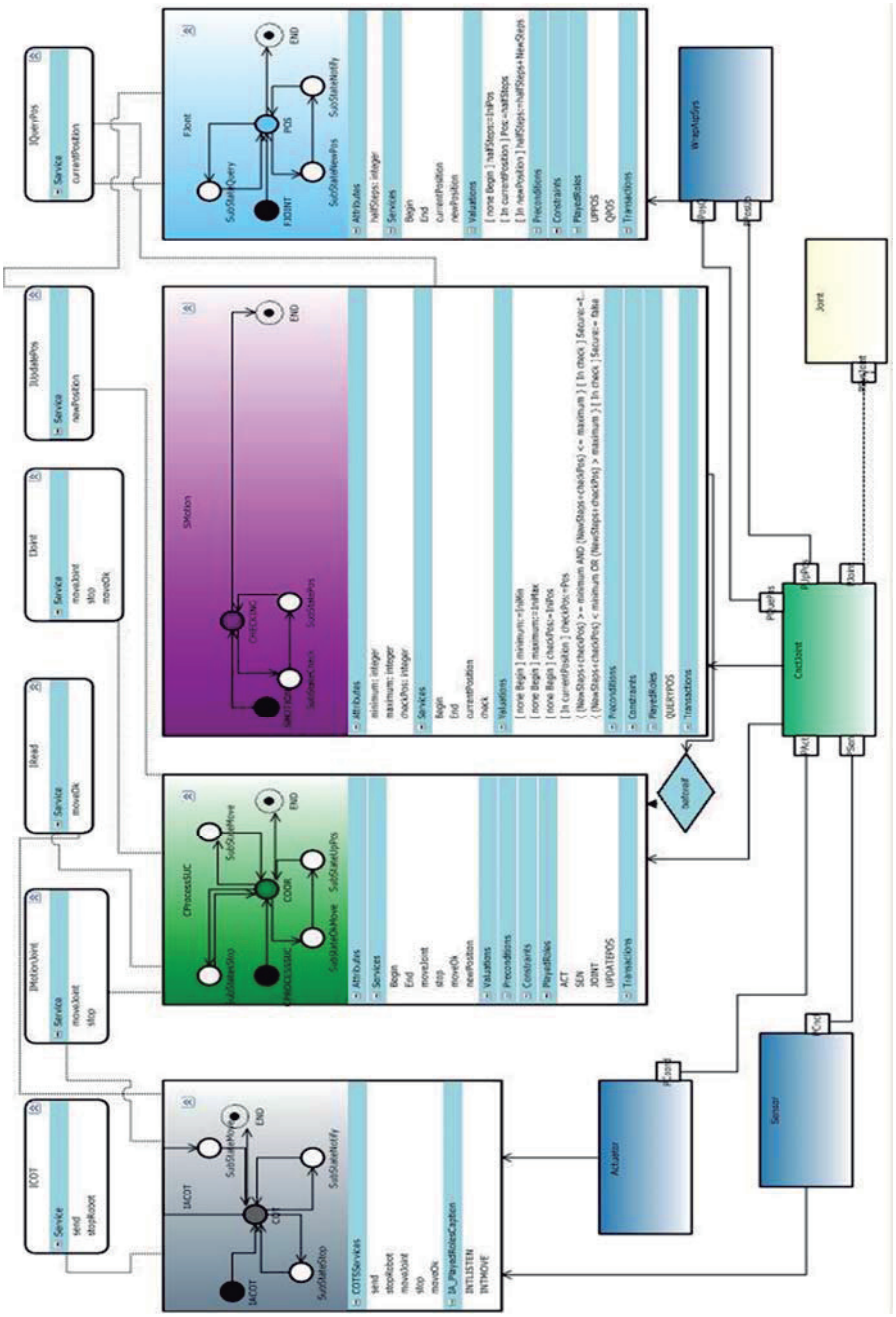


Figure 13: The PRISMA System Aspect-Oriented Architecture of a Joint for Tele-Operated Robots

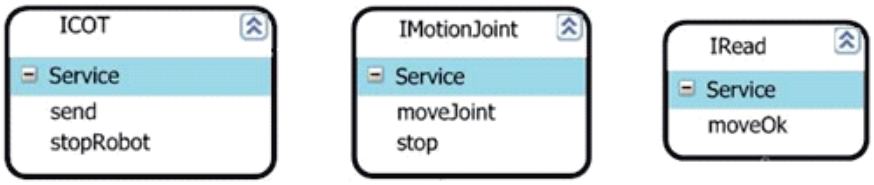


Figure 14: Interfaces ICOT, IMotionJoint and IRead

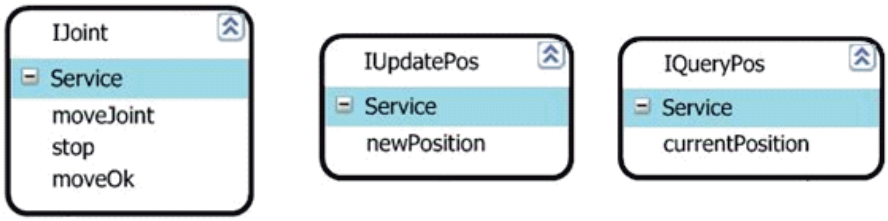


Figure 15: Interfaces IJoint, IUpdatePos and IQueryPos

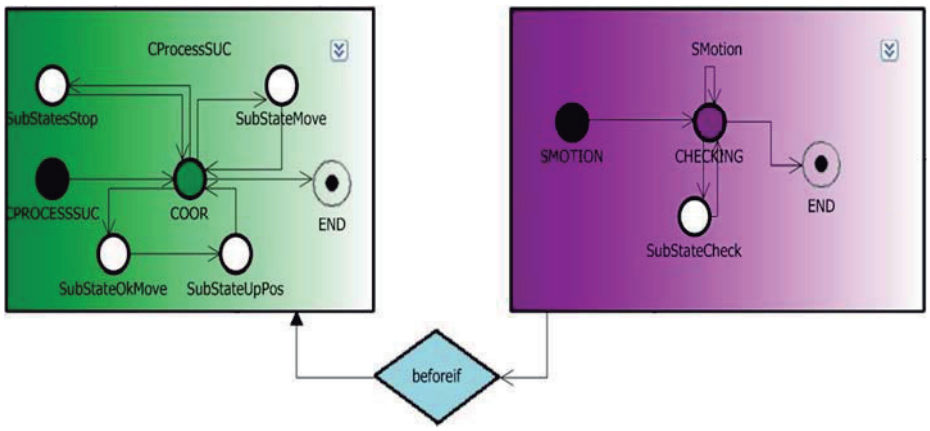


Figure 16: Weaving definition between the Aspects CProcessSuc and SMotion (shown in collapsed format)

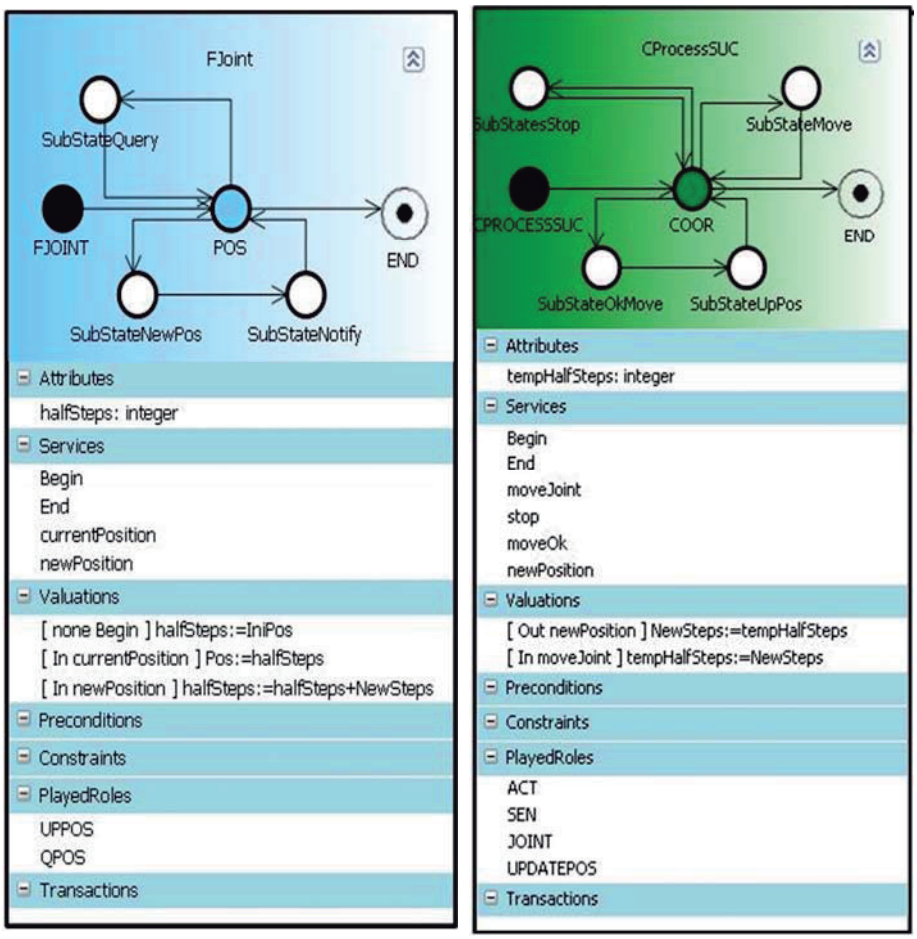


*Figure 17: Left: Functional Aspect **Fjoint** (shown in expanded format)*
*Right: Coordination Aspect **CProcessSUC** (shown in expanded format)*

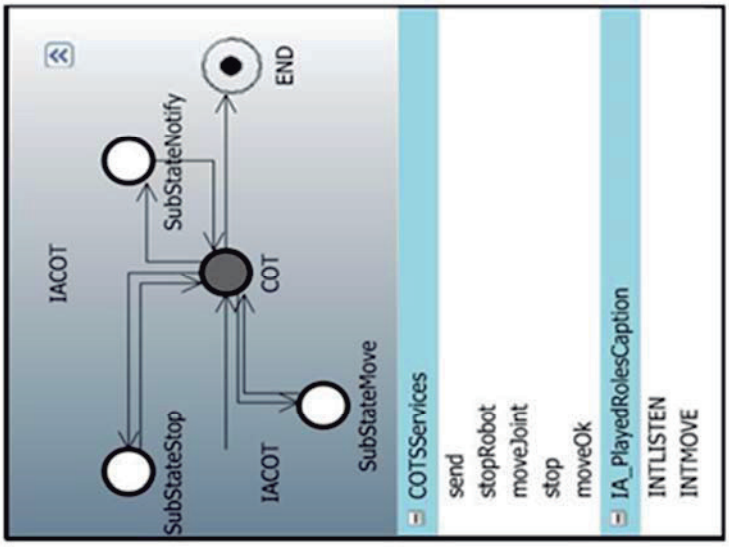


Figure 18: Functional Aspect IACOT (shown in expanded format)

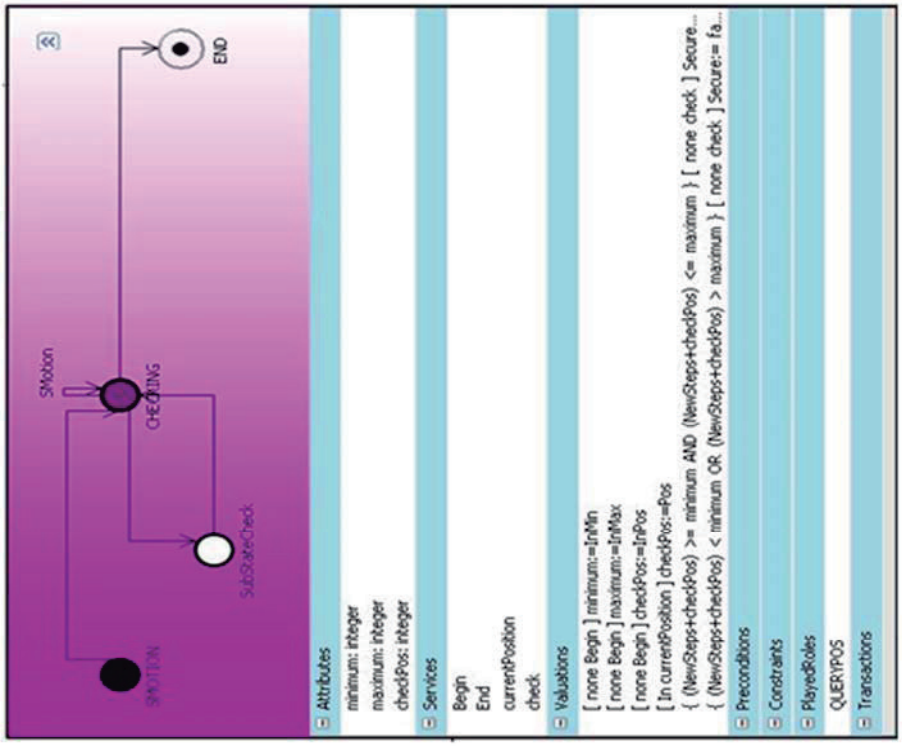


Figure 19: Safety Aspect SMotion (shown in expanded format)



ISIDRO RAMOS

PUBLICACIONES 1968-2013

VOLUMEN 4:
INTERFACES DE USUARIO (2000-2002)
GESTIÓN DE MODELOS (2002-2006)

VALENCIA, SEPTIEMBRE DE 2014

ÍNDICE GENERAL

VOLUMEN 1: MADRID, BILBAO, VALENCIA

Prólogo i

Días de hemeroteca v

… y noches de ordenador ix

Estructura del libro xi

El saber y el saber hacer en informática: una perspectiva histórica xiii

Índice general xxi

PARTE I: MADRID (1968-1974)

Prólogo de Roberto Moya Quiles 3

Isidro Ramos y el Centro de Cálculo de la Universidad de Madrid, por Ernesto García Camarero ex-director del CCUM 5

Seminarios del Centro De Cálculo De la Universidad Complutense de Madrid. 1968. 11

Gil Gayarre, M., Ramos, I., et al., Posibilidades de la Automatización en la corrección de exámenes. Revista de Automática. Año III, nº 7. 1970. 109

Fernández Flores, I. Ramos I., Traductor 70/13. Boletín CCUM. Volumen 10. 1970. 123

Moya R., Ramos, I. Aplicación de Hash-Coding al tratamiento automático de Gramáticas". Boletín CCUM. nº 17. 1971 137

Fernández Flórez, I., García Camarero, E., Ramos, I. A Basic Language Oriented to secondary School. Information Processing 71. 1972. 147

García Camarero, E., Ramos, I. Ensayo sobre la definición automática de tratamientos clínicos. Boletín CCUM. 1971. 151

Ramos, I. Lenguajes para la escritura de traductores. Revista de Automática. Enero-Marzo 1973. 163

Alfonseca, M., Ramos I. Un analizador sintáctico para un subconjunto de PL/I. Actas del Congreso Nacional de Automática. 1972. 169

Ramos, I. Un ejemplo de recursividad: HANOI. Boletín CCUM. nº 24. 1974. 205

Ramos, I., Lozano, C. Muñoz, A. Sistema de Creación y Mantenimiento de Bancos de Información. Revista de Automática, 1974 (Parte I), 1975 (Partes II y III). 211

PARTE II: BILBAO (1975-1983)

Prólogo de José Ramón Gariagoitia 255

Ramos, I. Système pour la création et gestion des banques de données. Seminaires Institut de Programmation (Université de Paris VI). 1974. 257

Moya R., Ramos I. Metaprogramación Estructurada. 3er Congreso Nacional de Informática y Automática. 1975. 263

Ramos, I., Moya, R., Martín, P. Evaluador Sintáctico del Lenguaje de Control de Comandos del Sistema Operativo EXEC-8. 3er Congreso Nacional de Informática y Automática. 1975. 289

Ramos, I. Aramberri, J. Cross-Macroassembler for Micro MCS-80 in a HP 2116-B Minicomputer. Euromicro. 1976. 301

Ramos, I. Informátique de Base: These, Antithèse, Synthèse. Convention Informatique. 1976. 309

Garitagoitia, J.R., Ramos, I. Relocating Direct Linking Loader for MCS-8 Microcomputer Series. Proc. of COMPCON 77. 1977. 313

Ramos, I. Sistema Editor / Perforador de textos. Revista de Automática e Informática. Volumen: nº 31. 1977. 319

Moya, R. Ramos, I. LPE: Un lenguaje en español para la programación estructurada. Proceso de Datos, no. 70, 1977. 325

Moya, R., Ramos I. SPL (Structured Programming Language). A language for structured programming. Proc. Of the 5th Annual III Conference. 1977. 331

Ramos, I. Aproximación prospectiva al software de base. Actas de la Convención Informática Latina. 1977. 339

Ramos, I., Aramberri, J. y Mellado, M. Modelo de realización de primitivas de sincronización para sistemas multimicroprocesador. Actas del 75 Aniversario Real Sociedad Española de Física y Química. 1978. 345

Aramberri, J., Ramos, I., Garitagoitia, J. R. Diseño y desarrollo de un sistema microprocesador. Actas I Jornadas sobre microprocesadores. 1978. 347

Moya, R., Ramos, I. SP Gramars (Context Free Gramars for Structured Programming). ACM SIGPLAN. Vol.4 Nº1. 1979. 357

Aramberri, J, Ramos, I., Mellado, M., Garitagoitia, J. R. Modelo de Sistema Microprocesador. Actas del 4º Congreso de Informática y Automática. 1979. 365

Sopeña, L., Ramos, I. Sistema de control en tiempo real de una red viaria. Actas del 4º Congreso de Informática y Automática. 1979. 375

Aguirre, J., Ramos, I. Sistema de programación estructurada con vuelta atrás. Actas del 4º Congreso de Informática y Automática, Madrid, 1979. 385

Ramos, I. Garitagoitia, J.R., Aramberri, J. Algunas soluciones al problema de interbloqueo entre procesos. 4º Congreso de Informática y Automática. 1979. 391

Guaresti, J.M., Garitagoitia, J.R. y Ramos I.A software chronological state controller. Actas del 4º Congreso de Informática y Automática. 1979. 403

Ramos, I. Calculadora de mesa. Un proyecto de clase sobre teoría de Lenguajes y Autómatas. Revista de Informática y Automática. 1979. 415

Zuloaga, J.L., Santamaria, R., Emaldi, R., Ramos, I. Sistema de Control lógico programable sobre microordenador. Revista de Informática y Automática. 1979. 421

Garitagoitia, J.R., Martinez,C., Ramos, I. MIKBUS: un bus para placas de microcomputadores cooperantes. Actas del II Simposio Español de Informática Repartida. 1982. 429

Garitagoitia, J.R., Ramos, I. Algoritmo de prevención para un sistema de gestión de ficheros. Actas de la Convención Informática Latina. 1983. 439

PARTE III: UNIVERSITAT DE VALÈNCIA - UNIVERSIDAD DE CASTILLA-LA MANCHA (1978-1987)

Prólogo de Francisco José Quiles Flor 455

Ramos, I. Semántica para SPL. Questió. Vol. 4. nº 1.1980. 459

Ramos, I. Un lenguaje de Programación con tipos Abstractos interpretado en PROLOG. Otra forma de compilar. Revista de Informática y Automática. 1984. 467

Ramos, I. Especificación de Pantallas. Revista de Informática y Automática, 1984. 471

Marín, J, Ramos, I. Metodología para la adquisición de material informático. Actas de las Jornadas de Informática en la Administración Local. 1984. 475

Ramos, I. The impact of new tecnologies on University organization: the example of Castilla-La Mancha. C.R.E. INFORMATION. Volumen: nº 70. 1985. 491

VOLUMEN 2: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

PARTE IV: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

Prólogo de María Alpuente 527

Celma, M., Moreno, L., Pastor, O., Ramos, I. Implementación de un sistema de ges-

tión de bases de datos deductivas. Revista de Informática y Automática, Vol. XX, Nº 3. 1987. 529

Alpuente, M., Botti, V., Galiano, I., Sanchis, E., Vila, F., Ramos, I. Entorno de programación LALI implementado en PROLOG. Revista de Informática y Automática . Vol. XX, Nº 4. 1987. 537

González del Río, A., Alpuente, M., Casamayor, J.C., Pastor, M.A., Ramírez, M.J., Ramos, I. Protolog : A conceptual Schema Facility for Automated Prototype Generation. 7th. IASTED International Symposium. 1989. 543

Ramos I., Canós, J.H., Forradellas, R., Oliver, J. A conceptual schema specification system for rapid prototyping. 8th IASTED International Symposium. Austria. Feb. 1990. 547

Ramos, I. Logics and OO-Data Bases: a declarative approach. Proc. International Conference on Database and Expert Systems Applications (DEXA 90), 1990. 551

Ramos, I.; Pastor, O.; Casado, V. OO and Active Formal Information System Specification. Proc. International Conference on Database and Expert System Applications (DEXA 91) Vol. II. 1991. 557

Pastor, O. Ramos, I. Canós, J.H. Object Oriented and Relational Specification of Information Systems. Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS. 1991. 563

Ramos, I., Cuevas, J., Devesa, J., Pastor, O., Canós, J.H. On the use of Algebras as Semantic domains of Object Societies. Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS. 1992. 579

Alpuente, M., Ramírez, M. J. An Equational Constraint Logic approach to Conceptual Modelling. In Ramos, I. and Tjoa, A. (eds), Proc. of the International Conference on Database and Expert Systems Applications (DEXA 92), 1992. 589

Ramos, I., Pastor, O., Cuevas J., Devesa , J. Objects as Observable Processes. Proc. of the 4th International Workshop of the Deductive Approach for DB and IS.1993. 595

Pastor, O., Ramos, I., Canós, J.H. From Analysis to Design in an OO and Deductive Environment. Proc of the International Conference on Database and Expert System Applications (DEXA´93). 1993. 611

Toval, A., Ramos, I. , Pastor, O. Prototyping Object Oriented Specifications in an Algebraic Environment. Proc. Of the International Conference on Database and Expert System Applications (DEXA'94), 1994. 617

Devesa, J., Cuevas, J., Ramos, I. Implementation of a Term Rewriting System for solving process queries in an object-oriented environment". Proc. Of the GULP-PRODE 94. 1994. 627

Canós, J. H., Penadés, M.C., Ramos, I., Pastor, O. A Knowledge-Base Architecture for Object Societies. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 643

Pastor, O., Ramos, I., Canós J.H. OASIS v2: A Class Definition Language. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 649

Ramos, I., Toval, A., García-Molina, J., Flores, M., González, M., Fernández, J. L., Nicolás, J., Pastor, O., Canós, J. H., Pelechano, V., Penadés, M.C., Bonet, B. Entorno General de Produccion automática de prototipos de software orientado a objetos: OOAP (Object-Oriented Assistant Prototyper). Actas de las I Jornadas de Informática. Asociación Española de Informática y Automática (AEIA). 1995. 659

Pastor O., Pelechano V., Bonet B., Ramos I. An object Oriented Approach for making Automated Prototyping Feasible. Proc. of the International Conference on Database and Expert System Applications (DEXA 96). 1996. 673

Carsí, J.A., Ramos, I., Penadés, M.C., Pelechano, V., Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic. Actas de las I Jornadas de Trabajo en Ingeniería del Software, 1996. 683

Penadés, M.C., Ramos, I., Canós, J.H., Tolsá, J., Ferrer, J. Análisis, diseño e implementación de un Entorno de Prototipación Automática. Actas de las I Jornadas de Trabajo en Ingeniería del Software,1996. 695

Sánchez, P. Letelier, P., Ramos, I. Representación de Objetos OASIS como Teorias y Traducción a un Modelo de Concurrencia. Computación y Sistemas, Vol. I, Nº. 2. 1997. 707

Lozano, M.D., Ramos, I. Evolución y perspectivas en el desarrollo de software: nuevas tendencias orientadas a objetos. Ensayos: revista de la Escuela Universitaria de Magisterio de Albacete, Nº. 12. 1997. 719

Sánchez, P., Letelier, P., Ramos, I. Constructs for Prototyping Information Systems using Object Petri Nets. Proc. of the IEEE International Conference on System, Man and Cybernetics.1997. 733

Lozano, M.D., Ramos, I., Cuenca, P. Application of an Object-Oriented Methodology for Automatic Software Production : OO-Method. Proc. Of the IEEE PACRIM 97. 1997. 739

Letelier, P., Sánchez, P., Ramos, I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitios. Proc. of the Argentine Symposium on Object Orientation. 1997. 743

Letelier, P. Sánchez, P., Ramos. I. Animation of Systems Specifications Using Concurrent Logic Programming. Proc. of the Symposium in Logical Approaches to Agent and Modelling and Design (ESSLLI´97). 1997. 751

Carsí, J.A., Canós, J.H., Ramos, I. Un intérprete de Transaction Frame Logic. Actas de las II Jornadas de Trabajo en Ingeniería del Software. 1997. 759

Tolsá, J.M., Ferrer, J., Penadés, M.C., Canós, J.H., Ramos, I. Meta-KAOS. Un metaentorno de prototipación automática. Actas de las II Jornadas de Trabajo en

Ingeniería del Software. 1997. 773

Ramos, I.,Canós, J.H., Pastor,O.+ investigadores proyecto MENHIR. MENHIR: Modelos, Entornos y Nuevas Herramientas para la Ingeniería de Requisitos. Actas de las III Jornadas sobre Tecnología de Objetos. 1997. 787

Canós, J.H., Lozano, M.D., Ramos, I. KAOS: An Object-Oriented Software Tool for the Objects Definition, Updating, Querying and Programming in an Object-Oriented Environment. Proc. of the IEEE Canadian Conference on Electrical & Computer Engineering.1998. 799

Letelier, P., Sánchez, P., Ramos, I. Especificaciones de proceso para objetos y su representación en Lógica Dinámica. Actas de las III Jornadas de Ingeniería de Software. 1998. 803

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto. Actas de las III Jornadas de Ingeniería del Software. 1998. 813

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a las pautas de representación de OASIS en un formalismo gráfico. Actas de las III Jornadas de Ingeniería del Software. 1998. 827

Carsí, J. A., Camilleri, S., Canós, J.H., Ramos, I. Propuesta de Interfaz Gráfica de Usuario homogénea para el diseño y la explotación de Sistemas de Información. Actas de las III Jornadas de Ingeniería del Software. 1998. 839

Letelier, P., Sánchez, P., Ramos, I. Un ambiente para especificación incremental y validación de modelos conceptuales. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99).1999. 853

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99). 1999. 867

Letelier, P., Sánchez, P., Ramos, I. Prototyping a requirements specification through an automatically generated concurrent logic program. Proc. of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99). 1999. 879

Pastor,O., Canós,J.H., Ramos,I. From CASE to CARE (Computer-Aided Requirements Engineering). Proc. of the International Conference on Conceptual Modeling (ER'99). 1999. 895

Lozano, M., Ramos, I. Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. Workshop on Integrating Human Factors into Use Cases and OO Methods at ECOOP'99. 1999. 911

Letelier, P., Sánchez, P., Ramos, I. Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales". Actas del VI Congreso Internacional de Investigación en Ciencias Computacionales (CIICC'99). 1999. 917

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. Modelado conceptual con un lenguaje

formal y orientado a objeto. Actas de la XXV Conferencia Latinoamericana de Informática (CLEI'99). 1999. 929

Letelier, P. Sánchez, P., Ramos, I. Animation of Conceptual Models using two Concurrent Environments: An overview. Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers. 1999. 943

Parets, J., Carsí, J., Canós, J.H., Anaya, A., Hurtado, M.V., Penadés, M.C., Paderewski, P., Ramos, I., Rodriguez, M.J., La evolución de modelos en el desarrollo de software: los enfoques MEDES y OASIS. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos. 1999. 951

Sánchez, P., Letelier, P., Ramos, I. Una Interpretación de la Herencia de OASIS en Lógica Dinámica. Computación y Sistemas, vol. 4, núm.4. 2000. 963

Sánchez, P., Letelier, P., Ramos, I. Animating formal specifications with Inheritance in a DL-based framework. Requirements Engineering Journal, vol.4, no.4. 2000. 973

Anaya, R., Ramos, I. Un lenguaje formal para especificación y refinamiento de Objetos de Negocio. Ingeniería del software y bases de datos: tendencias actuales. Universidad de Castilla-La Mancha, 2000. 995

Anaya, R., Ramos, I. AR2CA: Una Herramienta para la Construcción de componentes reutilizables a través de niveles de refinamiento. Actas de 3as Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software (IDEAS2000). 2000. 1011

Carsí, J. A., Ramos, I., Canós, J.H. Reflection in the OASIS object model to solve Schema Evolution Problems. Proc. of the International Symposium on the Principles of Software Evolution (ISPSE2000). 2000. 1025

Sánchez, P., Letelier, P., Ramos, I. Validation of Conceptual Models by Animation in an Scenario based approach. Workshop 'Scenario-based round-trip engineering' at OOPSLA 2000. 2000. 1031

Giret, A., Ramos, I.Full software agents as OASIS 3.0 objects. Proc. of the 2nd Constraint) Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2001. 2001. 1037

Letelier, P., Ramos, I. Trazabilidad de Requisitos y Calidad del Software: Una Aproximación basada en UML. En Piattini y García (Eds.), Calidad en el Desarrollo y Mantenimiento del Software. Ra-MA, 2002. 1053

Pérez, J., Carsí, J. A., Ramos, I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR) at ICSM'02. 2002. 1075

VOLUMEN 3: ARQUITECTURAS SOFTWARE (2002-2010)

PARTE V: ARQUITECTURAS SOFTWARE (2002-2010)

Prólogo de Patricio Letelier 1109

Ferrer, J., Lorenzo, M. A., Ramos, I.,Carsí, J. A., Pérez, J. Modeling Dynamic Aspects in Architectures and Multiagent Systems. Proc. of the Constraint Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2002. 2002. 1111

Pérez, J., Ramos, I., Lorenzo, M. A., Letelier, P., Jaén, J. PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos. Actas de las VII Jornadas de Ingeniería del Software y Bases de Datos. 2002. 1125

Ferrer, J., Lorenzo, M. A., Ramos, I., Carsí, J. A. Dinamismo en arquitecturas y sistemas multiagente. Actas de las V Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2002). 2002. 1137

Jaén, J., Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures. Proc. of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03). 2003. 1147

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. Proc. of the 3rd IEEE International Conference on Quality Software (QSIC 2003). 2003. 1153

Pérez, J., Ali, N., Ramos, I., Carsí, J. A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes. Workshop Desarrollo de Software Orientado a Aspectos (DSOA). Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1161

Pérez, J., Ali, N., Ramos, I., Pastor, J. A., Sánchez, P., Álvarez, B. Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1167

Lorenzo, M. A., Peñarrubia, A., Carsí, J.A., Ramos, I. Modelando con UML Arquitecturas Software Reflexivas desde la Aproximación Orientada a Aspectos: Un Caso de Estudio. Actas de las VI Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2003). 2003. 1177

Navarro, E., Ramos, I. Requirements and Architecture: a marriage for Quality Assurance. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1189

Silva, J., Ali, N., Carsí, J. A.,Ramos, I. El aspecto de distribución de PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1199

Ali, N., Silva, J., Jaén, J., Ramos, I., Carsí, J. A., Pérez, J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proc. of the 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2003). 2003. 1209

Pérez J., Ali, N. Carsí, J. A., Ramos, I. Navarro, E. Designing Software Architectures with an Aspect-Oriented Language. ADVICE: Journal of Aspect Orientation, Vol. 1, No. 1, 2004. 1217

Navarro, E., Letelier, P., Ramos, I. Goals and Quality Characteristics: Separating

Concerns. Proc. of Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004. 2004. 1239

Navarro, E., Letelier, P., Ramos, I. UML Visualization for an aspect and goal oriented approach. Proc. of the 5th Aspect-Oriented Modeling Workshop (AOM'04) at UML 2004 Conference. 2004. 1247

Navarro, E., Ramos, I., Pérez, J. Goals Model Driving Software Architecture. 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004). 2004. 1255

Ali, N., Pérez, J., Ramos, I. High Level Specification of Distributed and Mobile Information Systems. Proc. of the Second International Symposium on Innovation in Information & Communication Technology (ISSICT2004). 2004. 1263

Ali, N., Carsí, J. A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2004. 1277

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004). 2004. 1285

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Orientación a aspectos y Orientación a objetivos: una propuesta para su integración. Taller de Desarrollo de Software Orientado a Aspectos, junto a JISBD'2004. 2004. 1293

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Dynamic Evolution in Aspect-Oriented Architectural Models. European Workshop on Software Architecture. 2005. 1299

Pérez, J., Ali, N., Costa, C., Carsí, J. A., Ramos, I. Executing Aspect-Oriented Component-Based Software Architectures on . NET Technology. Proc. of the 3rd International Conference on .NET Technologies. 2005. 1317

Ali, N., Ramos, I., Carsí, J. A. A Conceptual Model for Distributed Aspect-Oriented Software Architectures. Proc. of the International Conference on Information Technology Coding and Computing, ITCC 2005. 2005. 1329

Limón, R., Torres, J., Ramos, I. Separación de "Crosscutting Concerns" en el Desarrollo Orientado a Aspectos a Nivel de Diseño Arquitectónico. Actas del 6º Simposium Iberoamericano de Computación e Informática (SICI'05). 2005. 1335

Navarro, E., Letelier, P., Ramos, I. Integrating Expressiveness of Modern Requirements Modeling Approaches. Proc. of the 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005). 2005. 1345

Costa, C., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1355

Cabedo, R., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Implementación Orientada a Aspectos en C# de un sistema robótico tele-operado. III Taller de Desarrollo de Software Orientado a Aspectos de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2005. 1363

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1371

Navarro, E., Letelier, P., Mocholí, J. A.,Ramos, I. A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems, XLVI(5), Special Issue on Systems Analysis and Design. 2006. 1379

Cabello, M. E., Costa, C., Ramos, I.,Carsí, J. A. Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System. Journal of the WSEAS Transactions on Information Science and Applications, Issue 10, Volume 3. 2006. 1391

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. Proc. of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006). 2006. 1399

Pérez, J., Navarro, E., Letelier, P., Ramos, I. A Modelling Proposal for Aspect-Oriented Software Architectures. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). 2006. 1415

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. A Goal-Oriented Approach for Safety Requirements Specification. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06). 2006. 1425

Limón, R., Ramos, I., Torres, J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development. Proc. of the International Conference the Computational Science and Its Applications (ICCSA 2006) . 2006. 1433

Limón, R., Ramos, I. Using Styles to Improve the Architectural Views Design. Proc. Of the International Conference on Software Engineering Advances (ICSEA 2006). 2006. 1443

Cabello, M. E., Costa, C., Ramos, I., Carsí, J. A. Generic Architecture of an Expert System for Diagnosis. Proc. of the 10th WSEAS International Conference on Computers. 2006. 1449

Cabello, M. E., Ramos, I., Costa, C. Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico. Actas del XIX Congreso Nacional y V Congreso Internacional de Informática y Computación del ANIEI-CIICC´07. 2006. 1455

Ali, N., Millan, C., Ramos, C. Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model. Proc. of the 8th International Sym-

posium on Distributed Objects and Applications (DOA). 2006. 1463

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes. IEEE Latin America Transactions, Special Edition JISBD'06, vol. 5, issue 4. 2007. 1481

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Mobile Ambients in Aspect-Oriented Software Architectures. Proc. of the IFIP Working Conference on Software Engineering Techniques - SET 2006. 2006. 1489

Navarro, E., Letelier, P., Reolid, D., Ramos, I. Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques. Proc. of the 15th International Conference on Information System Development. 2006. 1501

Navarro, E., Letelier, P., Ramos, I. Requirements and Scenarios: running Aspect Oriented Software Architectures. Proc. of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). 2007. 1515

Limón, R., Ramos, I. Analyzing Styles of the Modular Software Architecture View. Proc. of the First European Conference on Software Architecture (ECSA 2007). 2007. 1521

Limón, R., Ramos, I. Relating Software Architecture Views by Using MDA. Proc. of the ICCSA 2007 International Conference on Computational Science and its Applications. 2007. 1525

Limón, R., Ramos, I., Morales, M. Zárate, J. Analyzing and Designing Software Architecture Views driven by their Relationships. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 1537

Pérez, J., Costa, C., Carsí, J. A.,Ramos, I. Verificación de Modelos Arquitectónicos Orientados a Aspectos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos. 2007. 1547

Costa, C., Ali, N., Pérez, J., Carsí, J. A., Ramos, I. Dynamic Reconfiguration of Software Architectures Through Aspects. Proc. of the First European Conference on Software Architecture (ECSA'07). 2007. 1557

Limón, R., Cabello, M. E., Ramos, I. Establishing Relations among Software Architecture Views through MDA for SPL. Proc. of the 14th International Congress on Computer Science Research (CIICC'07). 2007. 1563

Pérez, J., Ali, N., Carsí, J. A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J. A. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Journal of Information and Software Technology 50 (2008) . 2008. 1575

Ali, N., Solís, C., Ramos, I. Comparing architecture description languages for mobile software systems. Proc. of the First International Workshop on Software Architectures and Mobility at ICSE'08. 2008. 1597

Navarro, E., Gómez, A., Letelier, P., Ramos, I. MORPHEUS: a supporting tool for MDD. Proc. of the18th International Conference on Information Systems

Development (ISD2009). 2009. 1603

Ali, N., Ramos, I., Solís, C. Ambient-PRISMA: Ambients in Mobile Aspect-Oriented Software Architecture. Journal of Systems and Software 83. 2010. 1615

Pérez, J., Ramos, I., Carsí, J.A., Costa, C.: Model-Driven Development of Aspect-Oriented Software Architectures. Journal of Universal Computer Science, vol. 19, no. 10. 2013. 1637

VOLUMEN 4: INTERFACES DE USUARIO (2000-2002) - GESTIÓN DE MODELOS (2002-2006)

PARTE VI: INTERFACES DE USUARIO - IDEAS (2000-2002)

Prólogo de María Dolores Lozano 1701

Lozano, M.D., Ramos, I. An Object Oriented Approach for the Specification of User Interfaces. Proceedings of PACRIM'99 Conference. 2000. 1703

Lozano, M. D., Ramos, I., González, P. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development". Proc. of the IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000). 2000. 1707

Lozano, M. D, González, P., Ramos, I. Models and Ontologies for the Specification of User Interfaces. Proc. of the Workshop on Model Engineering at ECOOP'00, 2000. 1717

Lozano, M. D., Ramos, I. Aproximación Basada en Modelos en la especificación de Interfaces de Usuario. Actas de las I Jornadas de Interacción Persona-Ordenador (INTERACCIÓN-20), 2000. 1725

Lozano, M. D., González, P., Ramos, I. User Interface Generation. Current Trends. INFORMATIK /INFORMATIQUE. Journal of the Swiss informatics societies. Vol. 2. 2001. 1733

Lozano, M. D., González, P., Ramos, I. IDEAS: An Object-Oriented Environment for developing User Interfaces". Proc. of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01). 2001. 1739

Lozano, M. D., González, P., Ramos, I., Montero, F., Molina, J. P. Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.16.2002. 1749

Lozano, M., González, P., Ramos, I. A Graphical User Interface Development Tool. 16th British Computer Society Conference on Human-Computer Interaction (BHCI'02). 1759

Lozano, M., González, P., Montero, F. Molina, J. P., Ramos, I. Integrating Usability within the User Interface Development Process of Web Applications. Proc. of the 2nd International Workshop on Web Oriented Software Technology at ECOOP'02. 2002. 1763

Montero, F. Lozano, M. D., González, P., Ramos, I. Designing Web Sites by Using Design Patterns. Proc. of SugarLoafPLoP 2002. 2002. 1779

PARTE VII: GESTIÓN DE MODELOS

Prólogo de José Ángel Carsí 1805

Carsí J.A, Ramos I., Silva J., Pérez J., Anaya V. Un Generador Automático de Planes de Migración de Datos. I+D Computación, Vol. 1, No. 1. 2002. 1809

Silva J.F., Carsí J.A., Ramos I. Comparación automática de esquemas conceptuales orientados a objeto, Revista Ingeniería Informática - Revista electrónica del Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción - Chile. 2002. 1821

Pérez J., Ramos I., Anaya V., Cubel J.M, Domínguez F., Boronat A., Carsí J.A. Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. Proc. of the Workshop on Software Evolution Through Transformations (SET'02) at ICGT2002. 2002. 1833

Silva, J., Carsí, J. A., Ramos, I. An algorithm to compare OO-Conceptual Schemas. Proc. of the International Conference on Software Maintenance (ICSM'2002). 2002. 1847

Pérez J., Carsí J.A, Ramos I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR'2002) at the ICSM'02. 2002. 1857

Pérez, J., Carsí, J. A., Ramos, I. ADML: A Language for Automatic Generation of Migration Plans. Proc. of the First Eurasian Conference on Advances in Information and Communication Technology. 2002. 1869

Boronat, A., Carsí, J. A., Ramos, I. Una plataforma semántica para la gestión de modelos. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos. 2003. 1879

Boronat, A., Pérez, J., Carsí, J. A.,Ramos, I. Two experiencies in software dynamics. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. 2004. 1889

Boronat, A., Carsí, J. A., Ramos, I. An Algebraic Baseline for Automatic Transformations in MDA. Proc. of the Workshop on Software Evolution Through Transformations at ICGT2004. 2004. 1919

Boronat, A., Pedrós, J., Carsí, J. A., Ramos, I. Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004). 2004. 1935

Boronat, A., Carsí, J. A., Ramos, I., Pedrós, J. An Approach for Cross-Model Semantic Transformation on the . NET Framework. Proc. of the .NET Technologies'2005 Conference. 2005. 1947

Boronat, A., Boronat, A., Carsí, J. A.,Ramos, I. Una Arquitectura para la Definición

de Metáforas Gráficas para Metamodelos. Revista IEEE América Latina, Special Edition - JISBD'2004. 2005. 1957

Silva, J., Carsí, J. A., Ramos, I. Análisis teórico experimental de criterios de comparación de esquemas conceptuales orientados a objeto. Computación y Sistemas, Vol. IX no.1. 2005. 1967

Boronat, A., Carsí, J. A., Ramos, I. Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. Proc. of the 9th European Conference on Software Maintenance and Reengineering. 2005. 1983

Boronat, A., Carsí, J. A., Ramos, I. Automatic Support for Traceability in a Generic Model Management Framework. Proc. of the European Conference on Model Driven Architecture - Foundations and Applications. 2005. 1995

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). 2005. 2011

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos - JISBD'05. 2005. 2019

Queralt, P., Hoyos, L., Boronat, A., Carsí, J. A., Ramos, I. Un motor de transformación de modelos con soporte para el lenguaje QVT Relations. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). 2006. 2025

Boronat, A., Carsí, J. A., Ramos, I. Exogenous Model Merging by means of Model Management Operators. Proc. of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change at ICGT 2006. 2006. 2035

Boronat, A., Carsí, J. A., Ramos, I.,Letelier, P. Formal Model Merging Applied to Class Diagram Integration. Proc. of the International ERCIM Workshop on Software Evolution (2006). 2006. 2055

Boronat, A., Carsí, J. A., Ramos, I. Algebraic Specification of a Model Transformation Engine. Proc. of the Fundamental Approaches to Software Engineering Workshop (FASE'06) at ETAPS'06. 2006. 2061

Boronat, A., Ramos, I., Carsí, J. A. Definition of OCL 2. 0 Operational Semantics by means of a Parameterized Algebraic Specification. Proc. of the 1st International Workshop on Algebraic Foundations for OCL and Applications. 2006. 2077

Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J. A. An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework. Proc. of the Second European Conference on Model Driven Architecture. 2006. 2093

Gómez, A., Boronat, A., Hoyos, L., Carsí, J. A., Ramos, I. Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos.

Actas de las XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006). 2006. 2109

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I., Täubner, C., Eckstein, S. Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2119

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2129

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2135

Lloret, E., Escobar, S., Palomar, M., Ramos, I. Incremental and adaptive Software Systems Development of Natural Language Applications. Proc. of the 22nd International Conference on Information Systems Development (ISD 2013). 2013. 2143

VOLUMEN 5: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

PARTE VIII: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

Prólogo de Emilio Insfrán y Silvia Abrahão 2177

Carsí, J. A., Insfrán, E., Abrahao, S., Genero, M., Piattini, M., Ramos, I. ¿Cómo utilizar experimentos controlados en la selección de transformaciones QVT para obtener diagramas de clases UML más fáciles de entender? Proc. of the 8th Argentine Symposium on Software Engineering (ASSE 2007). 2007. 2179

Limón, R. Ramos, I. Aragón, R. Transformación de Vistas Arquitectónicas Orientada por Modelos. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 2189

Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Marco de Trabajo basado en MDA para la Medición Genérica del Software. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2199

Mora, B., García, F., Francisco Ruíz, Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Software Measurement by Using QVT Transformations in an MDA Context (ICEIS 2008). Proc. of the Tenth International Conference on Enterprise Information Systems, Volume DISI. 2008. 2209

Abrahao, S., Genero, M., Insfrán, E., Carsí, J. A., Ramos, I., Piattini, M. Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In Rech, J. and Bunse, C. (Eds.), Model-Driven Software Development: Integrating Quality Assurance. 2008. 2217

Insfrán, E., Carsí, J. A., Abrahao, S., Genero, M., Piattini, M., Ramos, I. Towards Quality-Driven Model Transformations: A Replication Study. Proc. of the International Workshop on Empirical Studies of Model-Driven Engineering (ESMDE 2008) at MODELS 2008. 2008. 2247

Cabello, M. E., Ramos, I. A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines. Proc. of the International Conference of Health Informatics- HEALTHINF´08. 2008. 2257

Cabello, M. E., Ramos, I. Variability Management in Product Lines for Decision Support Systems. Proc. of the 10th International Conference on Enterprise Information Systems (ICEIS´08). 2008. 2267

Cabello, M. E., Ramos, I. The Baseline: the milestone of Software Product Lines for Expert Systems automatic development. Proc. of the 9th Mexican International Conference on Computer Science. 2008. 2277

Gómez, M., Gómez, A., Cabello, M. E.,Ramos, I. BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. Actas del VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). 2009. 2287

Cabello, M. E., Ramos, I., Gómez, A., Limón, R. Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for Expert Systems development. Proc. of the 1st Asia Conference on Intelligent Information and Database Systems (ACIIDS´09). 2009. 2297

Cabello, M. E., Ramos, I. Expert Systems development through Software Product Lines techniques. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2303

Cabello, M. E., Ramos, I. Model-Driven Development of Decision Support Systems: tackling the variability problem. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2315

Gómez, A., Ramos, I. Cardinality-based feature models and Model-Driven Engineering: fitting them together. Proc. of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. 2010. 2327

Gómez, A., Cabello, M. E., Ramos, I. BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. Proc. of the 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010) at SPLC 2010. 2010. 2335

Gómez, A., Ramos, I. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. Proc. of the 19th International Conference on Information Systems Development (ISD'10) 2010. 2343

Abrahao, S., Javier González, Insfrán, E.,Ramos, I. Software Evolution in Model-Driven Product Line Engineering. ERCIM News 88, Special Theme: Evolving Software. 2012. 2355

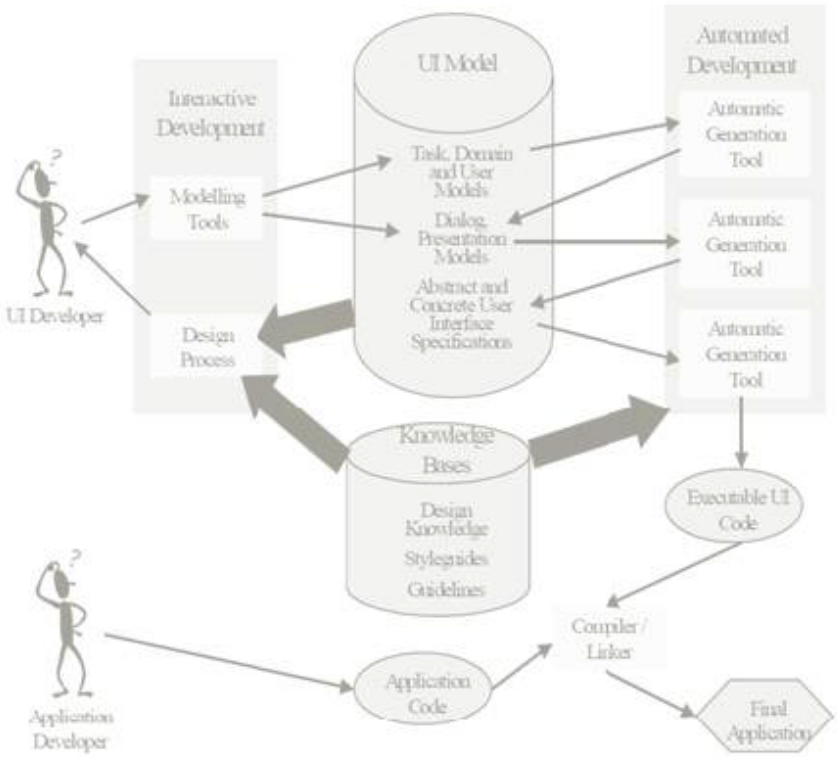
Cabello, M. E., Ramos, I., Gutiérrez, J. R., Gómez, A., Limón, R. SPL Variability Management, Cardinality and Types: an MDA Approach. International Journal of Intelligent Information and Database Systems (IJIIDS), Vol. 6, No. 2. 2012. 2357

Insfrán, E., Abrahao, S., Javier González, McGregor, J. D., Ramos, I. A Multimodeling Approach for Quality-Driven Architecture Derivation. Proc. of the 21st International Conference on Information Systems Development (ISD2012). 2012. 2383

PARTE IX: LOS AMIGOS

Los amigos 2397

Apéndice 2441





PARTE VI

INTERFACES DE USUARIO: IDEAS (2000-2002)

PRÓLOGO DE MARÍA DOLORES LOZANO

INTERFACES DE USUARIO: IDEAS (2000-2002)

PARTE VI

«Todo el mundo deja una huella en una persona y sin esa huella no somos nosotros mismos»

Recopilando la colección de artículos realizados como fruto de la etapa de investigación encaminada a mi Tesis Doctoral, dirigida por Isidro en 2001, me vienen a la cabeza mil recuerdos, todos buenos, todos gratos. Fueron unos años intensos, el primer artículo data de 1997. Y a partir de ahí arranca una carrera en el ámbito universitario en la que no me puedo sentir más «como pez en el agua». En esos momentos, con las dudas propias de la juventud, podría haber cogido cualquier camino de los que se me abrían por delante. Pero hubo una persona, Isidro, que me aportó luz en uno de esos caminos. Gracias a él, y junto a él, todos y cada uno de los miembros de su grupo de investigación, vi claro que mi camino era ese, establecer mi profesión en la universidad. No puedo estar más contenta de haber acertado en esta elección. Ahora tengo ocasión de agradecer a Isidro la gran influencia que su persona y su equipo tuvieron en esa decisión tan fundamental en la vida. Dicen que «la palabra convence y el ejemplo, arrasa». Y resume bien la influencia recibida de mis años de trabajo más estrechos con Isidro. Destacaría la inteligencia brillante, capaz de hacer simple lo complejo; la capacidad de trabajo y esfuerzo incansable; la habilidad de adelantarse al futuro, de apostar por ideas que veinte años después se han puesto de moda. Y todo esto y mucho más son esas cosas de una persona que no hace falta oírlas, porque se ven. Como reza la frase que abre este prólogo, inconscientemente todos, de alguna manera, dejamos huella en las personas con las que nos encontramos en la vida. Y esa

huella nos marca de algún modo en algún aspecto. No podría precisar en qué medida la huella recibida por Isidro me ha quedado marcada, pero sé con seguridad que ahí está y forma parte de mi historia personal.

María Dolores Lozano Pérez
Albacete, 13 de Julio de 2014

An Object Oriented Approach for the Specification of User Interfaces

María D. Lozano
Univ. of Castilla-La Mancha

Isidro Ramos
Technical Univ. of Valencia

ABSTRACT:

The object oriented paradigm is considered as one of the best approaches for systems development. There exists many models, techniques an tools developed following this paradigm. Nevertheless, most of the object-oriented techniques used in the development of systems does not take into account Human-Computer interaction issues. This is a very important aspect that must be tackled in order to correctly fulfil the user needs and requirements. The main purpose of this paper is to present an object oriented approach for the integration of a user model in an object oriented software production environment.

INTRODUCTION:

The main goal of this paper is to present a methodology for specifying user interfaces within an automatic software production environment**.** This environment is supported by an object oriented model, OASIS (Open and Active Specification of Information Systems) [Pas95 y Pat98] and a methodology associated to the model, OO-Method [Pas97], [Loz97], covering the typical phases of Analysis, Design and Implementation within the principles of the Object Oriented Paradigm.

However there exists a small gap within this environment; it is the case of interactive systems which implies a great level of user interaction and thus require to give great importance to the user interface.

Most of the existing approaches does not provide adequate mechanisms to identify and specify the user's needs and requirements, as well as the validation of these requirements during the development process form the beginning to the end.

The solution to this problem is the integration of human factors in the development process [Hayne-96], i.e., to take into account the end-user's needs from the very beginning and incorporate a user model, this is, user issues in the development process.

One of the most widely used approaches to perform these considerations is the incorporation of use cases. Another technique followed by many research groups in the area of Human-Computer Interaction is task analysis. This technique aims to achieve generic and thus abstract models of end-user tasks. There seems to be much similarity between task analysis and use cases.

Our aim in this work is to enrich the Formal and Object-Oriented Specification Language, OASIS, by including aspects of user interface specification. Thus, we propose a user model with the purpose of fulfilling this existing gap.

The user interface is an important entity placed between the user and the application that the user needs to interact with the system. Thus, the user interface must be designed to be capable of communicating with both the end user and the application.

So, in this paper we propose a user model which models the user-system interaction. Besides, we also show a specific example in which this methodology is applied in the phases of Analysis and Design, showing the different steps in which it is developed.

The model we propose includes the generation of graphical and object-oriented diagrams to represent user interface issues.

Lozano, M.D., Ramos, I. An Object Oriented Approach for the Specification of User Interfaces. Proceedings of PACRIM'99 Conference. 2000.

USER INTERFACE SPECIFICATION:

In order to include the specification of end-user needs in this development process we have followed a Task Analysis technique taking into account the ideas proposed by [Jaa95] and the principles of our OASIS model to develop a user model which fills the existing gap in the model as far as the integration of human factors in the model are concerned.

The location of the user interface specification process within the model is the following:

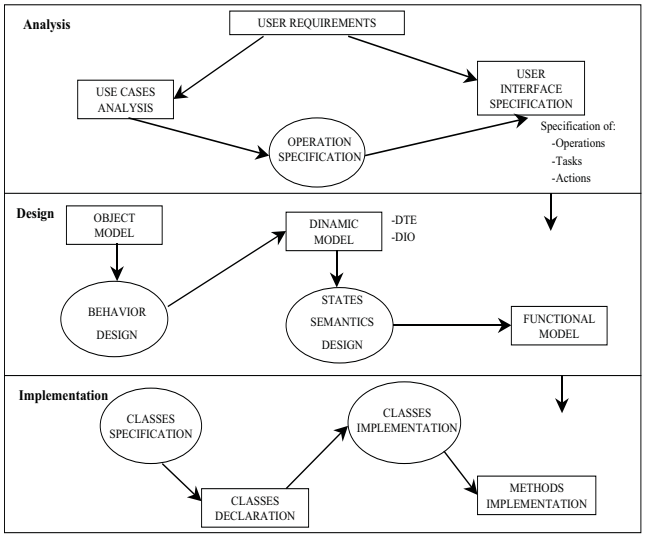


Fig. 1: Location of the object oriented specification process.

Next, we describe the model and in order to illustrate it, we will apply it to a simple case: a *Library*.

The process of developing the user interface for an Information System is tackled following the typical phases of Analysis, Design and Implementation.

1º) <u>Analysis Phase</u>:

The user model is represented by means of what we have called the *user diagrams*. These ***user diagrams*** consists of three different types of diagrams:

***** *The Dialog Structure Diagram*, which represents the windows and dialogs that the user needs to complete all the tasks he requires from the system, and the user selections to pass from one window to another.

In the *library* case this diagram would be the following:

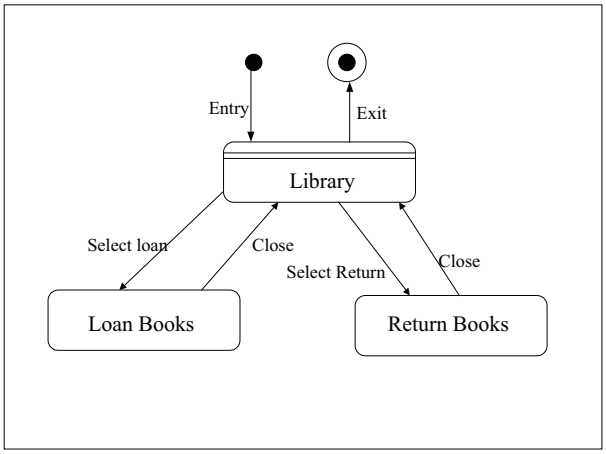


Fig. 2: Dialog Structure Diagram.

***** *The Component Specification Diagrams,* which models the contents of every one of the windows and dialogs represented in the previous diagram. Every window is composed of components and tools. For instance, the component specification diagram for the dialog *Loan_Book* would be the following:

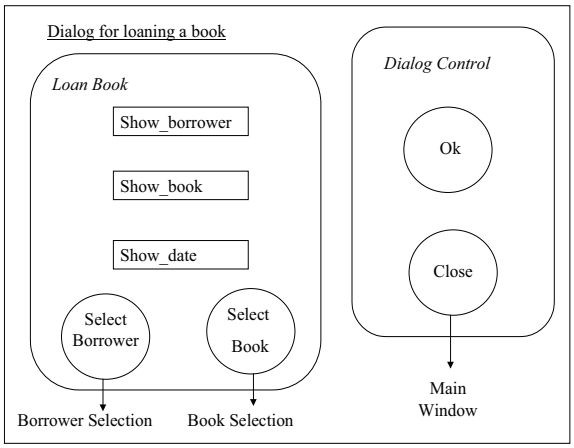


Fig. 3: Component Specification Diagram

***** *Components Behavior Definition Table* in which every one of the components and tools represented in the previous diagram is accurately described so that the user can understand the meaning of the components and tools through which he interacts and retrieves information from the system. In the following table we show the Components Behavior Definition Table for the dialog class *Loan_Book*.

Component	Control Tool	Visualisation Tool
Loan	***Select_Borrower***: Allows the selection of a borrower from the library database. ***Select_Book***: Allows the selection of a book from the library database.	***Show_Borrower***: Shows the borrower selected through the control tool. ***Show_Book***: Shows the book selected through the control tool. ***Show_date***: Shows the system date corresponding to the loan date.
Dialog Control	*Ok*: Allows the user to introduce a new loan in the library system. *Close*: Close the dialog and returns the user to the main window.	_____

Fig. 4: Component Behaviour Definition Table for the dialog class Loan_book.

2º) Design Phase:

The Graphical Models obtained in the first phase of Analysis are translated into an OASIS specification, which is the design tool within this methodology.

This phase in still in under study. Up to now all the process of translating the Conceptual Scheme models of the information System into an OASIS specification is correctly carried out in automatic way. So, the idea is to do the same with the user interface models.

3º) Implementation Phase:

This is the last step of the process. Implementation is tackled in a different way depending on the programming environment used.

In this phase we can use a Graphic User Interface Builder to get the visualisation of the specified interface.

CONCLUSIONS:

We can conclude that the methodology presented in this paper offers a set of operational methods which allows the design of user interfaces in an object oriented environment. It is not completely developed at this time, but, above all, it points out the importance of designing interaction between the end-user and the application before starting filling windows and windows with arbitrary sets of widgets (window gadgets) and other graphical elements. First to this, it is crucial to think about how should be the structure of the user interface of the system, considering first the general lines and then the details and final appearance, but only after having analysed the tasks, operations and components needed to fulfil the user's needs.

Besides, this object oriented approach facilitates the possibility of reusing user interface components. This possibility permits the construction of user interfaces form library classes with the advantage of implementing each individual feature only once.

Therefore, we can conclude that starting from these methods, some tools have been developed to cover all the phases of Software Production with a high level of automatization and in a well-defined Object-Oriented environment. Thus, OO-Method offers a set of operational methods which, starting from a graphical specification of the system generates its corresponding OASIS specification and automatically obtains the equivalent logic program.

ACKNOWLEGMENTS:

This work is partially supported by the MENHIR project from the Spanish government commission of Science and Technology with reference TIC97-0593-C05-01.

BIBLIOGRAPHY:

[Hayne96] Hayne, C. *Integrating Human Factors into the Software Development Lifecycle: A Literature Review. Computer Research Institute of Montreal, Canada. 1996*

[Jaa95] Jaaksi, A. *Object-Oriented Specification of User Interfaces.* Software Practice and Experience. November 95

[Let98] Letelier, P., Ramos, I., Sanchez, P. Pastor,O. *OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling.* SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[Loz97] Lozano, M.D., Ramos, I., Cuenca P. *"Application of an Object-Oriented Methodology for Automatic Software Production: OO-Method"*. Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'97). Pags. 985 – 989. Victoria, Canadá. 20-22 de Agosto, 1997.

[Pas95] Pastor,O, Ramos,I. *OASIS 2.1.1.: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach.* Octubre 95 (3 ed.)

[Pas96] Pastor,O, Pelechano,V., Bonet, B., Ramos,I. *An OO Methodological Approach for Making Automated Prototyping Feasible.* Proc. Of DEXA-96, Springer-Verlag, September 1996.

[Pas97] Pastor, O., Insfran, E., Pelechano, V., Romero, J., Meseguer, J. *OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods.* Procs. of 9th. International Conference, CAiSE'97. Barcelona, Catalonia, Spain, June 16-20, 1997.

User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development*

María D. Lozano, Pascual Gonzalez
Computer Science Department
University of Castilla-La Mancha
Albacete
(Spain)
{mlozano, pgonzalez}@info-ab.uclm.es

Isidro Ramos
Department of Information Systems and Computation
Polytechnic University
Valencia (Spain)
iramos@dsic.upv.es

Abstract

Nowadays, most of the research developed by industry and academy has led to several object-oriented methods highly adequate for the development of systems. But most of these methods do not include mechanisms (models) for identifying and specifying user needs and requirements as well as testing and validating requirements with end-users before, during and after development. These considerations are specially important in interactive systems, where the user interaction is very high and the user interface is a significant part of the system. As a result of this weakness, interactive systems developed using such methods can meet all technical requirements, be very robust, and yet be unusable by the end-user. This problem explain a large part of the frequently-observed phenomenon whereby large numbers of change requests to modify the services of an application are made after its deployment. As a solution to this problem we propose a method for integrating a user interface model, taking into account user needs and requirements, into the software life cycle. Besides, this approach to the development process ensures the quality of the delivered applications from the point of view of the end user.

1. Introduction.

Different studies have proved that the 48% of an application code is devoted to the user interface and that near 50% of the implementation time is devoted to implement the user interface [1].The more easy to use is the interface, the more difficult is its development. The user interface is that part of a computer program that manages the data flow produced during the interaction process between the user and the computer. The rest of the computer program is called the application. Due to the fact that the application should bring to the users different ways of conceiving, visualizing and manipulating the complex data structures, it is very important to establish a framework that allows to face and manage such complexity in a coherent way.

There exists several reasons that justify the complexity of user interface design and implementation [2], but perhaps the main reason is the difficulty of understanding the tasks the user must perform with the system and the characteristics of the different kind of users

* This work is partly supported by the MENHIR project from the Spanish government commission of Science and Technology with reference TIC97-0593-C05-01

Lozano, M. D., Ramos, I., González, P. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development". Proc. of the IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000). 2000.

managing it [3]. In this sense, it seems necessary to define new models that permit to collect user requirements and specify the different characteristics of the interface associated to a concrete application. This models should allow to define the specification of the interface independently [4], that is, the models used to specify the user interface should be different from those used to specify the application.

In this paper we present IDEAS (**I**nterface **D**evelopment **E**nvironment within o**AS**is), an object oriented and model-based approach for developing user interfaces within an automatic software production environment. This environment is supported by an object oriented model called OASIS (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [5], [6] and a methodology associated to the model, called OO-Method [7], [8], which covers the software life-cycle of analysis, design and implementation within the principles of the object oriented paradigm. So, for better understand our proposal, we present first the state-of-the-art of model-based interface development environments, and then, the ontology associated to the user interface model. After that we finally present IDEAS with an example of use.

2. Model-Based User Interface Development Environments. State of the Art

In this section we briefly describe the most important features of current model-based user interface development environments. In a state of the art report, B. Myers presents a classification of user interface software tools [9]. This classification is based on the way the user interfaces developers can specify the form and dynamic behavior of a user interface.

This classification is the following:

- *Language-based Tools*: These tools require the developer to program in a special purpose language.
- *Interactive Graphical Specification Tools*: These tools allow an interactive design of the user interface.
- *Model-based Generation Tools*: These tools use a high-level specification or model to generate the user interface automatically.

The last classification seems to be the most adequate to tackle the user interface development. The first two support the specification of either the dynamic behavior or the layout of the user interface in an easy way, but not both parts at the same time. And the same as most current tools, they only support the last phase of the user interface life-cycle development, and the abstractions they provide do not have a direct connection with the task analysis results.

On the other hand, model-based user interface development approach and its corresponding tools is an emerging technology that remedies these shortcomings.

The most representative feature of the model-based approach is that all aspects of the user interface design are represented using declarative models. This paradigm offers several benefits:

- User-centered development life-cycle.
- Centralized user interface specification.
- Design tools for an automated and interactive development.
- Reuse of the user interface designs.

So, for a user interface tool to be considered as a Model-based user interface development environment, it must accomplish the following criteria:

1) It must include a high-level, abstract and explicitly represented (declarative) model about the system under development.

2) It must clearly establish a technologically well supported relation between 1) and the final running user interface.

Currently there exist several model-based user interface software tools, as for instance, UIDE [10], HUMANOID [11], MOBI-D [12], MASTERMIND [13], [14], TRIDENT [15], FUSE [16], JANUS [17] and IDEAS our proposed model explained in following sections.

In the next figure we show the typical components and the development process of this model-based approach to better understand it.

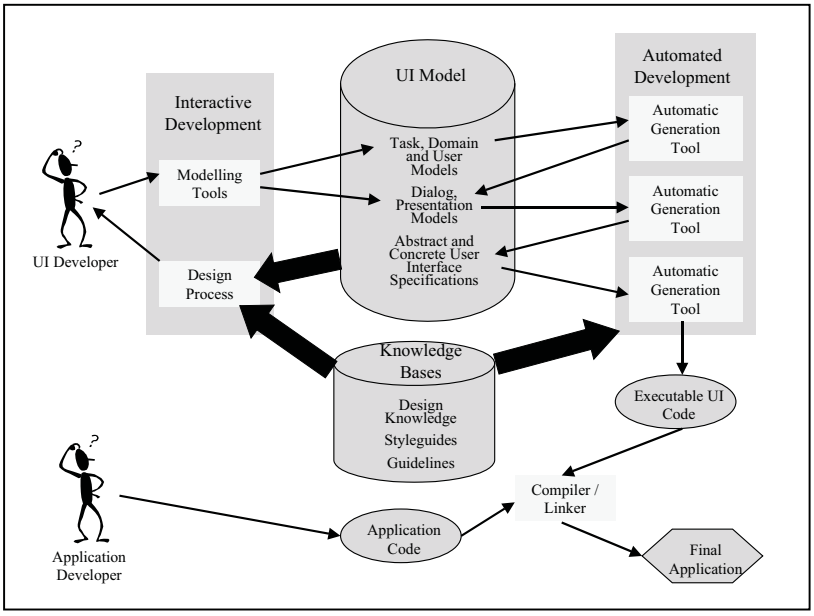


Figure 1: Generic architecture of a model-based user interface development environment.

The core of this architecture is the User Interface Model which includes different declarative models as for instance, the task model, domain model, presentation model, etc. This environment includes also tools for an interactive development (Modeling tools, design critics) and for an automated development.

The tools for automated development are in charge of the transformation between the declarative models and the final executable user interface.

The information needed by the design tools and automatic generation tools is represented in the knowledge bases.

Finally, the executable user interface code generated by the automatic generation tools is linked with the application code, obtaining so the final application.

3. IDEAS: User Interfaces Development within an Object Oriented Software Production Environment.

The aim of this research is the integration of a user interface model within an automatic and object oriented software production environment. The user interface specification process is tackled in parallel to the application development, and according to the common principles of Model-based User Interface Development Environments, as exposed in the previous section. This development process is depicted in the following figure:

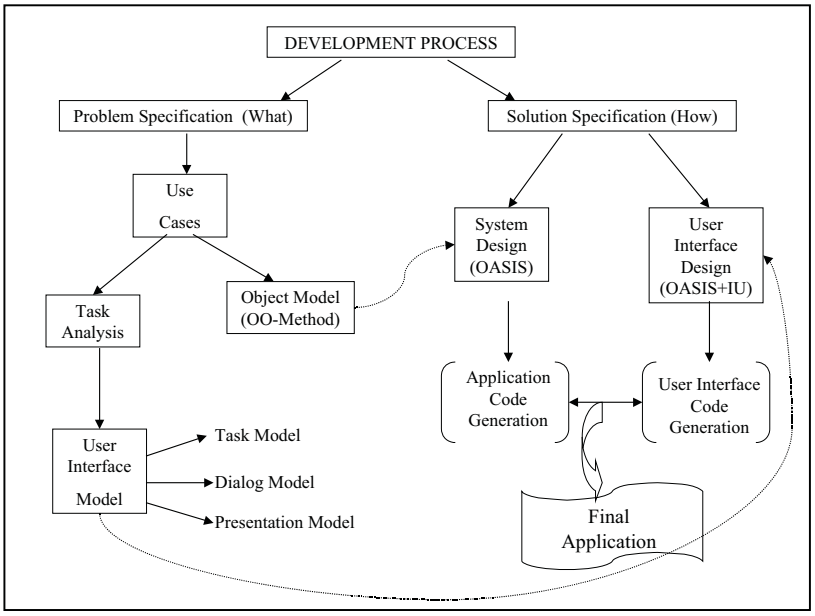


Figure 2: IDEAS: Interface development environment within OASIS.

So, IDEAS (**I**nterface **D**evelopment **E**nvironment within O**AS**IS) aims to be an automatic user interface development system integrated within the framework of OASIS and OO-Method, to support the automatic production of high-quality user interfaces. This is possible due to the fact that this environment is based on declarative models, with the advantages of this approach as stated in the previous section.

As depicted in figure 2, the user interface model is obtained from the task analysis, which is obtained from the use cases as starting point, not only to tackle the interface development but also the whole system.

The declarative models used in IDEAS are the following three: The task model, the dialog model and the presentation model.

The **Task Model** defines the ordered set of activities and actions the user has to perform to achieve a concrete purpose or goal.

The **Dialog Model** [18] is used to describe the "conversation" user-computer. It describes when the user can invoke commands, select or specify the input data and when the computer can require data from the user or display the output data. This model includes the generation of three different kinds of diagrams representing these matters, the *Dialog Structure Diagram*, the *Component Specification Diagram* and the *Component Behavior Definition Table* [19]. In order to illustrate this model, next we apply it to the very simple case of a *Library*.

* *The Dialog Structure Diagram* represents the windows and dialogs that the user needs to complete all the tasks he requires from the system, and the user selections to pass from one window to another. In the *library* case this diagram would be the following:

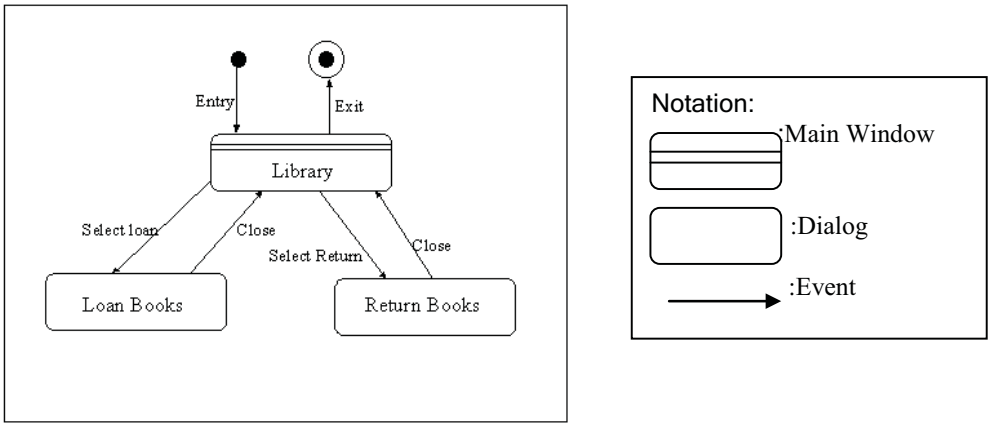


Figure 3: Dialog structure diagram.

* *The Component Specification Diagram,* which models the contents of every one of the windows and dialogs represented in the previous diagram. Every window is composed of components and tools. For instance, the component specification diagram for the dialog *Loan_Book* would be the following:

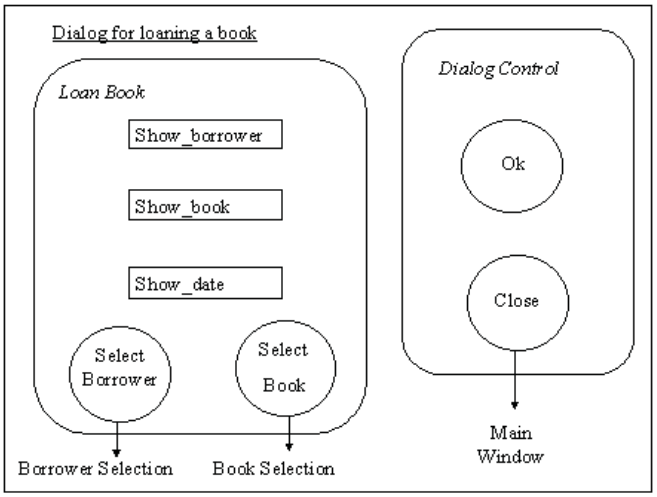


Figure 4: Component specification diagram

* *Component Behavior Definition Table* in which every one of the components and tools represented in the previous diagram is accurately described so that the user can understand the meaning of the components and tools through which he interacts and retrieves information from the system. In the following table we show the Component Behavior Definition Table for the dialog class *Loan_Book.*

Component	Control Tool	Visualisation Tool
Loan	***Select_Borrower***: Allows the selection of a borrower from the library database. ***Select_Book***: Allows the selection of a book from the library database.	***Show_Borrower***: Shows the borrower selected through the control tool. ***Show_Book***: Shows the book selected through the control tool. ***Show_date***: Shows the system date corresponding to the loan date.
Dialog Control	*Ok*: Allows the user to introduce a new loan in the library system. *Close*: Close the dialog and returns the user to the main window.	_____

Figure 5 : Component behavior definition table for the dialog class Loan_book.

The **Presentation Model** describes the final components that can appear in the end-user screen, its design characteristics and visual dependencies between them [20]. This model is closely related to the analysis model of the system as the information to be shown in this model is the same that is described and modeled in the analysis model, i.e., the class attributes identified in the system will be accessible by the end-user by means of the presentation model.

Starting form these declarative models it is possible to automatically generate the user interface code, that together with the application code generated by the parallel development process of the system will constitute the final resulting application.

As an intermediate step to this, in the design phase, the user interface specification is tackled with the specification language OASIS enriched to specify the user interface aspects in the same way that is currently used for the system specification.

Model-based code generation [21] produces application source code automatically from graphical models of system behavior and architecture. A formal system with precise syntax and semantics is needed for such an automation. OASIS is used as the target design language of the visual models. A translator generates OASIS code (extended with Graphical User Interface features) from the visual models. OASIS uses as formal model Dynamic Logic [22]. The operational semantics of such language is implemented as an Execution Model defining in a precise way [23] the mapping between problem domain concepts and solution domain ones. Pattern technology is used for establishing such mapping.

4. Ontological Model of IDEAS.

The Interface Model ontology of Interface Development Environments is based on two ideas: the programming language design, which provides foundations for any modeling process, and the object oriented user interface toolkit technology, which provides abstractions and control mechanisms. It can be said that user interface design is a process analogous to the design of a programming language [24]. So, the interface could be seen as composed of two

languages: one by means of which the end-user communicates with the computer, and the other by which the computer communicates with the end-user. This way, when developing the user interface of a system, it can be seen as a simultaneous design process of these two languages.

In this process we can distinguish three phases:

1) **Semantic Design:** The term "semantic" refers to the intentions of the end-user. To codify a system at this level we apply Task Analysis [25].This is the knowledge the end-user has in mind when using the system. This conceptual model is refined into a more detailed semantic model of the system by incorporating the functional requirements of the application, i.e., the interface specification will be completed by refining these models into more concrete models, which can be classified into syntactical and lexical. From them, the code generators transform the concrete models into the running application.

2) **Syntactic Design:** The syntactic model is defined to denote the detailed semantic model. It represents procedures that, when executed, cause the occurrence of some semantically prescribed effect. The main difference between the semantic and syntactic models is that syntactic models are specified in a procedural way, as a set of ordered steps, whereas semantic models are non-procedural, i.e., declaratives. Dialog techniques are syntactic whereas the user tasks that these dialog techniques are employed to accomplish are semantic.

3) **Lexical Design:** A lexical model, which describes low-level tokens, is defined together with the syntactic model. For the output language, these tokens includes keystrokes, mouse motion or mouse clicks, etc. Whereas for the output language these tokens includes output characters, sounds or graphical widgets.

Taking into account all this matters, the tasks organization and the functionality of the application are semantic concepts. Dialog is a syntactic concept and Presentation and interaction are lexical concepts. These models must define accurately the input and output languages and express their interrelation.

This ontology is depicted in the following figure:

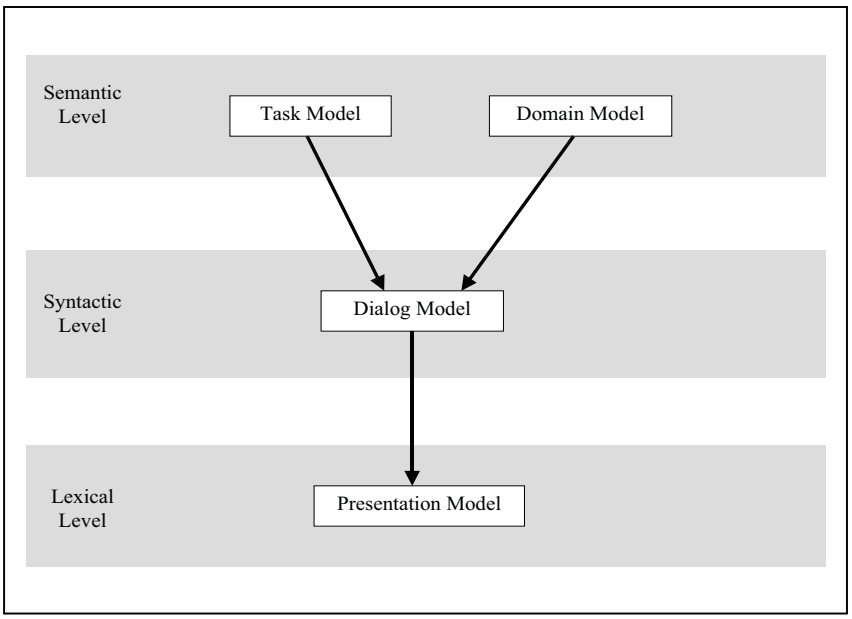


Figure 6: Ontology of the user interface model of IDEAS.

5. Conclusions.

The user interface development is quite a more complex task than the mere inclusion of different widgets in a series of windows. The complexity of designing and implementing a high-quality user interface provokes the need to include new models that collect the specification of such interface.

With the aim to achieve an object oriented environment for automatic software production, that not only can be used with the efficiency and simplicity that the end-user requires through a friendly but at the same time robust interface, we have presented IDEAS, a user interface development environment with the philosophy of OASIS and integrated into the development process of OO-Method.

IDEAS, as is based on declarative models, provides a descriptive information of the future components of the interface at a high level of abstraction. In the early development phases It also allows the study of how the interaction between the user and the system is performed. And in the last phases it allows the automatic code generation starting from the models.

To better understand the approach followed for the development of this environment, we have presented, in a state-of-the-art section, different model-based interface development environments, and the ontology associated to the interface model.

And finally, remark the more and more importance that the user interface is getting, above all regarding interactive systems, as a means of interacting with a system and taking the most profit of it. This is the main motivation of this research.

6. Bibliografy.

[1] B. A. Myers, M. B. Rosson: Survey on User Interface Programming. In Striking a Balance. Proceedings CHI'92. Monterey, May 1992, New York: ACM Press, 1992, 195-202

[2] B.A. Myers: *Why are Human-Computer Interfaces Difficult to Design and Implement*. Tech. Report CMU-CS-93-183, CMU, Jul 1993.

[3] J. Gould, C. Lewis: *Designing for Usability – Key Principles and What Designers Think*. Commun. ACM, Vol 28, 1985, pp.300-311.

[4] W. Swartout, R. Blazaer: *The Inevitable Intertwining of Specification and Implementation*. Commun ACM, Vol 25, No. 7, Jul 1982, pp.438-440.

[5] O. Pastor, I. Ramos. OASIS 2.1.1.: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach. Octubre 95 (3 ed.)

[6] P. Letelier, I. Ramos, P. Sanchez, O. Pastor. *OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling.* SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[7] O. Pastor, E. Insfran, V. Pelechano, J. Romero, J. Meseguer. *OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods.* Procs. of 9th. International Conference, CAiSE'97. Barcelona, Catalonia, Spain, June 16-20, 1997.

[8] M.D. Lozano, I. Ramos, P. Cuenca. *"Application of an Object-Oriented Methodology for Automatic Software Production: OO-Method".* Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'97). Pags. 985 – 989. Victoria, Canadá. 20-22 de Agosto, 1997.

[9] B. A. Myers: User Interface Software Tools. ACM Transactions on Computer-Human Interaction 2 (1995), 1, 64-103

[10] J. Foley, P. Sukaviriya: History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In F. Paterno (ed.): Interactive Systems: Design, Specification and Verification. Berlín: Springer, 1995, 3-14.

[11] P. Szekely et al: Beyond Interface Builders: Model-Based Interface Tools. In: Bridges between Worlds. Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 383-390.

[12] A. Puerta: The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 19-36.

[13] R. Neches et al.: Knowledgeable Development Environments Using Shared Design Models. In: Proceedings of the 1993 International Workshop on Intelligent User Interfaces (Orlando, Enero 1993). New York: ACM Press, 1993, 63-70.

[14] P. Szekely, et al.: Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach. In: Engineering for Human-Computer Interaction. Proceedings of the IFIP Conference on Engineering for Human-Computer Interaction (Yellowstone Park, Agosto 1995). London: Chapman &Hall, 1996, 120-150

[15] F. Bodart et al.: Towards a Systematic Building of Software Architectures: teh TRIDENT Methodological Guide. In: Design, Specification and Verification of Interactive Systems. Wien: Springer, 1995, 262-278.

[16] F. Lonczewski: The FUSE System: An Integrated User Interface Design Environment. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 37-56.

[17] H. Balzert et al.: The JANUS application Development Environment Generating More than the User Interface. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 183-205.

[18] M. Lozano, I. Ramos: An Object Oriented Approach for the Specification of User Interfaces. PACRIM'99, Victoria -Canadá- Agosto, 1999.

[19] M. Lozano, I. Ramos: Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. ECOOP'99, Lisboa -Portugal-, Junio, 1999.

[20] P.J. Molina: Especificación de Interfaz de Usuario en OO-Method. Proyecto Final de Carrera, Valencia, 1998.

[21]Pastor,O.;Canós,J.H.; Ramos,I. From CASE to CARE (Computer-Aided Requirements Engineering) Conceptual Modeling – ER'99. Lecture Notes in Computer Science (1728), págs: 278-293, Springer-Verlag, 1999, ISBN: 3-540-66686-9 ; ISSN : 0302-9743, Paris.

[22] P. Sánchez, P. Letelier, I. Ramos. Animating formal specifications with Inheritance in a DL-based framework. Requirements Engineering Journal, Springer-Verlag, vol.4 num.4, March, 2000.

[23] P. Letelier, P. Sánchez, I. Ramos. "Prototyping a requirements specification through an automatically generated concurrent logic program". Proceedings of First International Workshop on Practical Aspects of Declarative Languages (PADL'99), Gopal Gupta (Ed.), LCNS 1551, pp.31-45, ISSN 0302-9743, ISBN 3-540-65527-1, Springer-Verlag, Texas, USA, Enero 1999.

[24] J. Foley, A. Van Dam, S. Feiner, A. Hughes: Computer Graphics: Principles and Practice. (2nd Ed.) Reading: Addison Wesley, 1990.

[25] D. Diaper: Task Analysis for Human-Computer Interaction. Chichester: Ellis Horwood, 1989.

Models and Ontologies for the Specification of User Interfaces ⊕

María D. Lozano*, Pascual Gonzalez*, Isidro Ramos+

*Computer Science Department
University of Castilla-La Mancha
Albacete
(Spain)
e-mail: {mlozano, pgonzalez}@info-ab.uclm.es

+Department of Information Systems and Computation
Polytechnic University
Valencia (Spain)
e-mail : iramos@dsic.upv.es

Keywords: Object Oriented Specification and Modeling Techniques, Ontology, User Interfaces

Abstract

Nowadays, most of the research developed by industry and academy have led to several object-oriented methods highly adequate for the development of systems. But most of these methods do not include mechanisms (models) for identifying and specifying user needs and requirements as well as testing and validating requirements with end-users before, during and after development. These considerations are specially important in interactive systems, where the user interaction is very high and user interface is a significant part of the system. As a result of this weakness, interactive systems developed using such methods can meet all technical requirements, be very robust, and yet be unusable by the end-user. This problem explain a large part of the frequently-observed phenomenon whereby large numbers of change requests to modify the services of an application are made after its deployment. As a solution to this problem we propose a method for integrating a user interface model, taking into account user needs and requirements, in the software life cycle. Besides, this approach to the development process ensures the quality of delivered applications from the point of view of the end user.

1 Introduction.

Different studies have proved that the 48% of an application code is devoted to the user interface and that near 50% of the implementation time is devoted to implement the user interface [Mye92].The more easy to use is the interface, the more difficult is its development. The user interface is that part of a computer program which manages the in/out data flow produced during the interaction process between the user and the computer. The rest of the computer program is called the application. Due to the fact that the application should bring to the users different ways of conceiving, visualizing and manipulating the complex data structures, it is very important to establish a framework that allows to face and manage such complexity in a coherent way.

⊕ This work is partly supported by the MENHIR project from the Spanish government commission of Science and Technology with reference TIC97-0593-C05-01

Lozano, M. D, González, P., Ramos, I. Models and Ontologies for the Specification of User Interfaces. Proc. of the Workshop on Model Engineering at ECOOP'00, 2000.

There exists several reasons that justify the complexity of design and implementation of User Interfaces [Mye93], but perhaps the main reason is the difficulty of understanding the tasks the user must perform with the system and the characteristics of the different kind of users managing it [Gou85]. In this sense, it seems necessary to define **new models** that allow to collect user requirements and specify the different characteristics of the interface associated to a concrete application. This models should allow to define the specification of the interface independently [Swa82], that is, the models used to specify the user interface should be different from those used to specify the application.

In this paper we present IDEAS[1], an object oriented and **model-based approach** for developing user interfaces within an automatic software production environment. This environment is supported by an object oriented model called OASIS[2] [Pas95], [Let98] and a methodology associated to the model, called OO-Method [Pas97], [Loz97], which covers the software life-cycle of analysis, design and implementation within the principles of the object oriented paradigm. So, for better understand our proposal, we present first the state-of-the-art of model-based interface development environments, and then, the ontology associated to the user interface model. After that we finally present IDEAS with an example of use.

2 Model-Based User Interface Development Environments. State of the Art.

In this section we briefly describe the most understanding features of current model-based user interface development environments. In a state of the art report, B. Myers presents a classification of user interface software tools [Mye95]. This classification is based on the way the user interfaces developers can specify the form and dynamic behavior of a user interface. This classification is the following:

- Language-based Tools: These tools require the developer to program in a special purpose language.
- Interactive Graphical Specification Tools: These tools allow an interactive design of the user interface.
- Model-based Generation Tools: These tools use a high-level specification or model to generate the user interface automatically.

The last classification seems to be the most adequate to tackle the user interface development. The first two support the specification of either the dynamic behavior or the layout of the user interface in an easy way, but not both parts at the same time. And the same as most current tools, they only support the last phase of the user interface life-cycle development, and the abstractions they provide do not have a direct connection with the task analysis results.

On the other hand, model-based user interface development approach and its corresponding tools is an emerging technology that remedy these shortcomings.

---

[1] **I**nterface **D**evelopment **E**nvironment within o**AS**is.

[2] **O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems

The most representative feature of the model-based approach is that all aspects of the user interface design are represented using declarative models. This paradigm offers several benefits:

- User-centered development life-cycle.
- Centralized user interface specification.
- Design tools for an automated and interactive development.
- Reuse of the user interface designs.

So, for a user interface tool to be considered as a Model-based user interface development environment, it must accomplish the following criteria:

1) It must include a high-level, abstract and explicitly represented (declarative) model about the system under development.
2) It must clearly establish a technologically well supported relation between 1) and the final running user interface.

In the next figure we show the typical components and the development process of this model-based approach to better understand it.

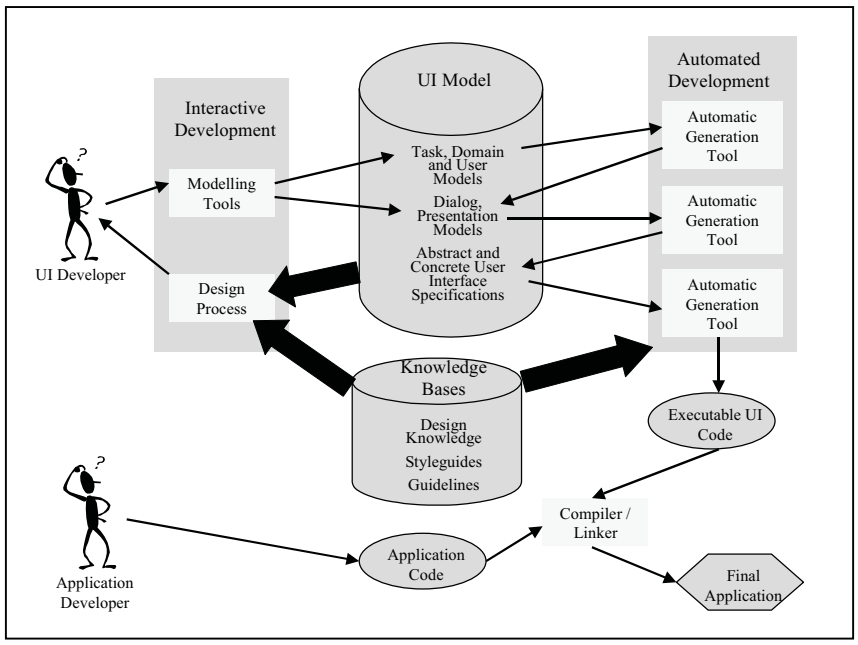


Figure 1: Generic Architecture of a Model-based User Interface Development Environment.

3 IDEAS: User Interfaces Development within an Object Oriented Software Production Environment.

The aim of this research is the integration of a User Interface Model within an automatic and object oriented software production environment. The user interface specification process is tackled in parallel to the application development, and according to the common principles of Model-based User Interface Development Environments, as exposed in the previous section. This model-based development process is depicted in the following figure:

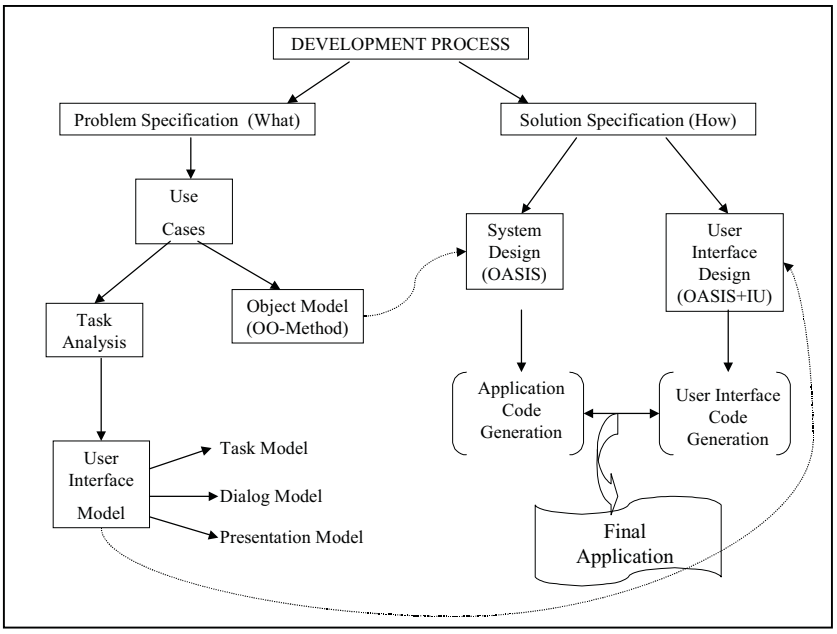


Figure 2: IDEAS: Interface Development Environment within OASIS.

So, IDEAS (**I**nterface **D**evelopment **E**nvironment within O**ASI**S) aims to be a model-based and automatic user interface development system integrated within the framework of OASIS and OO-Method, to support the automatic production of high-quality user interfaces. This is possible due to the fact that this environment is based on declarative models, with the advantages of this approach as stated in the previous section.

As depicted in figure 2, the user interface model is obtained from the task analysis, which for its part is obtained from the use cases as starting point, not only to tackle the interface development but also the whole system.

The declarative models used in IDEAS are the following three: The task model, the dialog model and the presentation model.

The **Task Model** defines the ordered set of activities and actions the user has to perform to achieve a concrete purpose or goal.

The **Dialog Model** [Loz99a] is used to describe the "conversation" user-computer. It describes when the user can invoke commands, select or specify the input data and when the computer can require data from the user or display the output data. This model includes the generation of three different kinds of diagrams representing these matters, the *Dialog Structure Diagram*, the *Component Specification Diagram* and the *Component Behavior Definition Table* [Loz99b]. In order to illustrate this model, next we apply it to the very simple case of a *Library*.

- *The Dialog Structure Diagram* represents the windows and dialogs that the user needs to complete all the tasks he requires from the system, and the user selections to pass from one window to another. In the *library* case this diagram would be the following:

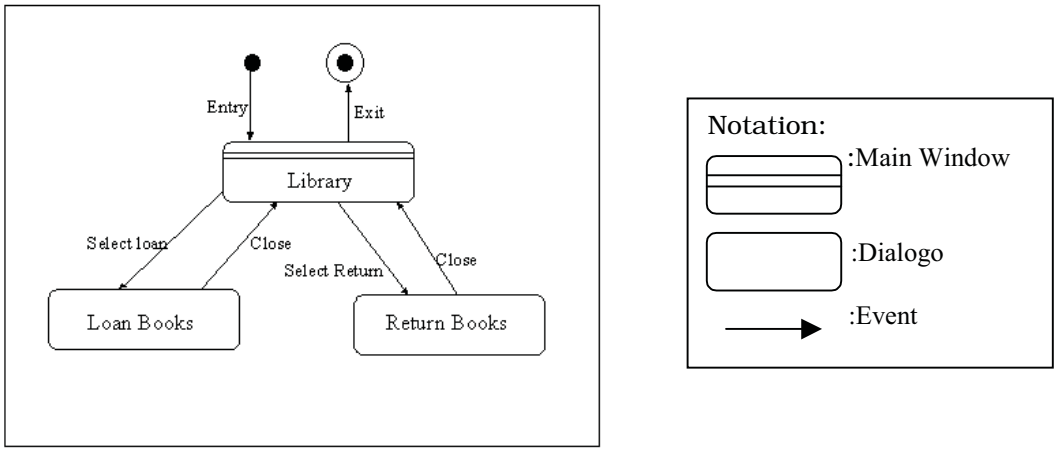


Fig. 2: Dialog Structure Diagram.

- *The Component Specification Diagrams,* which models the contents of every one of the windows and dialogs represented in the previous diagram. Every window is composed of components and tools. For instance, the component specification diagram for the dialog *Loan_Book* would be the following:

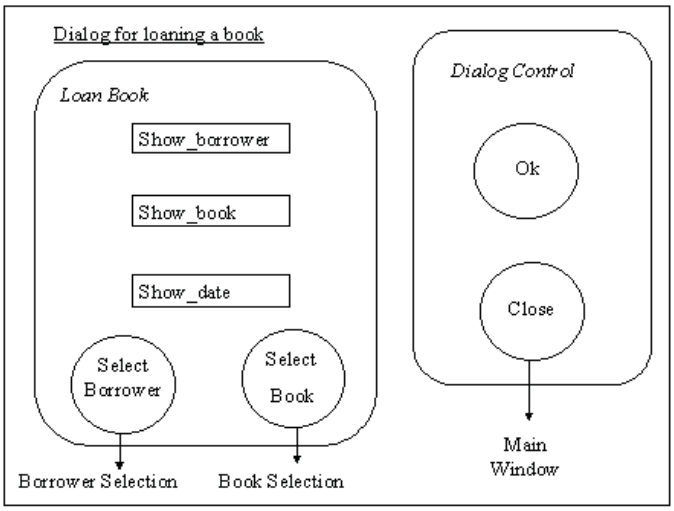


Fig. 3: Component Specification Diagram

- *Components Behavior Definition Table* in which every one of the components and tools represented in the previous diagram is accurately described so that the user can understand the meaning of the components and tools through which he interacts and retrieves information from the system. In the following table we show the Components Behavior Definition Table for the dialog class *Loan_Book*.

Component	Control Tool	Visualisation Tool
Loan	***Select_Borrower***: Allows the selection of a borrower from the library database. ***Select_Book***: Allows the selection of a book from the library database.	***Show_Borrower***: Shows the borrower selected through the control tool. ***Show_Book***: Shows the book selected through the control tool. ***Show_date***: Shows the system date corresponding to the loan date.
Dialog Control	*Ok*: Allows the user to introduce a new loan in the library system. *Close*: Close the dialog and returns the user to the main window.	————

Fig. 4: Component Behaviour Definition Table for the dialog class Loan_book.

The **Presentation Model** describes the final components that can appear in the end-user screen, its design characteristics and visual dependencies between them [Mol98]. This model is closely related to the analysis model of the system as the information to be shown in this model is the same that is described and modeled in the analysis model, i.e., the class attributes identified in the system will be accessible by the end-user by means of the presentation model.

Starting form these declarative models it is possible to automatically generate the user interface code, that together with the application code generated by the parallel development process of the system will constitute the final resulting application.

As an intermediate step to this, in the design phase, the user interface specification is tackled with the specification language OASIS enriched to specify the user interface aspects in the same way that is currently used for the system specification.

4 Ontology Model of IDEAS.

The Interface Model ontology of Interface Development Environments is based on two ideas: the programming language design, which provides foundations for any modeling process, and the object oriented user interface toolkit technology, which provides abstractions and control mechanisms. It can be said that user interface design is process analogous to the design of a programming language [Fol90]. So, the interface could be seen as composed of two languages: one by means of which the end-user communicates with the computer, and the other by which the computer communicates with the end-user. This way, when developing the user interface of a system, it can be seen as a simultaneous design process of these two languages.

In this process we can distinguish three phases:

1) **Semantic Design:** The term "semantic" refers to the intentions of the end-user. To codify a system at this level we apply Task Analysis [Dia89].This is the knowledge the end-user has in mind when using the system. This conceptual model is refined into a more detailed semantic model of the system by incorporating the functional requirements of the application, i.e., the interface specification will be completed by refining these models into more concrete models,

which can be classified in syntactic and lexical. From them, the code generators transforms the concrete models into the running application.

2) **Syntactic Design:** The syntactic model is defined to denote the detailed semantic model. It represents procedures that, when executed, cause the occurrence of some semantically prescribed effect. The main difference between the semantic and syntactic models is that syntactic models are specified in a procedural way, as a set of ordered steps, whereas semantic models are non-procedural, i.e., declaratives. Dialog techniques are syntactic whereas the user tasks that these dialog techniques are employed to accomplish are semantic.

3) **Lexical Design:** A lexical model, which describes low-level tokens, is defined together with the syntactic model. For the output language, these tokens includes keystrokes, mouse motion or mouse clicks, etc. Whereas for the output language these tokens includes output characters, sounds or graphical widgets.

Taking into account all this, the tasks organization and the functionality of the application are semantic concepts. Dialog is a syntactic concept and Presentation and interaction are lexical concepts. These models must define accurately the input and output languages and express their interrelation.

This ontology is depicted in the following figure:

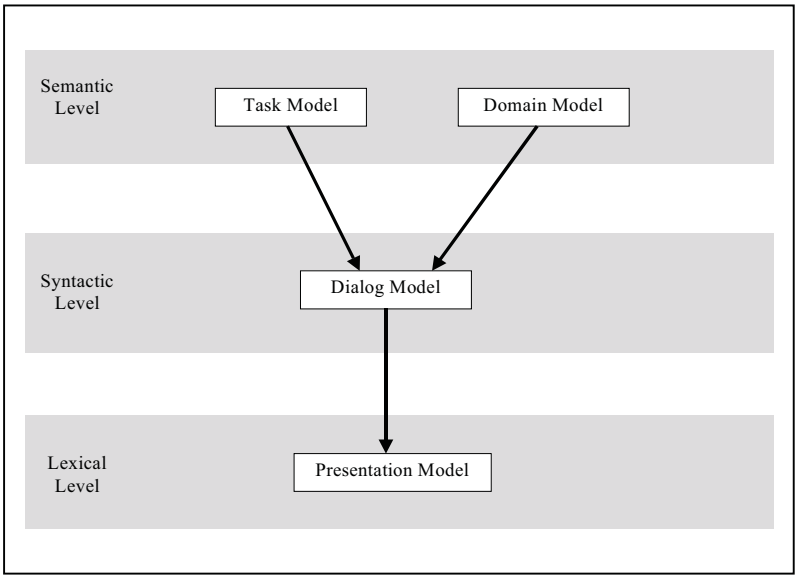


Figure 3: Ontology of the User Interface Model of IDEAS.

5 Conclusions.

The user interface development is a quite more complex task than the mere inclusion of different widgets in a series of windows. The complexity of designing and implementing a high-quality user interface provokes the necessity of including new models that collects the specification of such interface.

With the aim to achieve an object oriented environment for automatic software production, we have presented IDEAS, a user interface development environment with the philosophy of OASIS and integrated into the development process of OO-Method.

IDEAS, as is based on declarative models, provides a descriptive information of the future components of the interface at a high level of abstraction. In the early development phases It also allows the study of how the interaction between the user and the system is performed. And in the last phases it allows the automatic code generation starting from the models.

To better understand the approach followed for the development of this environment , we have presented, in a state-of-the-art section, different model-based interface development environments, and the ontology associated to the interface model.

6 Bibliografy.

[Dia89] D. Diaper: Task Analysis for Human-Computer Interaction. Chichester: Ellis Horwood, 1989.

[Fol90] J. Foley, A. Van Dam, S. Feiner, A. Hughes: Computer Graphics: Principles and Practice. (2nd Ed.) Reading: Addison Wesley, 1990.

[Gou85] J. Gould, C. Lewis: *Designing for Usability – Key Principles and What Designers Think*. Commun. ACM, Vol 28, 1985, pp.300-311.

[Let98] P. Letelier, I. Ramos, P. Sanchez, O. Pastor. *OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling.* SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[Loz97] M.D. Lozano, I. Ramos, P. Cuenca. *"Application of an Object-Oriented Methodology for Automatic Software Production: OO-Method".* Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'97). Pags. 985 – 989. Victoria, Canadá. 20-22 de Agosto, 1997.

[Loz99a] M.Lozano, I. Ramos: An Object Oriented Approach for the Specification of User Interfaces. PACRIM'99, Victoria -Canadá- Agosto, 1999.

[Loz99b] M. Lozano, I. Ramos: Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. ECOOP'99, Lisboa -Portugal-, Junio, 1999.

[Mol98] P.J. Molina: Especificación de Interfaz de Usuario en OO-Method. Proyecto Final de Carrera, Valencia, 1998.

[Mye92] B. A. Myers, M. B. Rosson: Survey on User Interface Programming. In Striking a Balance. Proceedings CHI'92. Monterey, May 1992, New York: ACM Press, 1992, 195-202

[Mye93] B.A. Myers: *Why are Human-Computer Interfaces Difficult to Design and Implement*. Tech. Report CMU-CS-93-183, CMU, Jul 1993.

[Mye95] B. A. Myers: User Interface Software Tools. ACM Transactions on Computer-Human Interaction 2 (1995), 1, 64-103

[Pas95] O. Pastor, I. Ramos. OASIS 2.1.1.: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach. Octubre 95 (3 ed.)

[Pas97] O. Pastor, E. Insfran, V. Pelechano, J. Romero, J. Meseguer. *OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods.* Procs. of 9th. International Conference, CAiSE'97. Barcelona, Catalonia, Spain, June 16-20, 1997.

[Swa82] W. Swartout, R. Blazaer: *The Inevitable Intertwining of Specification and Implementation*. Commun ACM, Vol 25, No. 7, Jul 1982, pp.438-440.

Aproximación Basada en Modelos para la Especificación de Interfaces de Usuario

María Dolores Lozano[†], Pascual González[†], Isidro Ramos[‡]

[†]Departamento de Informática
Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha 02071-ALBACETE
Tlf: 967.59.92.00 Fax: 967.59.92.24 [mlozano, pgonzalez]@info-ab.uclm.es

[‡]Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n 46020-VALENCIA
Tlf: 96.387.93.59 Fax: 96.387.73.59. iramos@dsic.upv.es

Resumen

La importancia del diseño y creación de interfaces de usuario es cada vez mayor, siendo considerado como un campo de investigación de gran relevancia y tal vez uno de los que producen un mayor impacto en los diferentes usuarios de los sistemas de información. Podemos considerar varias disciplinas dentro de esta área de investigación, que pasan desde la definición de métodos de análisis y diseño de interfaces de usuario, la creación de nuevas alternativas de representar o acceder a la información, hasta el estudio de los factores humanos que están detrás del manejo de dichas interfaces. En este trabajo vamos a realizar una revisión de las investigaciones más interesantes realizadas hasta el momento en el ámbito de los métodos de definición de interfaces de usuario, basados en la utilización de diferentes modelos semánticos. Indicando finalmente nuestra aportación en este terreno, un entorno de desarrollo de interfaces de usuario dentro del proceso de desarrollo de OO-Method y siguiendo la filosofía de OASIS.

Palabras clave: Interfaces de Usuario, Diseño de Sistemas Interactivos.

1 Introducción.

Existen varios motivos que justifican la complejidad del diseño e implementación de los IU [1], Pero tal vez la principal es la dificultad de comprender las tareas que el usuario desea realizar sobre el sistema y las

Lozano, M. D., Ramos, I. Aproximación Basada en Modelos en la especificación de Interfaces de Usuario. Actas de las I Jornadas de Interacción Persona-Ordenador (INTERACCIÓN-20), 2000.

características de los diferentes usuarios que lo manejarán [2]. En este sentido parece necesario definir nuevos modelos que permitan recoger los requerimientos y especificar las diferentes características de la interfaz asociada a una determinada aplicación. Estos modelos deben permitir que se defina una especificación de la interfaz independiente [3]. Es decir, los modelos utilizados para especificar la interfaz deben ser distintos de los utilizados para especificar la aplicación.

Diferentes estudios han demostrado que el 48% del código de una aplicación se dedica a la interfaz de usuario, y que cerca del 50% del tiempo dedicado a implementación es para implementar la parte de Interfaz de Usuario [4]. A medida que la interfaz de usuario es más fácil de usar, implica que es más difícil de construir. Los desarrolladores de IUs necesitan herramientas que proporcionen un buen soporte para el desarrollo de IUs avanzadas.

En este trabajo, se realiza una presentación de los principales estudios sobre definición y utilización de modelos en la especificación de interfaces de usuario. Después de dicha presentación, se indican las características básicas de los nuevos entornos de desarrollo de interfaces de usuario basados en modelos. Y finalmente se expone el modelo de interfaz de usuario aportado en el contexto de OO-Method y como enriquecimiento del lenguaje de especificación OASIS, así como la ontología asociada a este modelo.

2 Entornos de desarrollo de interfaces de usuario basada en modelos. Estado del arte.

En este apartado se analizan los diferentes usos de modelos declarativos como parte del Modelo de Interfaz en un entorno de desarrollo de interfaces de usuario basadas en modelos, de ahora en adelante, EDIBAM, para abreviar.

Durante los últimos años se han creado distintas herramientas para soportar el desarrollo de IUs, por ejemplo, Toolkits, User Interface Management Systems, Interface Builders, UIDE, etc. En un report sobre el estado del arte, B. Myers presenta una clasificación de estas herramientas software de Interfaz de Usuario [5]. Esta clasificación se basa en la manera en la que los desarrolladores de IUs pueden especificar la forma y el comportamiento dinámico de una IU.

La clasificación es la siguiente:

- *Herramientas basadas en lenguajes*: Estas requieren que el desarrollador programe en un lenguaje de propósito especial.

- *Herramientas de especificación gráfica interactiva*: Estas permiten un diseño interactivo de la IU.
- *Herramientas de generación basadas en modelos*: Estas hacen uso de un modelo o especificación de alto nivel para generar la IU de forma automática.

De estos tres tipos, las últimas parecen ser las más adecuadas para abordar el desarrollo de la IU. Las dos primeras permiten la especificación tanto del comportamiento dinámico como de la estructura de la IU de forma sencilla, pero no ambas partes a la vez. Y al igual que la mayoría de las herramientas actuales soportan solamente la fase final de desarrollo del ciclo de vida de la IU, y las abstracciones que proporcionan no tienen una conexión directa con los resultados del análisis de tareas.

2.1 Arquitectura genérica de los EDIBAMs

Los componentes típicos y el procedimiento de desarrollo principal de un EDIBAM se muestran en la Fig. 1

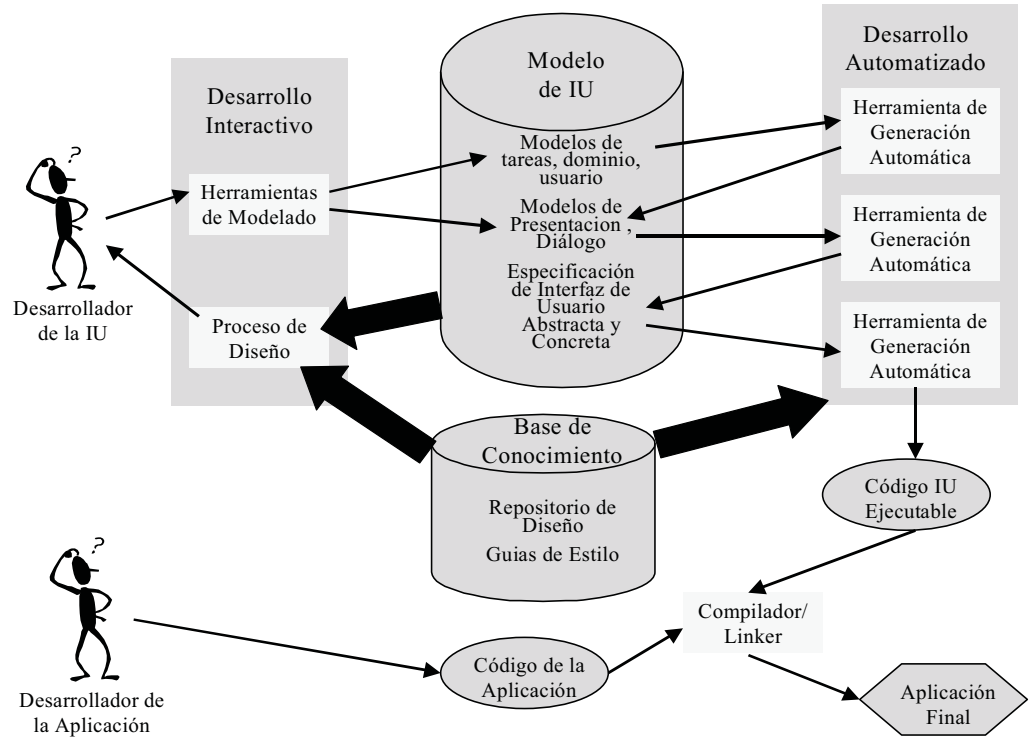


Figura 1: Arquitectura Genérica de un Entorno de Desarrollo de Interfaces de Usuario Basado en Modelos.

El componente central del EDIBAM es el Modelo de Interfaz que incluye diferentes modelos declarativos. En el siguiente apartado se detallarán como son estos modelos declarativos. En este entorno también intervienen herramientas para el desarrollo interactivo (Herramientas de Modelado y de Diseño) y para el desarrollo automático. Las herramientas para la generación automática se encargan de las transformaciones entre los diferentes modelos declarativos y la representación implementada y ejecutable de la IU deseada. La información requerida por las herramientas de generación automática y de diseño está representada en la Base de Conocimiento.

3 IDEAS: Desarrollo de Interfaces de Usuario en un Entorno de Producción Automática de Software Orientado a Objetos.

El objetivo de este trabajo es la integración de un Modelo de Interfaz de Usuario dentro de OASIS a partir del cual se pueda generar la interfaz de la aplicación de forma automática. El proceso de especificación de la Interfaz de Usuario se realiza de forma paralela al desarrollo de lo que es la aplicación en sí, y de acuerdo a los principios de los EDIBAMs estudiados previamente. Este proceso de desarrollo se puede ver en la siguiente figura:

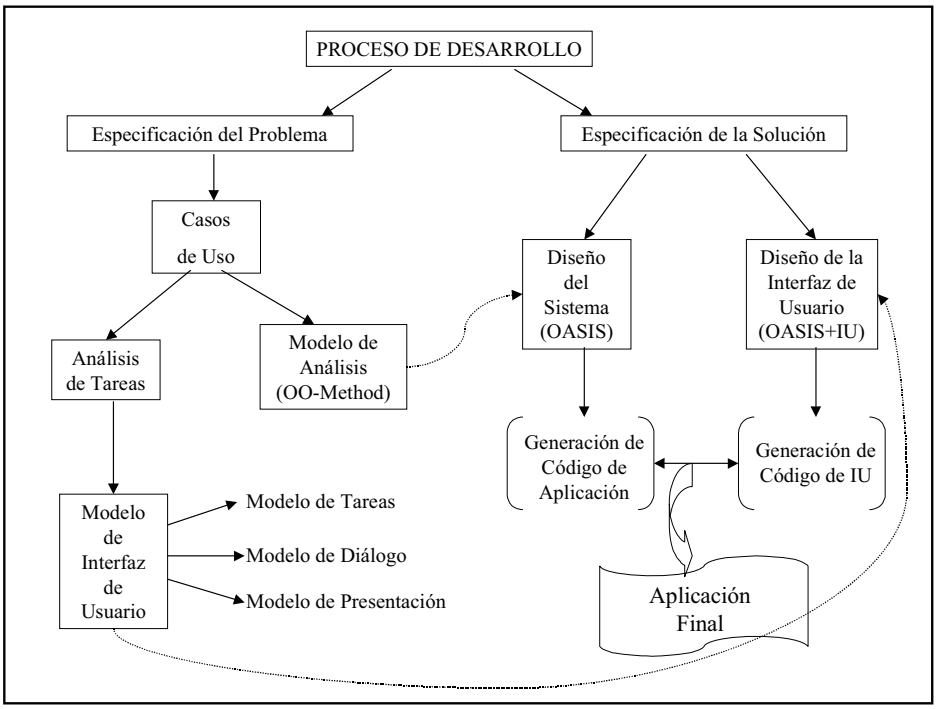


Figura 2: Proceso de Desarrollo de Sistemas en el Entorno de Producción Automática de Software de OASIS enriquecido con el Modelo de IU

De esta forma IDEAS (**I**nterface **D**evelopment **E**nvironment within O**AS**IS), pretende ser un sistema de desarrollo automático de Interfaces de Usuario integrado en el marco de trabajo de OASIS [6] y OO-Method [7], [8], para soportar de forma automática la producción de IUs de alta calidad. Esto es posible porque está basada en modelos declarativos, con las ventajas ya comentadas de estas aproximaciones.

Los modelos declarativos utilizados en IDEAS son tres: el modelo de tareas, el modelo de diálogo y el modelo de presentación.

El **modelo de tareas** define el conjunto ordenado de actividades y acciones que tiene que llevar a cabo el usuario para conseguir un objetivo concreto.

El **modelo de diálogo**[9], que se utiliza para describir la "conversación" hombre-sistema. Describe cuando el usuario puede invocar comandos, seleccionar o especificar entradas de información y cuando el computador puede requerir información del usuario y presentar la información de salida. Este modelo incluye la generación de tres tipos de diagramas que representan estos aspectos, el Diagrama de Estructura de Diálogos (DED), el Diagrama de Especificación de Componentes (DEC) y la Tabla de Definición de Comportamiento de Componentes (TDC), [10].

El **modelo de presentación**, que describe las componentes que pueden aparecer en la pantalla del usuario final, sus características de diseño y las dependencias visuales existentes entre ellas. Este modelo está estrechamente relacionado con el modelo de análisis del sistema, ya que la información que se muestra en este modelo es aquella descrita en el modelo de análisis, es decir, los atributos característicos de las clases identificadas en el sistema serán accesibles al usuario a través del modelo de presentación.

A partir de los modelos declarativos que componen el modelo de interfaz de usuario, se genera automáticamente el código de IU, que junto con el código de la aplicación generado en el proceso de desarrollo del sistema paralelo formará la aplicación resultante final.

Como paso intermedio a esto, en la fase de diseño la especificación de la IU se lleva a cabo en el lenguaje de especificación OASIS enriquecido para especificar la parte de interfaz de usuario, de la misma forma que se viene utilizando para la especificación del sistema.

4 Conclusiones

Con el objetivo de conseguir un entorno de producción automática de software orientado a objetos, que no sólo genere sistemas software de calidad, sino que además permita ser utilizado con la sencillez y eficacia que requiere el usuario a través de una interfaz de usuario amigable pero a la vez robusta, se ha presentado IDEAS, un entorno de desarrollo de interfaces de usuario con la filosofía de OASIS e integrado en el proceso de desarrollo de OO-Method.

IDEAS, al estar basada en modelos declarativos, proporciona información descriptiva de alto nivel de abstracción de los futuros componentes de la interfaz de usuario, a la vez que permite estudiar de forma precisa cómo se llevará a cabo la interacción del usuario con el sistema en las primeras etapas de desarrollo. Y en las fases finales, permite la generación automática de código a partir de los modelos.

Destacar por último la importancia que va cobrando cada vez más, sobre todo en los sistemas interactivos, la interfaz de usuario como medio para interactuar y sacar el máximo partido de un sistema. Esta es la principal motivación de la investigación que se está llevando a cabo.

Referencias

[1] B.A. Myers: *Why are Human-Computer Interfaces Difficult to Design and Implement*. Tech. Report CMU-CS-93-183, CMU, Jul 1993.

[2] J. Gould, C. Lewis: *Designing for Usability – Key Principles and What Designers Think*. Commun. ACM, Vol 28, 1985, pp.300-311.

[3] W. Swartout, R. Blazaer: *The Inevitable Intertwining of Specification and Implementation*. Commun ACM, Vol 25, No. 7, Jul 1982, pp.438-440.

[4] B. A. Myers, M. B. Rosson: Survey on User Interface Programming. In Striking a Balance. Proceedings CHI'92. Monterey, May 1992, New York: ACM Press, 1992, 195-202

[5] B. A. Myers: User Interface Software Tools. ACM Transactions on Computer-Human Interaction 2 (1995), 1, 64-103

[7] O. Pastor, E. Insfran, V. Pelechano, J. Romero, J. Meseguer. *OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods.* Procs. of 9th. International Conference, CAiSE'97. Barcelona, Catalonia, Spain, June 16-20, 1997.

[8] M.D. Lozano, I. Ramos, P. Cuenca. *"Application of an Object-Oriented Methodology for Automatic Software Production: OO-Method"*. Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'97). Pags. 985 – 989. Victoria, Canadá. 20-22 de Agosto, 1997.

[6] P. Letelier, I. Ramos, P. Sanchez, O. Pastor. *OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling.* SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[9] M.Lozano, I. Ramos: An Object Oriented Approach for the Specification of User Interfaces. PACRIM'99, Victoria -Canadá- Agosto, 1999.

[10] M. Lozano, I. Ramos: Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. ECOOP'99, Lisboa -Portugal-, Junio, 1999.

User Interface Generation: Current Trends

María Dolores Lozano, Pascual González and Isidro Ramos

Various studies have proved that 48% of application code is devoted to the user interface and that nearly 50% of the implementation time is devoted to implementing it. The more easy the interface is to use, the more difficult is its development. In fact, the user interface is becoming more and more important as a means of interacting with a system. For this reason, in this article we have tackled the subject of user interface generation within the research area of Human-Computer Interaction.

Keywords: User Interfaces, Object Oriented Specification and Modelling Techniques

1 Introduction

New technologies and the user's constant demand for information have made the design and development of user interfaces more and more important in the area of software engineering. In fact, all aspects of Human-Computer Interaction (HCI) are being considered as a research field of great relevance and perhaps the one which has the greatest impact on the final users of information systems. Several disciplines can be considered within this research area, which range from the definition of user interface analysis and design methods, through the creation of new methods of representing information, to the study of the human factors involved in the management of such interfaces. In this article we review the most interesting investigations made up to now in the field of user interface generation and definition methods. Finally a new proposal is suggested, a user interface development environment within the software development process conforming to the object oriented model proposed in OASIS [Letelier et al. 98].

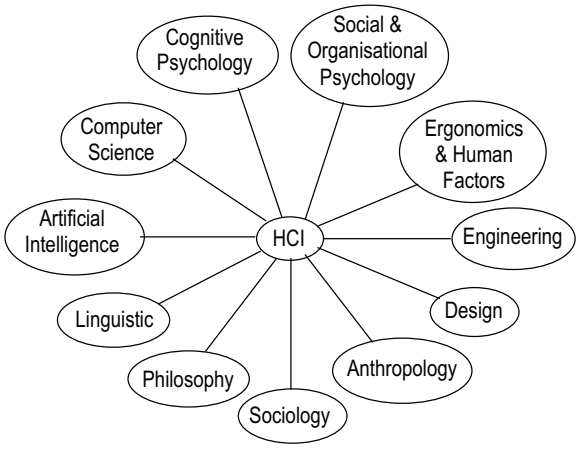


Fig. 1: HCI components.

2 Human Computer Interaction (HCI).

HCI is a discipline covering the design, evaluation and implementation of interactive systems for human use and entails the study of all the phenomena around them [ACM 92]. In the middle of the 1980's, the term HCI was adopted to reflect every factor involved in the interaction between users and computers.

As a matter of fact this term is becoming more and more important, mainly due to the "invasion" of computers in people's everyday activities. The computing evolution has contributed to this and has led to a greater preoccupation with user interface design when acknowledging that computer

María Dolores Lozano received her M.Sc. degree in Computer Engineering from the Technical University of Valencia. She is currently working on her Ph.D. degree in the area of user interface generation. In 1997 she joined the Department of Computer Engineering at the University of Castilla-La Mancha where she is an Assistant Professor for Software Engineering. Her research interests include human-computer interaction, user interface development, modelling techniques, object-orientation and specification languages. <mlozano@info-ab.uclm.es>

Pascual Gonzalez received the PhD. degree in Computer Engineering from the Technical University of Madrid. In 1991 he joined the Department of Computer Engineering at the University of Castilla-La Mancha where he is an Associate Professor for Software Engineering and director of the Information Technology Department in the Regional Development Institute. His research interests include human-computer interaction, user interface design, 3D interaction techniques, patterns, modeling techniques. <pgonzales@info-ab.uclm.es>

Isidro Ramos Salavert, is Full Professor in the Computation and Information Systems Dept. of the Technical University of Valencia, with 31 years of experience in different universities (Complutense de Madrid, Politécnica de Valencia, del País Vasco, Nancy). His research interests are: Object Oriented Conceptual Modelling, Foundations of Object Orientation. He has been a key researcher in 19 Research Projects, a member and president of the Programme Committee of relevant national and international conferences (CAISE, DEXA, EDBT…), and author of about 40 papers. He is presently responsible for International Coordination of the VII CYTED Subprogram on Applied Electronics and Computing. <iramos@dsic.upv.es>

Lozano, M. D., González, P., Ramos, I. User Interface Generation. Current Trends. INFORMATIK /INFORMATIQUE. Journal of the Swiss informatics societies. Vol. 2. 2001.

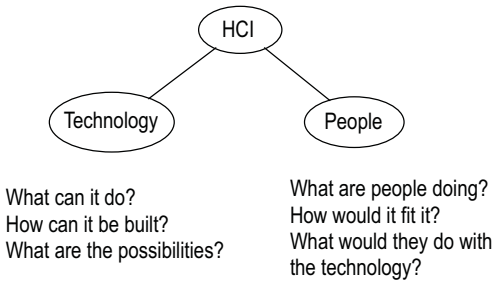


Fig. 2: HCI: Technological-Human Factors

programs should be designed to cover the user's needs and capabilities. That is, the user should not need to make radical changes to adapt himself to the system but the system must be designed to adjust to the user's needs.

Underlying HCI research is the belief that the people who use computer systems are the most important element. Their needs, capabilities and preferences to carry out their activities should dictate the design, implementation and use of the systems.

Human-Computer Interaction is a multidisciplinary science that not only includes computing or technological issues but also takes into account the other disciplines depicted in fig. 1.

The impact of engineering and design disciplines in HCI has been through Software Engineering due mainly to the fact that analysis and design methods, above all from the graphical point of view, have influenced the design of user interfaces.

The challenge of HCI is to acquire the maximum knowledge of technological capability and users needs so as to develop new technological artifacts that help and facilitate the human-machine interaction as depicted in figure 2.

3 User Interface Development Environments. State of the Art.

In this section we briefly describe the most outstanding features of current model-based user interface development environments. In a state of the art report, B. Myers presents a classification of user interface software tools [Myers 95]. This classification is based on the way that user interface developers can specify the layout and the dynamic behaviour of a user interface. This classification is as follows:

- *Language-based Tools:* These tools require the developer to program in a special purpose language.
- *Interactive Graphical Specification Tools:* These tools allow an interactive design of the user interface.
- *Model-based Generation Tools:* These tools use a high-level specification or model to generate the user interface automatically.

The last classification seems to be the most suitable to tackle user interface development. The first two support the specification of either the dynamic behaviour or the layout of the user interface in an easy way, but not both parts at the same time. In the same was as most current tools, they only support the last phase of the user interface life-cycle development, and the abstractions they provide do not have a direct connection with the task analysis results.

The typical components and the development process within a model-based user interface development environment are shown in figure 3.

The main component of these environments is the User Interface Model, which includes different declarative models such as the task, dialogue, domain, user and presentation models. Each one of them represents the different kinds of information needed for the final construction of the interface. Interactive and automatic development tools also have a role in this environment. The automatic generation tools are responsible for the transformations between the different declarative models and the final implementation of the desired user interface. The information required by the automatic generation and design tools is contained in the knowledge base.

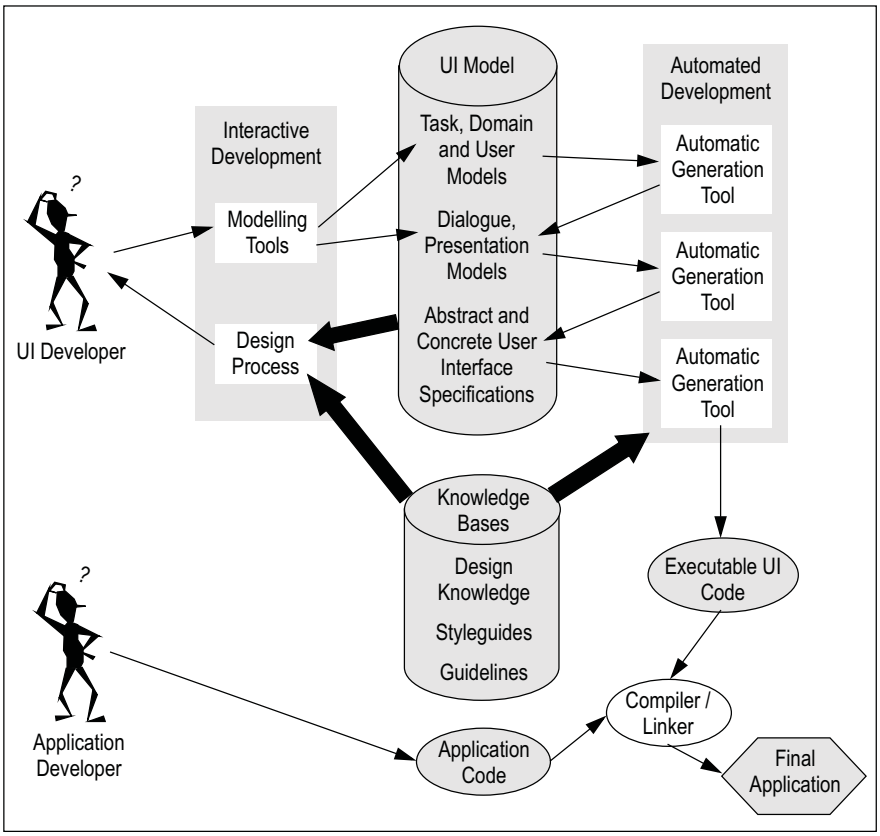


Fig. 3: Generic Architecture of a Model-based User Interface Development Environment.

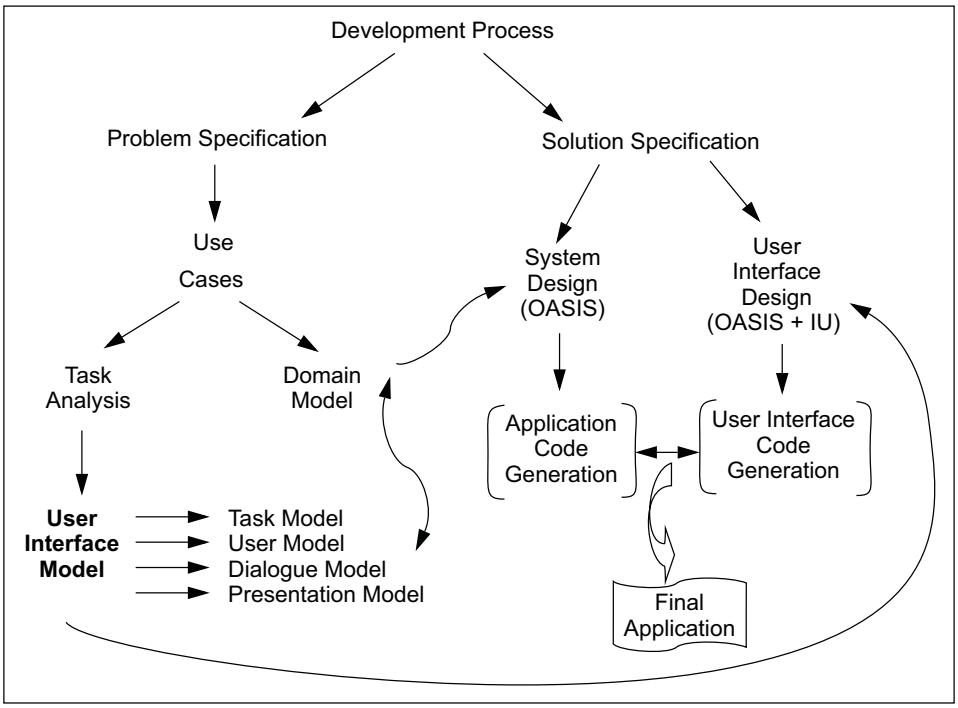


Fig. 4: IDEAS: Interface Development Environment within OASIS.

4 IDEAS: User Interface Generation within an Object Oriented Software Development Environment.

In this section we propose the integration of a user interface model within the software development environment defined by the object oriented model OASIS [Letelier et al. 98] from which the interface of an application can be generated. The user interface specification process is tackled in parallel to the application development. This development process is depicted in figure 4.

IDEAS [Lozano/Ramos 99b], [Lozano et al. 00] aims to be a model-based and automatic user interface development system integrated within the framework of OASIS to support the automatic production of high-quality user interfaces. This is possible due to the fact that this environment is based on declarative models, with the advantages of this approach as stated in the previous section.

Next, we describe this development process by levels and following the order in which the different models of our proposal are performed. This development process is depicted in figure 5.

At requirements level, the use cases [Jacobson et al. 92] technique is used. In this first stage and taking into account the use cases, the different kinds of users are identified. Subsequently, each use case is refined and the business rules, entities and actors that participate in it are specified.

At analysis level three models are generated. Firstly the Task Model, identifying the task the user has to perform with the system. This is done starting from the previous definition of use cases.

Following this, the Domain Model is performed. This model consists of two diagrams. The first one is the Sequence Diagram, which defines the system behaviour. The second one is the Roles Model, which defines the structure of the classes that take part in the associated sequence diagram together with the relationships among these classes, specifying the role of each one of them. Once the task and domain models have been defined, the user model can be defined. For every kind of user, the subset of tasks the user is allowed to perform is established. And for every task, a projection of the actions within the task that the user is allowed to accomplish is established. Finally, and according to the user's particular characteristics, the most appropriate way to show the information is defined.

At design level, the Dialogue Model is performed [Lozano/Ramos 99a]. All the models that have been generated up to now do not contain any graphical aspect of the final user interface. It is from now on that these issues start to be addressed and the way in which the user-system interaction will be performed is especially important. The dialogue model includes the generation of two different kinds of diagrams, the Dialogue Structure Diagram and the Component Specification Diagram. The first diagram specifies the user interface behaviour, this is, it represents the windows and dialogues that the user needs to complete all the tasks he requires from the system, and the user selections to pass from one window to another. For the generation of this diagram, we have to take into account the sequence diagram obtained in the analysis phase. An example of this type of diagram is shown in figure 6.

The second diagram entails establishing, for every one of the windows and dialogues obtained in the first diagram, the set of

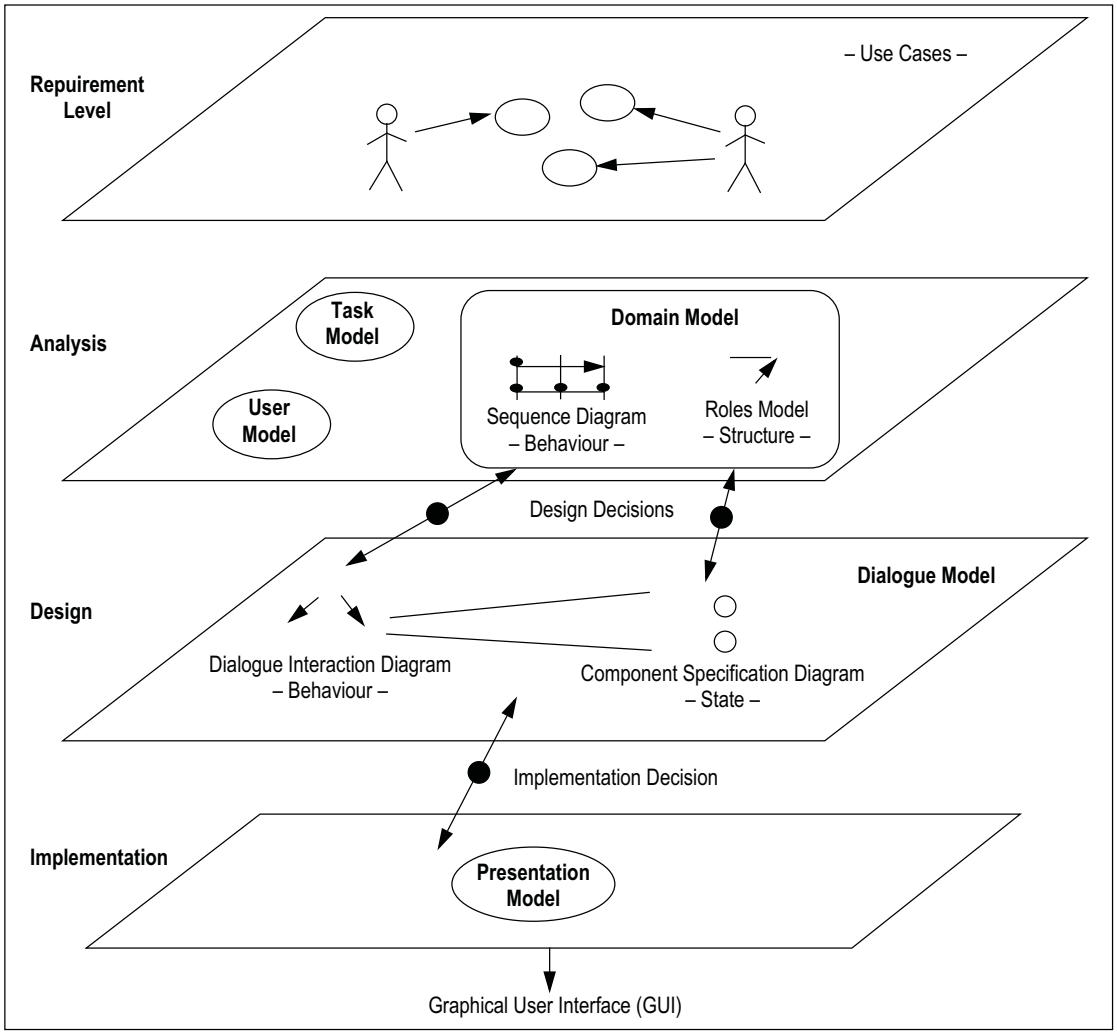


Fig. 5: User Interface Development Process.

control and visualization tools needed to give the user the functionality he requires to perform the corresponding task. This diagram defines the state of the user interface as depicted in figure 7:

At implementation level the Presentation Model is performed. In this model, the actual interaction objects comprising the Graphical User Interface are defined taking into account implementation decisions, the final implementation platform and according to the style guide followed.

5 Conclusions

The user interface is becoming more and more important as a means of interacting with a system. For this reason, in this article we have tackled the subject of user interface generation within the research area of Human-Computer Interaction.

With the aim of achieving an object oriented environment for automatic software production, that not only can be used with the efficiency and simplicity that the end-user requires through a friendly but at the same time robust interface, we have presented IDEAS, a user interface development environment with the philosophy of OASIS.

IDEAS, based on declarative models, provides descriptive information of the future components of the interface at a high level of abstraction. In the early development phases it also allows the study of how the interaction between the user and the system is performed. In the last phases it allows the automatic code generation based on the models.

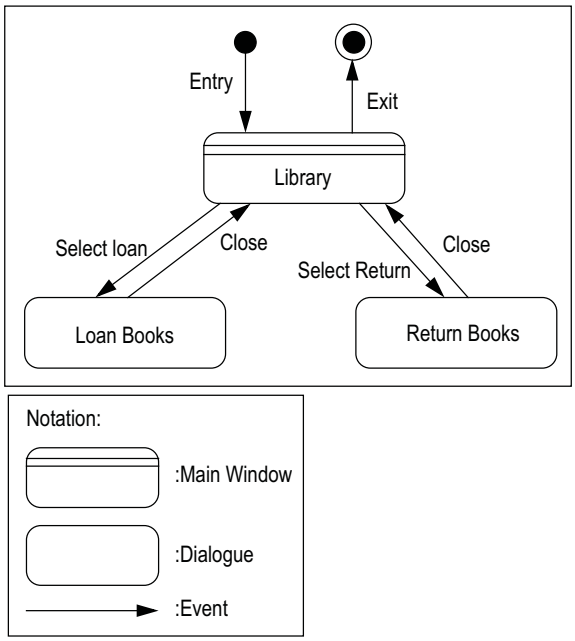


Fig. 6: Dialogue Interaction Diagram.

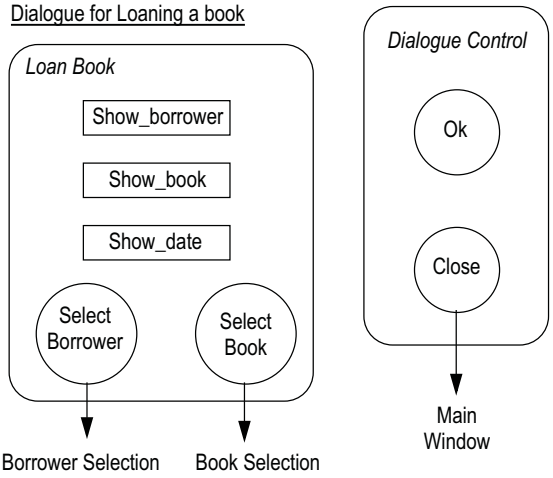


Fig. 7: Component Specification Diagram.

References

[ACM 92]
ACM Special Interest Group on Computer-Human Interaction. Curricula for Human-Computer Interaction. 1992.

[Jacobson et al. 92]
I.Jacobson, M. Christerson, et al.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

[Letelier et al. 98]
P. Letelier, I. Ramos, P. Sanchez, O. Pastor. OASIS version 3.0: A Formal Approach for Object Oriented Conceptual Modelling. SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[Lozano/Ramos 99a]
M.Lozano, I. Ramos: An Object Oriented Approach for the Specification of User Interfaces. PACRIM'99, Victoria -Canad, August, 1999.

[Lozano/Ramos 99b]
M. Lozano, I. Ramos: Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. ECOOP'99, Lisboa -Portugal, June, 1999.

[Lozano et al. 00]
M.Lozano, I. Ramos, P. González. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development. 34th International Conference on Technology of Objetct-Oriented Languages and Systems. TOOLS-USA 2000. Santa Barbara, CA. 30 July – 3 August, 2000

[Myers 95]
B. A. Myers: User Interface Software Tools. ACM Transactions on Computer-Human Interaction 2 (1995), 1, 64–103

IDEAS: An Object Oriented Environment for Developing User Interfaces

María D. Lozano, Pascual González
Dpto. Informática
Universidad de Castilla-La Mancha
Albacete, SPAIN
Tel: +34 967 59 92 00
[mlozano, pgonzalez]@info-ab.uclm.es

Francisco Montero
Dpto. Tecnología de Computadores
Univ. Polit. de Cartagena
Cartagena, SPAIN
Tel: +34 968 32 64 62
fmontero@upct.es

Isidro Ramos
Dpto. Sist. Informáticos. y Computación
Univ. Polit. de Valencia
Valencia –SPAIN–
Tfno: +34 96 387 9359
iramos@dsic.upv.es

ABSTRACT
The main objective of information systems development is obtaining a system that allows final users to reach their goals and perform the tasks needed to achieve the goals in an efficient and effective way. To achieve this aim it is necessary, not only taking into account system requirements, but also incorporate new techniques that help to capture user needs considering usability criteria. In this way, it is possible to develop intuitive and easy-to-use user interfaces that help the final users to derive the maximum benefit from computer applications. With this aim, a new user interface development environment within the object-oriented software development process is proposed in this paper.

KEYWORDS: User Interface, Object-Oriented Modeling Techniques, Human Computer Interaction, Usability.

INTRODUCTION
Most of the research developed by industry and academy has led to several object-oriented methods highly adequate for the development of systems. But most of these methods do not include mechanisms (models) for identifying and specifying user needs and requirements as well as testing and validating requirements with end-users before, during and after development. These considerations are especially important in interactive systems, where the user interaction is very high and user interface is a significant part of the system.

As a result of this weakness, interactive systems developed using such methods can meet all technical requirements, be very robust, and yet be unusable by the end-user. This problem explains a large part of the frequently observed phenomenon whereby large numbers of change requests to modify the services of an application are made after its deployment. As a solution to this problem we propose a method for integrating a user interface model, taking into account user needs and requirements, in the software life cycle. Besides, this approach to the development process ensures the quality of delivered applications from the point of view of the end user.

There exists several reasons that justify the complexity of design and implementation of user interfaces [1], but perhaps the main reason is the difficulty of understanding the tasks the user must perform with the system and the characteristics of the different kind of users managing it [2]. In this sense, it seems necessary to define new models that allow collecting user requirements and specifying the different characteristics of the interface associated to a concrete application. These models should allow defining the specification of the interface independently [3], that is, the models used to specify the user interface should be different from those used to specify the application.

In this paper we present IDEAS[1] an object oriented and model-based approach for developing user interfaces within an automatic software production environment. This environment is supported by an object-oriented model called OASIS[2] [4], [5], which covers the software life cycle of analysis, design and implementation within the principles of the object oriented paradigm. So, for better understand our proposal, we first present the state-of-the-art of model-based user interface development environments, and then we finally present IDEAS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference '00, Month 1-2, 2000, City, State.
Copyright 2000 ACM 1-58113-000-0/00/0000…$5.00.

[1] **I**nterface **D**evelopment **E**nvironment within o**AS**is.

[2] **O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems

Lozano, M. D., González, P., Ramos, I. IDEAS: An Object-Oriented Environment for developing User Interfaces". Proc. of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01). 2001.

MODEL-BASED USER INTERFACE DEVELOPMENT ENVIRONMENTS. STATE OF THE ART

In this section we briefly describe the most understanding features of current model-based user interface development environments. In a state of the art report, B. Myers presents a classification of user interface software tools [6]. This classification is based on the way the user interfaces developers can specify the layout and dynamic behavior of a user interface. This classification is the following:

- Language-based Tools: These tools require the developer to program in a special purpose language.
- Interactive Graphical Specification Tools: These tools allow an interactive design of the user interface.
- Model-based Generation Tools: These tools use a high-level specification or model to generate the user interface automatically.

The last classification seems to be the most adequate to tackle the user interface development. The first two support the specification of either the dynamic behavior or the layout of the user interface in an easy way, but not both parts at the same time. And the same as most current tools, they only support the last phase of the user interface life-cycle development, and the abstractions they provide do not have a direct connection with the task analysis results.

On the other hand, model-based user interface development approach and its corresponding tools is an emerging technology that remedies these shortcomings. Interfaces are developed by using specialized design-time tools to build and refine models. Developers specify "what" features the interface should have, rather than write programs that specify "how" to make the computer exhibit the desired behavior.

The main shortcoming of the model-based approach is that building models is difficult. A rich model describing all the features of an interface is complex and non-trivial to specify.

The most representative feature of the model-based approach is that all aspects of the user interface design are represented using declarative models. This paradigm offers several benefits:

- User-centered development life cycle.
- Centralized user interface specification.
- Design tools for an automated and interactive development.
- Reuse of the user interface designs.

For a user interface tool to be considered as a Model-based user interface development environment, it must accomplish the following criteria:

1. It must include a high-level, abstract and explicitly represented (declarative) model about the system under development.
2. It must clearly establish a technologically well-supported relation between 1. and the final running user interface.

Currently there exist several model-based user interface software tools, as for instance, UIDE [7], HUMANOID [8], MOBI-D [9], MASTERMIND [10], [11], TRIDENT [12], FUSE [13], JANUS [14] and IDEAS our proposed model explained in following sections.

A comparison of IDEAS with different model-based interface development environments is shown in the following table.

	Task	Domain	User	Dialogue	Presentation
UIDE		✖ C++ Classes			
ADEPT	✖		✖		
MASTERMIND	✖	✖ CORBA IDL			✖
MECANO		✖			
MOBI-D	✖	✖ MIMIC Object Model	✖	✖	✖
TRIDENT	✖	✖ E-R Model			
AME		✖ OO Model			
GENIUS		✖ E-R Model		✖	
FUSE	✖	✖ Algebraic Specification	✖	✖	
JANUS		✖ OO Model			
TADEUS	✖	✖ OO Model	✖	✖	
IDEAS	✖	✖ OO Model	✖	✖	✖

Table 1. Model-Based Interface Development Environments and their corresponding declarative models.

In the next figure we show the typical components and the development process of this model-based approach to better understand it

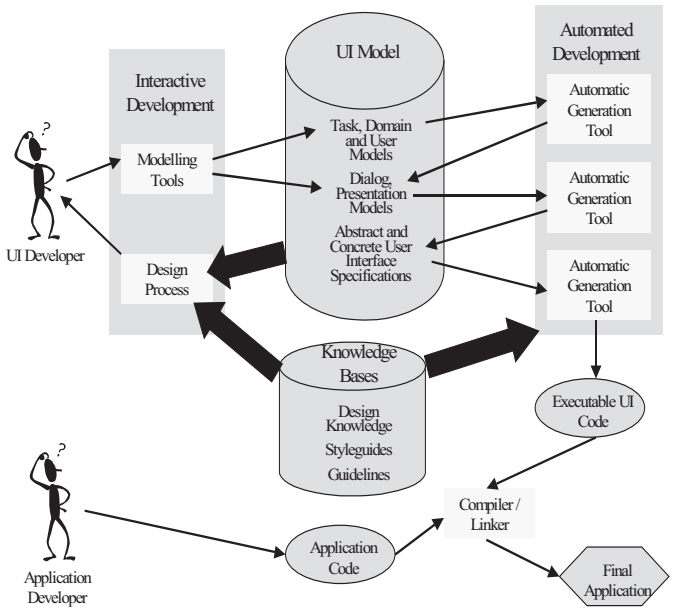


Figure 1. Generic Architecture of a Model-based User Interface Development Environment.

The main component of these environments is the User Interface Model, which includes different declarative models such as the task, dialogue, domain, user and presentation models. Each one of them represents the different kinds of information needed for the final construction of the interface.

Interactive and automatic development tools also have a role in this environment. The automatic generation tools are responsible for the transformations between the different declarative models and the final implementation of the desired user interface. The information required by the automatic generation and design tools is contained in the knowledge base.

IDEAS: INTERFACE DEVELOPMENT ENVIRONMENT WITHIN OASIS

The aim of this research is the integration of a user interface Model within an object oriented software production environment. The user interface specification process is tackled in parallel to the application development, and according to the common principles of Model-based User Interface Development Environments, as exposed in previous section. This development process is depicted in the following figure:

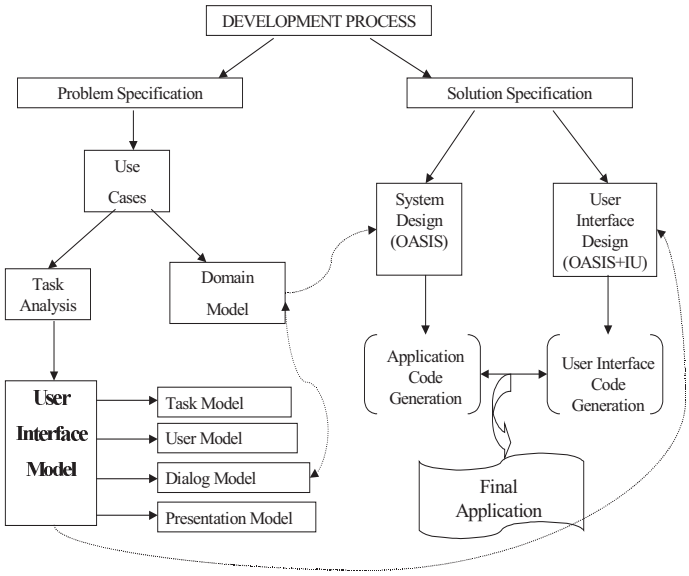


Figure 2. IDEAS: Interface Development Environment within OASIS.

IDEAS aims to be an automatic user interface development system integrated within the framework of the object oriented model OASIS [5] to support the automatic production of high-quality user interfaces. This is possible due to the fact that this environment is based on declarative models, with the advantages of this approach as stated in the previous section.

As depicted in figure 2, the development process is split into two pieces: a specification of the problem (what must the user accomplish with the aid of the system) and a specification of the solution (how will this solution be enabled).

Object-oriented development practice often splits the problem specification into two pieces. The first piece is a set of use cases providing the requirements of the domain activity to be supported. The second piece is an analysis model (domain model), the conceptual object model that describes the objects identified within the system together with the relations among them.

But use cases are not enough to describe user tasks from the point of view of the final user. Use cases describe the systems in terms of functional requirements, nevertheless, one of their main deficiencies is that they do not capture non-functional requirements, such as reliability, efficiency, maintainability and above all, usability, which undoubtedly

are critical factors for final users to accept a computer application. Use cases do not capture user needs either, which is essential to develop user interfaces. For this reason, task analysis techniques are included in this point. Starting from task analysis, the User Interface Model is built.

All in all, the approach we propose is to take a use-case-driven development process adding task analysis techniques to enable the specification of the user interface. These techniques mainly help to define who the users of the system are, what tasks should they perform, what the main goal of those tasks is, and son on.

The solution specification can also be broken into two pieces. The first piece is the design of the user interface. The second piece is the design of the system supporting the solution. In many ways these two designs specify two separate but interdependent systems.

In IDEAS, the declarative models composing the user interface model are the following: The Task, User, Domain, Dialog and Presentation Models. Next, we define them.

The Task Model defines the ordered set of activities and actions the user has to perform to achieve a concrete purpose or goal. Artifacts are essential for task development. They are modeled as objects and represented in the domain model. There exist a strong relation between the task model and the domain model.

According to [15], a task description should include a goal, a non-empty set of actions necessary to achieve the goal, a plan for the selection of actions and a model of the artifact involved in the task development.

This description of what a task is and must contain can be used to construct a task definition language. We propose a template (fig. 8), based on the one proposed by [16], to describe in natural language all these issues.

The question now is what tasks should be modeled in the task model. The answer is all those tasks the final users have to perform with the system.

The User Model describes the characteristics of the different types of users. The purpose of this model is to support the creation of individual and personalised user interfaces. Firstly, it defines, for each kind of user, the set of tasks he/she can perform. Secondly, for each kind of user in relation with a concrete task, a projection on the actions within the task that he/she can perform is established. And finally, depending on the user's particular characteristics (child, adult, handicapped...), the information and the interaction established by the Dialog Model to show the information contained in the Domain Model is adapted to the user.

The Dialog Model [17] is used to describe the user-computer "conversation". The purpose of this model is to describe the syntactic structure of the user-system interaction. It establishes when the user can invoke commands, select or specify the input data and when the computer can require data from the user or display the output data. This model includes the construction of three different kinds of diagrams representing these matters, the Dialog Structure Diagram, the Component Specification Diagram and the Component Behavior Definition Table [18].

To illustrate the construction of these diagrams, we show and example built with the software tool we are currently constructing to implement the IDEAS environment:

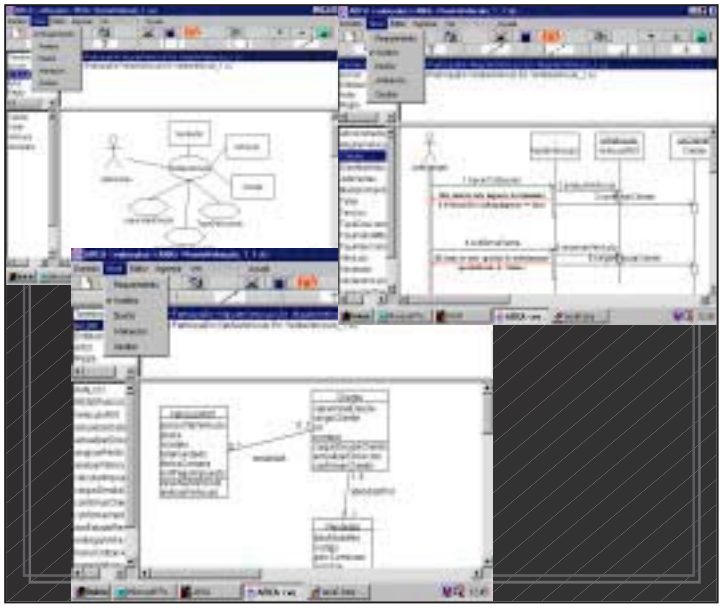


Figure 3. Software Tool for implementing IDEAS.

Finally, the Presentation Model describes the concrete interaction objects composing the final GUI, its design characteristics and visual dependencies among them. The final GUI has a static and a dynamic part. The first one is the standard widgets presentation and the second one shows the data dependent on the application that change at run time.

This model leads to the visualization of the final GUI according to the final platform style guides. The final GUI generation is performed by using XML based languages, such as XUL [19] or UIML [20] in order to make it as much independent as possible from the final platform

where the application is going to run and so get the maximum compatibility with new web technologies.

La generación de la interfaz se plantea realizarla en el lenguaje de marcado XML para hacerla lo más independiente posible de la plataforma donde se vaya a ejecutar la aplicación y conseguir la máxima compatibilidad con las nuevas tecnologías web.

DEVELOPMENT PROCESS

The user interface development process proposed in IDEAS is depicted in figure 4.

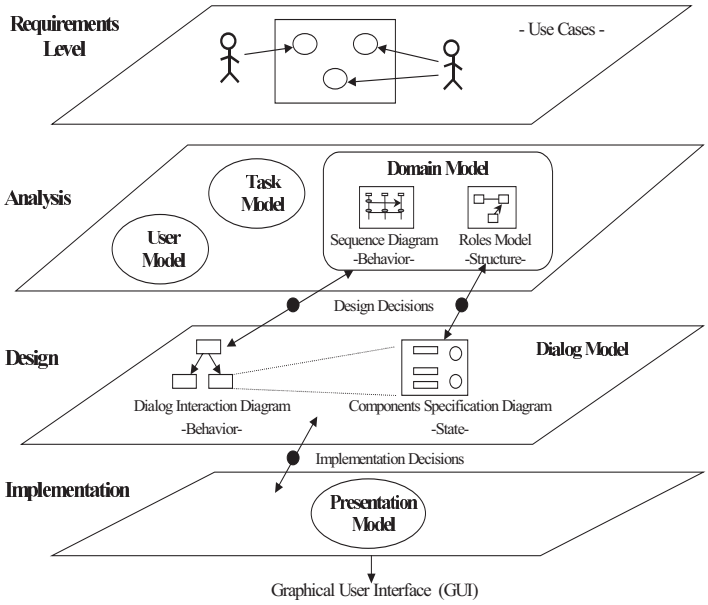


Figure 4: User Interface Development Process.

At requirements level, the use cases technique [UML] is used. In this first stage and taking into account the use cases, the different kinds of users are identified. Subsequently, each use case is refined and the business rules, entities and actors that participate in it are specified.

At analysis level three models are generated. Firstly the Task Model, identifying the task the user has to perform with the system. This is done starting from the previous definition of use cases.

Following this, the Domain Model is developed. This model consists of two diagrams. The first one is the Sequence Diagram, which defines the system behavior. The second one is the Roles Model, which defines the structure of the classes that take part in the associated sequence diagram together with the relationship among these classes, specifying the role of each one of them.

Once the task and domain models have been defined, the user model can be defined. For every kind of user, the subset of tasks the user is allowed to perform is established. And for every task, a projection of the actions within the task that the user is allowed to accomplish is established. Finally, and according to the user's particular characteristics, the most appropriate way to show the information is defined.

At design level, the Dialogue Model is developed [17]. All the models that have been generated up to now do not contain any graphical aspect of the final user interface. It is from now on that these issues start to be addressed and the way in which the user-system interaction will be performed is especially important. The dialogue model includes the generation of two different kinds of diagrams, the Dialogue Structure Diagram and the Component Specification Diagram. The first diagram specifies the user interface behavior, this is, it represents the windows and dialogues that the user needs to complete all the tasks he/she requires from the system, and the user selections to pass from one window to another. For the generation of this diagram, we have to take into account the sequence diagram obtained in the analysis phase. The second diagram entails establishing, for every one of the windows and dialogues obtained in the first diagram, the set of control and visualization tools needed to give the user the functionality he requires performing the corresponding task. This diagram defines the state of the user interface.

At implementation level the Presentation Model is created. In this model, the concrete interaction objects comprising the Graphical User Interface are defined taking into account implementation decisions, the final implementation platform and according to the style guide followed.

This development process considers the three-level software architecture: boundary, control and entity as proposed in the Unified Software Development Process [21], but the scope of our proposal is to specify the boundary classes corresponding with the final graphical user interface. As depicted in figure 5, users interact with these boundary classes (user interface classes) to perform their tasks. The boundary classes will be associated to the corresponding control class/es to achieve the task goal, and the entity classes correspond to long-lived and persistent information. The specification of boundary classes is the goal of user interface design whereas the specification of control and entity classes is the goal of domain analysis model. So, we center on boundary classes to represent the application user interface.

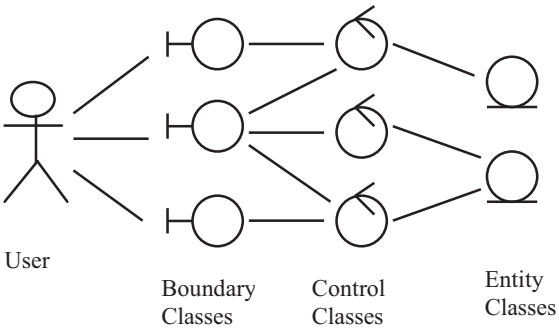


Figure 5: Three-level software architecture.

USABILITY IN USER INTERFACE DEVELOPMENT

The main aim of software construction is to help users to achieve their goals in an easy way. That is, make the system to be easy to learn, easy to manage and useful, i.e. it should contain all the functionality the users need to accomplish their tasks. All in all, the basic goal is to get a computer application with a high grade of "usability" [2].

Computer systems built with usability criteria have several benefits such as productivity improvement, cost and learning time reduction and a notable increment of final user autonomy.

Landauer [22] stated that 80% of maintenance costs – what represents the 80% of software development total costs – are due to user-system interaction problems and not to technical problems. Besides, he indicates that 64% of these costs are related with usability problems.

This situation highlights the importance of usability and justifies the need to incorporate usability criteria before, during and after software systems development. This means that computer application development is a multidisciplinary activity that must incorporate human factors and ergonomic techniques with the aim of improving the work conditions of final users. This implies the participation of different kind of experts, not only computer engineers but also experts on disciplines such as psychology, ergonomics and human factors and so on.

The ISO definition of usability

The ISO 9241 standard [23] defines usability as the "extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use".

- effectiveness: The "accuracy and completeness with which users achieve specified goals".
- efficiency: The "resources expended in relation to the accuracy and completeness with which users achieve goals".
- satisfaction: The "freedom from discomfort, and positive attitude to the use of the product".

The components and the relationships between them are illustrated in figure 6.

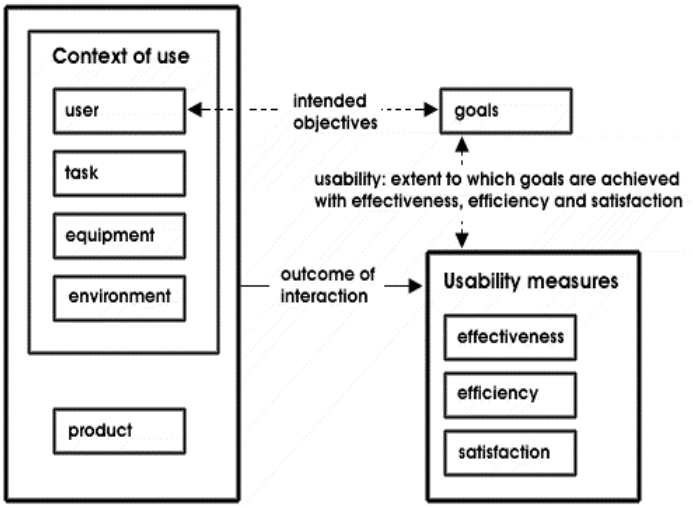


Figure 6: Usability framework according to ISO/DIS 924111.2.

The standard states that when specifying or measuring usability, the following information is needed:

- A description of the intended goals.
- A description of the components of the context of use including users, tasks, equipment and environments. This may be a description of an existing context, or a specification of intended contexts. The relevant aspects of the context and the level of detail required will depend on the scope of the issues being addressed. The description of the context needs to be sufficiently detailed so that those aspects of the context which may have a significant influence on usability could be reproduced.
- Target or actual values of effectiveness, efficiency, and satisfaction for the intended contexts.

Usability context of use

The context of use defined by the standard includes the following factors [24]:

Description of users. Characteristics of the users need to be described. These can include knowledge, skill, experience, education, training, physical attributes, and motor and sensory capabilities. It may be necessary to define the characteristics of different types of user, for example users having different levels of experience or performing different roles.

Description of tasks. Tasks are the activities undertaken to achieve a goal. Characteristics of tasks which may influence usability should be described, e.g. the frequency and the duration of the task.

Detailed descriptions of the activities and processes may be required if the description of the context is to be used as a basis for the design or valuation of details of interaction with the product. This may include description of the allocation of activities and steps between the human and technological resources. Tasks should not be described solely in terms of the functions or features provided by a product or system. Any description of the activities and steps involved in performing the task should be related to the goals which are to be achieved.

Description of equipment. The description of the hardware, software and materials may be in terms of a set of products, one or more of which may be the focus of usability specification or evaluation, or it maybe in terms of a set of attributes or performance characteristics of the hardware, software and other materials.

Description of environments. Relevant characteristics of the physical and social environment need to be described. Aspects which may need to be described include attributes of the wider technical environment (e.g. the local area network), the physical environment (e.g. workplace, furniture), the ambient environment (e.g. temperature, humidity) and the social and cultural environment (e.g. work practices, organizational structure and attitudes).

Usability measures. Usability measures include effectiveness, efficiency and satisfaction. These are measured in user trials of the product.

Goals of trial identify what the projects are trying to find out in their trials. The goal of the user trial may be to help in defining user requirements, to validate that the technology works in real conditions, to measure user attitudes, etc.

All in all, usability means that the people who use the product can do so quickly and easily to accomplish their own tasks. [Dum93]. This definition rests in four important points:

- Usability means focusing on users.
- People use products to be productive.
- Users are busy people trying to accomplish tasks.
- Users decide when a product is easy to use.

Usability means focusing on users. To develop a usable product, you have to know, understand and work with people who represent the actual or potential users of the product.

People use products to be productive. People consider a product "easy to learn and use" in terms of the time it takes to do what they want, the number of steps they go through, and the success they have in predicting the right action to take. To develop usable products, therefore, you have to understand user's performance goals.

Users are busy people trying to accomplish tasks. People connect usability with productivity, because no one get paid for time spend just sitting at a computer. They get paid for processing invoices, or for noticing and resolving alarms on the computer network, or for analysing samples in a laboratory machine. Hardware and Software are tools to help busy people do the work.

Users are concerned with productivity and with accomplishing their own goals at home as well as at work. You may want to use your VCR to record a program so that you can watch it later. Your goal is to watch the program; the VCR is the tool you are using to achieve that goal.

Many VCRs fail a test of usability for this function. Users cannot program their VCRs in the time and effort that they are willing to spend on this task. The VCR may have high functionality (the feature works as it was designed to work), but they have low usability (people cannot use them quickly and easily to accomplish their tasks).

Users decide when a product is easy to use. Users not designers or developers, determine when a product is easy to use.

How many functions of your word processor, spreadsheet, voice mail, or microwave are you not using that you might use if they were easy enough to learn, use, and remember next time you need them?.

B. Shneiderman [25] explains that, within the "usability engineering", one of the basis to reach a high quality level is using iterative design methods. These methods permit developers to use prototypes, getting feedback from user reactions.

J. Nielsen [26] states that the best way to successfully applying usability criteria is putting the maximum effort before the design process of the interface layout starts. With this concern, the most important thing is to make a previous study which allows the designer to "know the users". This knowledge is based not only on the study of users' characteristics or individual abilities, but also it is necessary to know the tasks they need to carry out. In this sense, it is very important that users could perform evaluations (usability tests) upon the prototype. With this

practice we can obtain improvements in next versions of the product.

USABILITY IN IDEAS

For the reasons stated before, our proposal includes high-level models which permit "know the users". For this matter, IDEAS includes use case, task and user models. The first ones permit to describe user tasks accurately. The last one describes the individual characteristics of the different kind of users interacting with the system.

After knowing the users and the tasks they perform, it is time to set the goals or basic usability criteria the system must reach. The definition of these criteria will allow performing the usability test associated to such criteria further on.

In last phases of the development, when designing the presentation model, the designer must take into account different heuristics and style guides to improve the usability of the layouts. Starting from the presentation models, the user interface prototype is generated. Final users evaluate these prototypes with the usability tests prepared previously.

Finally, all the previously stated phases and models are incorporated into an iterative design process. This way of doing permits to perform a feedback process by introducing the information collected in the tests performed by the users in the user interface development. This way, the usability of the designed interfaces improves notably.

In figure 7, this iterative process integrating usability is depicted.

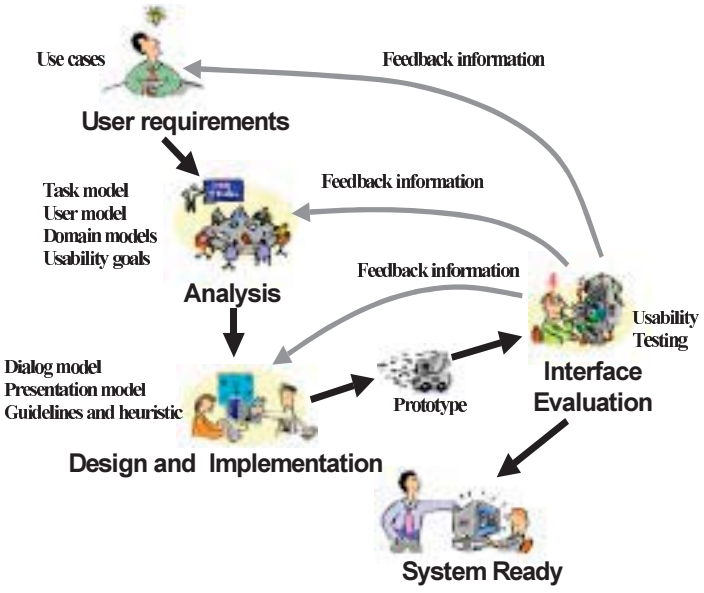


Figure 7: Iterative development process.

CONCLUSIONS

In this paper we propose a user interface development environment within the object-oriented software development process. This development process is performed applying use cases and task analysis techniques, from which a user interface model is built. This model consists of five declarative models that describe different aspects involved in user interface construction.

It has been also described the way to incorporate usability criteria in order to achieve high usable user interfaces. Finally, we have stated an iterative design process which permits the incorporation of the consecutive improvements the users introduce in the design process.

And finally, remark the more and more importance that is getting the user interface, above all in interactive systems, as a means of interacting with a system and taking the most profit of it. This is the main motivation of this research.

REFERENCES

1. B.A. Myers: Why are Human-Computer Interfaces Difficult to Design and Implement. Tech. Report CMU-CS-93-183, CMU, Jul 1993.

2. J. Gould, C. Lewis: Designing for Usability – Key Principles and What Designers Think. Commun. ACM, Vol 28, 1985, pp.300-311.

3. W. Swartout, R. Blazaer: The Inevitable Intertwining of Specification and Implementation. Commun ACM, Vol 25, No. 7, Jul 1982, pp.438-440.

4. O. Pastor, I. Ramos. OASIS 2.1.1.: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach. Octubre 95 (3 ed.)

. P. Letelier, I. Ramos, P. Sanchez, O. Pastor. OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling. SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

. B. A. Myers: User Interface Software Tools. ACM Transactions on Computer-Human Interaction 2 (1995), 1, 64-103

. J. Foley, P. Sukaviriya: History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In F. Paterno (ed.): Interactive Systems: Design, Specification and Verification. Berlín: Springer, 1995, 3-14.

. P. Szekely et al: Beyond Interface Builders: Model-Based Interface Tools. In: Bridges between Worlds.

Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 383-390.

9. A. Puerta: The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 19-36.

10. R. Neches et al.: Knowledgeable Development Environments Using Shared Design Models. In: Proceedings of the 1993 International Workshop on Intelligent User Interfaces (Orlando, Enero 1993). New York: ACM Press, 1993, 63-70.

11. P. Szekely, et al.: Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach. In: Engineering for Human-Computer Interaction. Proceedings of the IFIP Conference on Engineering for Human-Computer Interaction (Yellowstone Park, Agosto 1995). London: Chapman &Hall, 1996, 120-150

12. F. Bodart et al.: Towards a Systematic Building of Software Architectures: teh TRIDENT Methodological Guide. In: Design, Specification and Verification of Interactive Systems. Wien: Springer, 1995, 262-278.

13. F. Lonczewski: The FUSE System: An Integrated User Interface Design Environment. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 37-56.

14. H. Balzert et al.: The JANUS application Development Environment Generating More than the User Interface. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 183-205.

15. G. Storrs: The Notion of Task in Human-Computer Interaction. In: People and Computers X. Proceeding British HCI'95 (Huddersfield UK, Agosto 1995). Cambridge: Cambridge University Press, 1995, 357-365.

16. A. Cockburn: Structuring Use Cases with Goals. Journal of Object-Oriented Programming, 10(5) 1997. http://members.aol.com/acockburn/papers/usecases.htm

17. M. Lozano, I. Ramos: Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. ECOOP'99, Lisboa -Portugal-, Junio, 1999.

18. M.Lozano, I. Ramos, P. González. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development. 34th International Conference on Technology of Objetct-Oriented Languages and Systems. TOOLS-USA 2000. Santa Barbara, CA. 30 Julio - 3 Agosto, 2000.

19. http://www.xulplanet.com/tutorials/xultu

20. http://www.uiml.org

21. I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process. Addison Wesley. 1998

22. Landauer, T.K. "The Trouble with Computers: Usefulness, Usability and Productivity. MIT Press, 1995.

23. ISO/DIS 9241: Ergonomic Requirements for Office Work with Visual Display Terminals (VDT's). Geneva: International Standards Organization. 1998.

24. Concejero, P. Clarke, A., Ramos, R. "ACTS Usability Evaluation Guideline" Report of USINACTS Usability in Acts. June, 1999.

25. B. Shneiderman. Designing the User Interface. Strategies for Effective Human-Computer Interaction. Addison Wesley, 1998.

26. J. Nielsen. Usability Engineering. Morgan Kaufmann, 1993.

Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso

María Dolores Lozano[†], Pascual González[†], Isidro Ramos[‡], Francisco Montero[†], Jose Pascual Molina[†]

[†]Departamento de Informática
Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha
-ALBACETE-
[mlozano, pgonzalez]@info-ab.uclm.es

[‡]Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
-VALENCIA-
iramos@dsic.upv.es

Resumen

El objetivo de todo desarrollo de sistemas de información es obtener un sistema que permita a los usuarios finales alcanzar sus objetivos y llevar a cabo, de forma efectiva y eficiente, las tareas necesarias para conseguirlos. Para lograr esto se hace necesario, no sólo tener en cuenta los requisitos del sistema, sino además, incorporar nuevas técnicas que ayuden a captar las necesidades del usuario incorporando criterios de usabilidad, para conseguir además desarrollar interfaces de usuario intuitivas y fáciles de usar que ayuden al usuario final a sacar el máximo provecho de los sistemas informáticos. Con este objetivo en este trabajo se propone un entorno de desarrollo de interfaces de usuario partiendo de técnicas de casos de uso y análisis de tareas.

Palabras clave: Interfaces de Usuario, Interacción Hombre-Máquina, Orientación a Objetos, Técnicas de Especificación y Modelado, Sistemas Interactivos.

1. Introducción

El principal objetivo a la hora de construir una aplicación informática es facilitar a los usuarios finales alcanzar sus objetivos y llevar a cabo sus tareas con ese sistema de forma efectiva y eficiente. Por tanto, en este proceso se hace necesario, no sólo tener en cuenta los requisitos del sistema, que nos llevarían a la obtención de un buen sistema, robusto y completo desde el punto de vista informático, pero que tal vez el usuario final no fuera capaz de utilizar de forma correcta al no haber tenido en cuenta sus necesidades concretas como usuario final, sino que se hace necesario incorporar en este proceso nuevas técnicas que ayuden a captar las necesidades del usuario incorporando criterios de usabilidad, para conseguir además desarrollar interfaces de usuario intuitivas y fáciles de usar que ayuden al usuario final a sacar el máximo provecho de los sistemas informáticos.

Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.16 (2002), pp. 83-91
ISSN: 1137-3601. © AEPIA (http://www.aepia.dsic.upv.es/)

Lozano, M. D., González, P., Ramos, I., Montero, F., Molina, J. P. Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.16.2002.

Esta forma de construir los sistemas se conoce como diseño orientado al usuario. El principal objetivo de esta forma de desarrollar software es hacer los sistemas más usables y accesibles a los usuarios finales, siendo éstos la pieza principal en este proceso de desarrollo. De acuerdo a la ISO/DIS 13407 [ISO-98], el proceso de diseño orientado al usuario implica:

- Activa participación del usuario así como una clara comprensión del contexto de uso y de las tareas y requisitos del usuario final.
- Una apropiada distribución de funciones entre los usuarios y la tecnología, especificando qué funciones deben ser llevadas a cabo por los usuarios y cuales son de índole tecnológica.
- La iteración de las soluciones de diseño en las que la retroalimentación del usuario es esencial como fuente de información.
- Una perspectiva de diseño multidisciplinar que requiere de gran variedad de especialidades o disciplinas (desarrolladores de software, expertos en usabilidad, especialistas en ergonomía, etc).

Una de las aproximaciones más ampliamente utilizadas por muchos grupos de investigación en el área HCI (Human Computer Interaction) para llevar a cabo esto es el análisis de tareas. El propósito de esta técnica es desarrollar modelos genéricos y por tanto abstractos de las tareas del usuario final. Otra técnica utilizada con este fin son los casos de uso.

En este trabajo se propone un entorno de desarrollo de interfaces de usuario basado en modelos [Pue94b] dentro del proceso de desarrollo de software orientado a objetos, que satisfaga los objetivos comentados anteriormente. En este proceso de desarrollo se hace uso de técnicas de casos de uso y análisis de tareas y se tienen en cuenta criterios de usabilidad. Así, se propone un modelo de interfaz de usuario que modela la interacción del usuario con el sistema y proporciona una representación formal del diseño de interfaz. Este modelo incluye además la generación de diagramas gráficos para representar los diferentes aspectos de la IU y permite generar de forma automática la IU final a partir de los modelos definidos.

Para cubrir estos aspectos, el resto del artículo se estructura de la siguiente manera: en el segundo apartado se introduce brevemente el estado del arte para situar este trabajo dentro del contexto de otros trabajos similares; en el siguiente apartado se analizan las técnicas de casos de uso y análisis de tareas y se indica su uso en el entorno propuesto. A continuación se presenta brevemente OASIS[1] [Let98], el modelo orientado a objetos que establece el marco de trabajo donde se propone el entorno de desarrollo de interfaces de usuario, entorno que se describe en el apartado cuarto. Finalmente se introducen los criterios de usabilidad necesarios en este proceso de desarrollo de interfaces de usuario.

2. Estado del Arte

De entre las herramientas software de generación de interfaces de usuario basadas en modelos [Pue94b] construidas, podemos destacar las siguientes: UIDE [Fol95], ADEPT [Joh95], HUMANOID [Sze93], ITS [Wie90], MECANO [Pue94a], MOBI-D [Pue96], AME [Mär96], FUSE [Lon96], MASTERMIND [Sze96], TRIDENT [Bod95], [Bod96], GENIUS [Jan93], JANUS [Bal96], TADEUS [Elw95], IDEAS, modelo propuesto para OASIS.

A modo de resumen, en la siguiente tabla se indica el alcance de cada una de ellas:

	Casos de Uso	Tareas	Dominio	Usuario	Diálogo	Presentación
UIDE			✖ Clases C++			
AME			✖ Modelo OO			
JANUS			✖ Modelo OO			
GENIUS			✖ Modelo E-R		✖	
ADEPT		✖		✖		
MASTER MIND		✖	✖ CORBA IDL			✖
MECANO			✖ Modelo OO			
MOBI-D		✖	✖ Modelo Objetual MIMIC	✖	✖	✖
TRIDENT		✖	✖ Modelo E-R			
FUSE		✖	✖ Especif. Algebraica	✖	✖	
TADEUS		✖	✖ Modelo OO	✖	✖	
OVID		✖	✖ Modelo OO	✖	✖	✖
IDEAS	✖	✖	✖ Modelo OO	✖	✖	✖

Tabla1. Modelos declarativos y su aplicación en diferentes entornos de desarrollo de IUs

[1] ***O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems*

3. Casos de Uso y Análisis de Tareas

Los casos de uso son una técnica ampliamente utilizada en ingeniería de requisitos orientada a objetos ya que proporcionan un buen mecanismo para determinar los límites de un sistema así como para describir requisitos. Los casos de uso describen los sistemas desde el punto de vista del usuario en términos de los requisitos funcionales y por tanto se pueden considerar como el punto de partida para poder especificar las necesidades de los usuarios finales. Sin embargo, una de sus principales deficiencias es que no capturan requisitos no funcionales tales como fiabilidad, eficiencia, mantenibilidad, portabilidad y sobre todo usabilidad, que son sin duda factores críticos para la aceptación de un sistema por parte de los usuarios finales, y tampoco sirven para capturar las necesidades de los usuarios, aspecto esencial a la hora de desarrollar interfaces de usuario.

Los casos de uso fueron introducidos en el método OOSE (Object-Oriented Software Engineering) por Jacobson [Jac92] y servían principalmente para capturar la funcionalidad del sistema (requisitos funcionales) desde el punto de vista del usuario final y para asegurar la trazabilidad de los requisitos, es decir, servían de hilo conductor a lo largo del proceso de desarrollo del software.

Los casos de uso normalmente se emplean, y así se hace en este trabajo, en la fase de definición de requisitos donde su papel es identificar la forma en la que será usado el futuro sistema.

Por otra parte, el Análisis de Tareas es una técnica utilizada en el análisis de requisitos de la interfaz de usuario. El análisis de tareas proporciona información acerca de cómo los usuarios llevan normalmente a cabo su trabajo y sirve para identificar la funcionalidad que debería proporcionar la interfaz de usuario del futuro sistema.

Ambos son conceptos muchas veces confundidos, por lo que en este apartado se pretende clarificar y distinguir el uso que se hace en este trabajo de estos conceptos.

Las técnicas de análisis de casos de uso normalmente son utilizadas por ingenieros de software involucrados en el desarrollo de sistemas orientados a objetos, mientras que las técnicas de análisis de tareas se utilizan principalmente en el área HCI con relación al análisis de requisitos para el desarrollo de interfaces de usuario. En este trabajo se utilizan para identificar las tareas que debe llevar a cabo el usuario para conseguir sus objetivos. Estas tareas se recogen en el modelo de tareas, uno de los 5 modelos que constituyen el modelo de interfaz propuesto.

El término "caso de uso" no tiene una definición fija, de hecho un reciente estudio [Hur97], ha identificado mas de treinta variantes de casos de uso y técnicas relacionadas. En este trabajo se utiliza una variante de la propuesta por Cockburn [Coc97], que es muy similar a la propuesta originalmente en [Jac92] pero mucho más fácil de comprender y utilizar.

Cockburn proporciona las siguientes definiciones:

Un *escenario* es una secuencia de interacciones que ocurren bajo ciertas condiciones para conseguir el objetivo del actor principal, y que tienen un resultado determinado con respecto a ese objetivo.

Las interacciones comienzan a partir de una acción de disparo (trigger) y continúan hasta que el objetivo es alcanzado o abandonado por alguna razón.

Un *caso de uso* es una colección de posibles escenarios entre el sistema y los actores externos, caracterizados por el objetivo del actor principal y que muestra cómo dicho objetivo puede ser completado con éxito o puede fallar.

Destacar que al ser el objetivo el hilo conductor a la hora de identificar los casos de uso según la aproximación de Cockburn, hace que la correspondencia con el análisis de tareas donde el énfasis está en comprender las tareas del usuario en términos de las actividades que tiene que llevar a cabo para conseguir el objetivo perseguido sea mucho más fácil y natural. En este trabajo se considera que un caso de uso se corresponde con una o más tareas de usuario.

4. OASIS como Lenguaje de Especificación de Interfaces de Usuario

El modelo OASIS provee una rica expresividad semántica para la especificación de modelos conceptuales bajo el paradigma orientado a objetos. La presentación de OASIS 3.0 como lenguaje, junto con su gramática se encuentra detallada en [Let98]. En este apartado y a modo de presentación, se describen brevemente sus características principales.

Uno de los objetivos a realizar en el trabajo de investigación que se lleva a cabo es precisamente realizar las extensiones, tanto sintácticas como semánticas, necesarias para que sirva como soporte

formal del proceso de desarrollo de interfaces de usuario.

En OASIS, los bloques básicos de construcción en un sistema de información son los *objetos*. Un **objeto** OASIS es un proceso observable [Ram93] que encapsula estado y comportamiento. Cada objeto tiene un identificador (oid) que será único y lo distingue de cualquier otro objeto que haya existido, exista o pueda existir. Los objetos poseen propiedades, representadas en el modelo a través de la noción de *atributo.*

El estado de un objeto viene dado por el conjunto de valores de sus atributos, de forma que un cambio en alguno de dichos valores representará un cambio de estado. La causa de que se produzca un cambio de estado en un objeto se denomina *evento*. La ocurrencia de un evento en la vida de un objeto sólo tendrá éxito si el objeto se encuentra en un estado que verifica la *precondición* asociada a dicho evento. Además, no todos los estados de los objetos van a ser estados permitidos; en este sentido, el modelo permite la definición de *restricciones de integridad* que definen precisamente qué estados serán aceptables en la vida de los objetos, en términos de valores permitidos para los atributos.

Los objetos no están aislados. La interacción entre objetos se modela en OASIS de dos formas: mediante compartición de eventos y relaciones de disparo. Las relaciones de disparo, representadas en el modelo OASIS por *triggers*, introducen actividad en la sociedad de objetos, permitiendo que un objeto actúe como agente de un evento de otro objeto cuando se satisfagan en él determinadas condiciones.

Los objetos que comparten las mismas propiedades se agrupan en *clases*. Un objeto individual es una *instancia* o ejemplar de una clase. En este sentido, la sociedad de objetos de nuestro modelo OASIS está compuesta de las siguientes clases de objetos:

Las **clases primitivas** o dominios están constituidas por "objetos" que tienen un único estado y que siempre existen. En general, abarcan el dominio de los tipos abstractos de datos (TAD). La sociedad de objetos se construirá tomándolos como nivel estructural básico sobre el que se declaran las demás clases de objetos.

A partir de los dominios se construyen las **clases elementales**. Estas vienen caracterizadas por el tipo que una colección de objetos comparte. En el tipo se definen los atributos, los eventos, las restricciones de integridad, las precondiciones, las relaciones de disparo, las transacciones y la parte proceso que caracterizan a los objetos de la clase.

Mediante un mecanismo constructivo, se definen las **clases complejas** a partir de las clases elementales o de otras clases complejas definidas de antemano, aplicando sobre ellas los operadores de clases**.** Los operadores más utilizados son la *especialización, agregación*, *generalización*, *asociación* y la *composición paralela*.

El modelo objetual visto está soportado por el lenguaje OASIS. OASIS es un lenguaje formal de especificación para sistemas de información abiertos y activos, según el modelo OO presentado. El marco formal viene dado por la lógica dinámica, propuesta por Harel [Har84]. En este trabajo, se utiliza la versión OASIS 3.0 [Let98], basada en la lógica dinámica que permite, entre otras cosas, representar los operadores de obligación, prohibición y permiso usados en lógica deóntica.

La especificación formal de la interfaz de usuario se abordará teniendo en cuenta el nivel de abstracción donde nos encontremos. Así se producirán unas especificaciones a nivel de requisitos y análisis para formalizar los modelos correspondientes a estas fases y otras especificaciones a nivel de diseño, donde se formaliza el diseño de la interfaz.

5. IDEAS[2]: Desarrollo de Interfaces de Usuario en un Entorno de Producción Automática de Software Orientado a Objetos

El objetivo de este trabajo es la integración de un Modelo de Interfaz de Usuario dentro del entorno de desarrollo de software definido por el modelo OASIS presentado en el apartado anterior, a partir del cual se pueda generar la interfaz de la aplicación.

El proceso de especificación de la Interfaz de Usuario se realiza de forma paralela al desarrollo de lo que es la aplicación en sí. Este proceso de desarrollo se puede ver en la figura 1.

De esta forma, IDEAS [Loz99b, Loz00] pretende ser un sistema de desarrollo automático de Interfaces de Usuario integrado en el marco de trabajo de OASIS para soportar de forma automática la producción de interfaces de usuario de alta calidad.

[2] ***I**nterface **D**evelopment **E**nvironment within OA**SI**S*

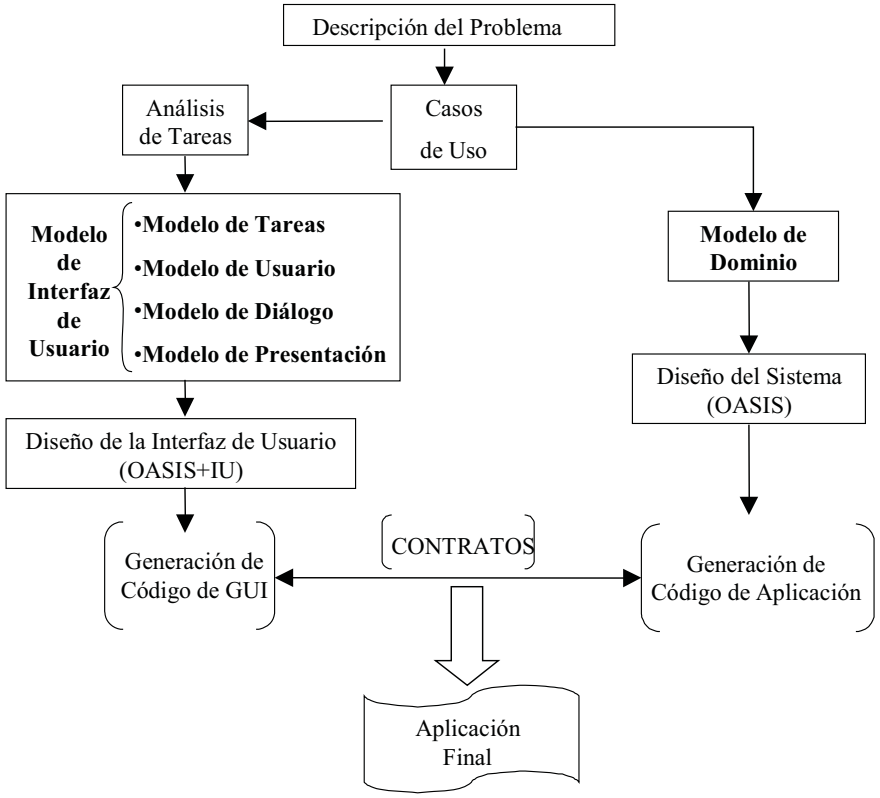


Figura 1. Integración del Proceso de Desarrollo de Interfaces de Usuario en el Marco de OASIS.

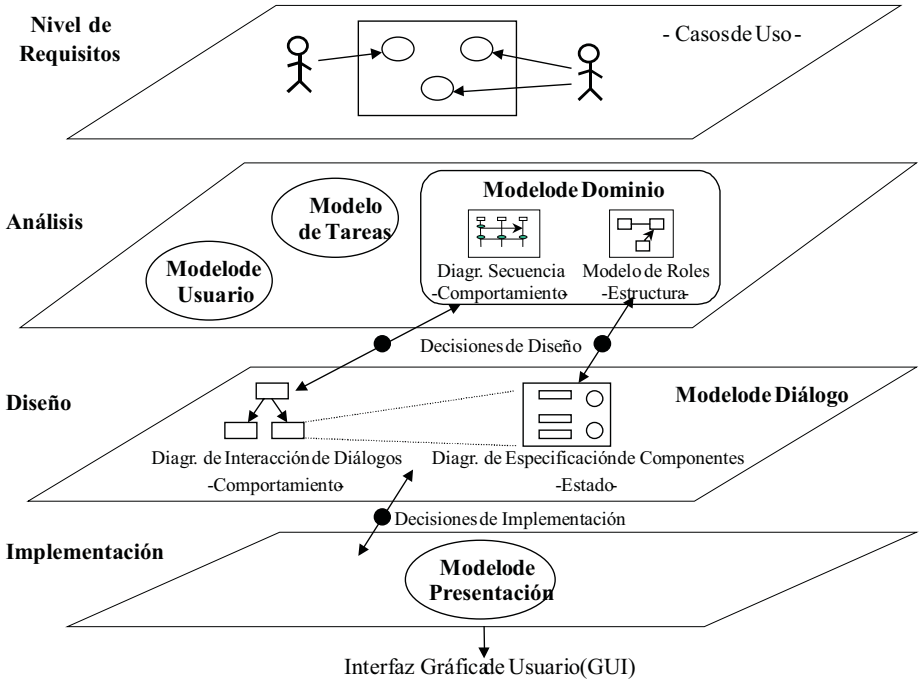


Figura 1. Proceso de Desarrollo de Interfaces de Usuario Propuesto.

A continuación describimos este proceso de desarrollo por niveles y siguiendo el orden en el que se van desarrollando los diferentes modelos que conforman nuestra propuesta. En la figura 2 se representa este desarrollo.

A nivel de requisitos se emplea la técnica de casos de uso [Jac92]. En esta primera etapa y en relación con cada caso de uso se van identificando los diferentes tipos de usuarios del sistema. Posteriormente cada caso de uso se refina y se especifican las políticas del negocio, entidades y actores que participan en él.

A nivel del análisis se generan, en primer lugar el Modelo de Tareas, identificando para cada caso de uso de la fase anterior la/s tarea/s correspondiente/s.

A continuación se realiza el Modelo de Dominio, que en este caso viene dado por el Diagrama de Secuencia asociado a cada uno de los casos de uso (tareas) identificados previamente, que define el comportamiento del sistema, y el Modelo de Roles, que define la estructura de las clases que intervienen en el diagrama de secuencia asociado junto con la relación existente entre ellas especificando el rol que desempeña cada una de las clases en ese caso concreto.

Una vez definidos el modelo de tareas y el modelo de dominio, se define el Modelo de Usuario. Para cada tipo de usuario del sistema se establece el subconjunto de tareas que le están permitidas llevar a cabo. A su vez para cada una de esas tareas se establece una proyección de las acciones que dentro de esa tarea puede o no acometer. Y por último y de acuerdo a las características particulares (independientes de la aplicación) del usuario se establece la forma más adecuada de mostrar la información.

A nivel de diseño se elabora el Modelo de Diálogo [Loz99a]. Hasta ahora los modelos generados no contienen ningún aspecto gráfico o de contenido de la interfaz de usuario. Es a partir de aquí cuando se empiezan a diseñar estos aspectos y se hace especial hincapié en cómo se llevará a cabo la interacción del usuario con el sistema.

El modelo de diálogo consta de dos diagramas: el Diagrama de Interacción de Diálogos y el Diagrama de Especificación de Componentes. El primero especifica el comportamiento de la IU, es decir, las transiciones de ventanas o diálogos que van teniendo lugar en función de las selecciones que sobre la IU va haciendo el usuario. Para generar este diagrama se tiene en cuenta el Diagrama de Secuencia elaborado en la etapa de análisis. En función de las decisiones de diseño que tome el diseñador de la IU se establece la secuencia de ventanas o diálogos necesarios para llevar a cabo el comportamiento requerido. Un ejemplo de este diagrama se puede ver en la siguiente figura.

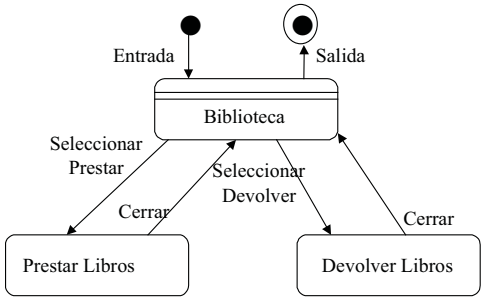


Figura 3. Diagrama de Interacción de Diálogos.

El segundo diagrama consiste en establecer, para cada una de las ventanas o diálogos resultantes del primer diagrama, el conjunto de herramientas de control y de visualización necesarias para dotarlos de la funcionalidad requerida para llevar a cabo la tarea de usuario en cuestión. Este segundo diagrama define el estado de la IU tal y como se muestra en la figura 4.

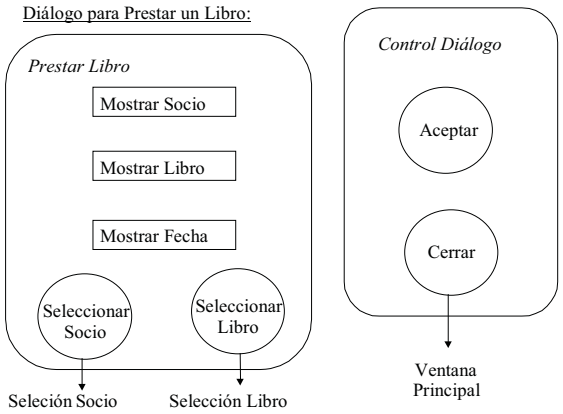


Figura 4. Diagrama de Especificación de Componentes.

A nivel de implementación se genera el Modelo de Presentación. En este caso y en función de las decisiones de implementación que se tomen, la plataforma final de implementación y de acuerdo a la guía de estilo seguida, se especifican los objetos de interacción concretos que formarán parte de la interfaz gráfica de usuario (GUI).

6. Usabilidad. Aspecto esencial en el desarrollo de interfaces de usuario

El objetivo principal a la hora de construir software es facilitar a los usuarios la consecución de objetivos concretos de forma sencilla. Esto es, conseguir que el sistema sea fácil de aprender y de recordar su manejo, sea útil, es decir, contenga las funciones que los usuarios necesitan en su trabajo, y que su manejo sea fácil y agradable. En definitiva el objetivo básico es conseguir un sistema con un alto grado de *"usabilidad"* [Gou85].

Los sistemas construidos con criterios de usabilidad tienen una serie de ventajas tales como la mejora de la productividad, la reducción del coste y del tiempo de aprendizaje, y el incremento de la autonomía de los usuarios finales.

Es más, Landauer [Lan95] observó que el 80% de los costes de mantenimiento – lo que representa el 80% del total de los costes del desarrollo de software – se deben a problemas de los usuarios con el sistema y no a problemas técnicos. Además indica que el 64% de estos costes están relacionados con problemas de usabilidad. Además, varios estudios sobre el tema de la productividad en herramientas software concluyen que la facilidad de uso y aprendizaje son unas de las razones más importante a la hora de incrementar de forma efectiva la productividad de los usuarios del software.

Esta situación resalta la importancia de la usabilidad y justifica la necesidad de incorporarla antes, durante y después del desarrollo de sistemas software. Esto significa que el desarrollo de las aplicaciones informáticas es una actividad multidisciplinar que incorpora factores humanos y técnicas y conocimientos ergonómicos con el objetivo de mejorar las condiciones de trabajo del usuario final.

Como plantea B. Shneiderman [Shn98], dentro de lo que se denomina "ingeniería de la usabilidad", una de las bases para alcanzar altas cotas de calidad es utilizar métodos de diseño iterativo que permitan realizar pruebas tempranas sobre prototipos, revisiones basadas en la retroalimentación de las observaciones y reacciones de los usuarios y los refinamientos sucesivos propuestos por los expertos en usabilidad. A su vez, J. Nielsen [Nie93] plantea que el modo más barato de aplicar con éxito los criterios de usabilidad es realizar el mayor esfuerzo posible antes de que el proceso de diseño de la presentación de la interfaz comience. En este aspecto lo más importante es realizar un estudio previo que permita "conocer a los usuarios". Este conocimiento no sólo se basa en estudiar sus características o habilidades individuales sino que también es necesario conocer las tareas que necesitan realizar. Igualmente dentro del ámbito de la ingeniería de la usabilidad es importante que los usuarios puedan realizar evaluaciones (test de usabilidad) sobre prototipos del sistema final, con lo que pueden aportar mejorar a incorporar en la siguiente versión.

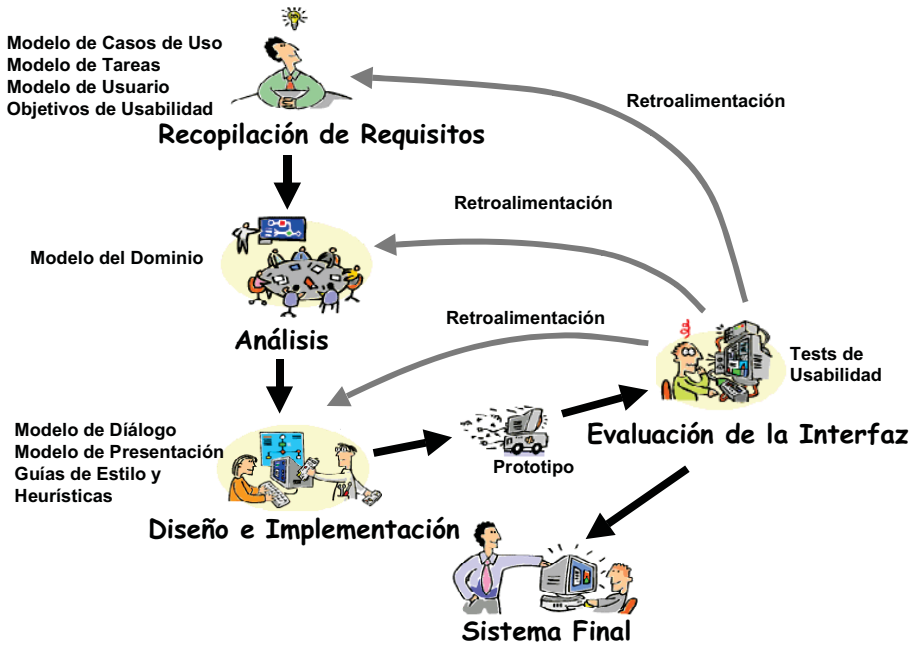


Figura 5. Proceso de Desarrollo Iterativo Incorporando Criterios y Test de Usabilidad.

Por ello, nuestra propuesta incluye modelos de alto nivel que permitan "conocer a los usuarios". En este punto, IDEAS propone la utilización de modelos de casos de uso, modelos de tareas y modelos de usuario. Los primeros permiten describir de manera precisa las tareas que realiza cada tipo de usuario, y el último nos ayuda a especificar las características individuales de los diferentes usuarios.

Tras conocer a los usuarios y las tareas que realizan, se propone un estudio que permita establecer las metas o criterios básicos de usabilidad que el sistema debe alcanzar. Esta definición permitirá más adelante realizar test de usabilidad asociadas a dichos criterios.

En las últimas etapas del desarrollo, a la hora de diseñar el modelo de presentación, el diseñador debe tener presente diferentes guías de estilo y heurísticas que le permitan mejorar la usabilidad de sus diseños. A su vez, a partir de dichos modelos de presentación se generan los prototipos de las interfaces. Estos prototipos son evaluados (test de usabilidad) por los usuarios finales de la aplicación.

Finalmente, todo lo anterior se integra dentro de un proceso de diseño iterativo. Este permite realizar el proceso de retroalimentación introduciendo en el desarrollo de la interfaz la información recogida en las pruebas realizadas por los usuarios, mejorando notablemente la usabilidad de las interfaces diseñadas tal y como se refleja en la figura 5.

7. Conclusiones.

En este trabajo se ha tratado el tema de la generación de interfaces de usuario dentro del proceso de desarrollo de software orientado a objetos. Este desarrollo se lleva a cabo aplicando técnicas de casos de uso y análisis de tareas, que nos llevan a la construcción de los diferentes modelos que constituyen el modelo de interfaz de usuario, y que representan cada uno de ellos los diferentes aspectos involucrados en la construcción de la interfaz.

Junto a lo anterior se ha descrito el modo en que se incorporan los criterios y tareas encaminadas a conseguir interfaces de alta usabilidad. Se han descrito qué modelos aportan información relevante dentro del desarrollo de sistemas usables y se han introducido ciertas tareas encaminadas a mejorar dicha usabilidad. Finalmente se ha planteado un proceso de diseño iterativo que permite incorporar las sucesivas mejoras que los usuarios introducen en el proceso de diseño.

Referencias

[Bal96] H. Balzert et al.: The JANUS application Development Environment Generating More than the User Interface. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 183-205.

[Bod95] F. Bodart et al.: Towards a Systematic Building of Software Architectures: teh TRIDENT Methodological Guide. In: Design, Specification and Verification of Interactive Systems. Wien: Springer, 1995, 262-278.

[Bod96]F. Bodart et al.: Key Activities for a Development Methodology of Interactive Applications. In : Critical Issues in User Interface Systems Engineering. London: Springer, 1996, 109-134.

[Coc97]A. Cockburn: Structuring Use Cases with Goals. Journal of Object-Oriented Programming, 10(5) y 10(7), 1997. http://members.aol.com/acockburn/papers/usecases.htm

[Elw95] T. Elwert, E. Schlungbaum: Modelling and Generation of Graphical User Interfaces in the TADEUS Approach. In: Designing, Specification and Verification of Interactive Systems. Wien: Springer, 1995, 193-208.

[Fol95] J. Foley, P. Sukaviriya: History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In F. Paterno (ed.): Interactive Systems: Design, Specification and Verification. Berlín: Springer, 1995, 3-14.

[Gou85] J.D. Gould, C. Lewis. Designing for usability: key principles and what designers think. Communications of the ACM, (1985)28, 300-311.

[Har84] D. Harel: Dynamic Logic, In Handbook of Philosophical Logic II, editors D.M. Gabbay, F; Guenter ; pags, 497-694. Reidel 1984.

[ISO98]Human-Centered Design Processes for Interactive Systems. Draft of the ISO-DIS 13407 International Standard. 1998.

[Jac92] I.Jacobson, M. Christerson, et al.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

[Jan93]C. Janssen: Generating User Interfaces from Data Models and Dialog Net Specifications. In: Bridges between Worlds. Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 418-423.

[Joh95] P. Johnson, H. Johnson, S. Wilson: Rapid Prototyping of User Interfaces Driven by Task Models. In: J. Carroll (ed.) Scenario-Based Design.London: John Wiley & Son, 1995, 209-246.

[Lan95] Landauer, T.K. “The Trouble with Computers: Usefulness, Usability and Productivity. MIT Press, 1995.

[Let98] P. Letelier, I. Ramos, P. Sanchez, O. Pastor. OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling. SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[Lon96] F. Lonczewski: The FUSE System: An Integrated User Interface Design Environment. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 37-56.

[Loz00] M.Lozano, I. Ramos, P. González. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development. 34th International Conference on Technology of Objetct-Oriented Languages and Systems. TOOLS-USA 2000. Santa Barbara, CA. 30 Julio - 3 Agosto, 2000

[Loz99a] M.Lozano, I. Ramos: An Object Oriented Approach for the Specification of User Interfaces. PACRIM'99, Victoria -Canadá- Agosto, 1999.

[Loz99b] M. Lozano, I. Ramos: Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. ECOOP'99, Lisboa -Portugal-, Junio, 1999.

[Mär96] C. Märtin: Software Life Cycle Automation for Interactive Applications: The AME Design Environment. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 57-74.

[Nie93] J. Nielsen. Usability Engineering. Morgan Kaufmann, 1993.

[Pas92] O. Pastor. Diseño y Desarrollo de un Entorno de Producción Automática de Software Basado en el Modelo Orientado a Objetos. Tesis Doctoral. Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia. 1992.

[Pue94a] A. Puerta et al.: Beyond Data Models for Automated User Interface Generation. In: People and Computers IX. Proceedings British HCI'94 (Glasgow UK, Agosto 1994). Cambridge: Cambridge University Press, 1994, 353-366.

[Pue94b] A. Puerta, P. Szekely: Model-based Interface Development. CHI'94 Tutorial Notes, 1994.

[Pue96] A. Puerta: The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 19-36.

[Ram93] Ramos, I. et. al., Objects as observable processes, in Proc. of the 4th International Workshop on the Deductive Approach to Information Systems and Databases, Tech. Report, DLSI, U. P. Catalunya, 1993.

[Shn98] B. Shneiderman. Designing the User Interface. Strategies for Effective Human-Computer Interaction. Addison Wesley, 1998.

[Sze93] P. Szekely et al: Beyond Interface Builders: Model-Based Interface Tools. In: Bridges between Worlds. Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 383-390.

[Sze96] P. Szekely, et al.: Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach. In: Engineering for Human-Computer Interaction. Proceedings of the IFIP Conference on Engineering for Human-Computer Interaction (Yellowstone Park, Agosto 1995). London: Chapman &Hall, 1996, 120-150

[Wie90] C. Wiecha et al.: ITS: A Tool for Rapidly Developing Interactive Applications. ACM Transactions on Information Systems 8 (1990), 3, 204-236.

A GRAPHICAL USER INTERFACE DEVELOPMENT TOOL

María Lozano
Computer Science Department
University of Castilla-La Mancha
Albacete, Spain
mlozano@info-ab.uclm.es

Pascual González
Computer Science Department
University of Castilla-La Mancha
Albacete, Spain
pgonzalez@info-ab.uclm.es

Isidro Ramos
Department of Information Systems and Computation
Valencia University of Technology
Valencia, Spain
iramos@dsic.upv.es

ABSTRACT

The main goal of information systems development is getting a system that allows final users to reach their goals and perform their tasks in an effective and efficient way. In order to achieve this aim it is necessary, not only to take into account system requirements, but it is also important to incorporate new techniques to capture the final user's needs by incorporating usability criteria. This way, we can develop more intuitive and easy to use user interfaces that can help final users to make the most of computer systems. With this aim, a new user interface development tool within the object-oriented software development process is proposed in this paper.

Keywords

User Interface, Object-Oriented Modeling Techniques, Human Computer Interaction.

1. INTRODUCTION

Most of the research developed by industry and academy has led to several object-oriented methods and tools highly adequate for the development of computer systems. But most of these tools do not include mechanisms (models) for identifying and specifying user needs and requirements as well as testing and validating requirements with end-users before, during and after development. These considerations are especially important in interactive systems, where the user interaction is very high and user interface is a significant part of the system.

As a result of this weakness, interactive systems developed using such tools can meet all technical requirements, be very robust, and yet be unusable by the end-user. As a solution to this problem we propose a method for integrating a user interface model, taking into account user needs and requirements, in the software life cycle. Besides, this approach to the development process ensures the quality of delivered applications from the point of view of the end user. In this paper we present IDEAS[1] an object oriented and model-based approach for the development of user interfaces within an automatic software production environment which covers the software life cycle of analysis, design and implementation within the principles of the object oriented paradigm. So, for better understand our proposal, we first present the state-of-the-art of model-based user interface development environments, and then we finally present our approach with an example of use.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Proceedings Volume 2 of the
16th British HCI Conference
London, September 2002

© 2002 British HCI Group

2. MODEL-BASED USER INTERFACE DEVELOPMENT ENVIRONMENTS. STATE OF THE ART.

In this section we briefly describe the most understanding features of current model-based user interface development environments. In a state of the art report, B. Myers [1] presents a classification of user interface software tools. This classification is based on the way the user interfaces developers can specify the layout and dynamic behaviour of a user interface. This classification is the following:

- Language-based Tools: These tools require the developer to program in a special purpose language.
- Interactive Graphical Specification Tools: These tools allow an interactive design of the user interface.
- Model-based Generation Tools: These tools use a high-level specification or model to generate the user interface automatically.

The last classification seems to be the most adequate to tackle the user interface development. The first two support the specification of either the dynamic behaviour or the layout of the user interface in an easy way, but not both parts at the same time. And the same as most current tools, they only support the last phase of the user interface life-cycle development,

[1] **I**nterface **D**evelopment **E**nvironment within o**AS**is.

and the abstractions they provide do not have a direct connection with the task analysis results.

On the other hand, model-based user interface development approach and its corresponding tools is an emerging technology that remedies these shortcomings. Interfaces are developed by using specialized design-time tools to build and refine models. Developers specify "what" features the interface should have, rather than write programs that specify "how" to make the computer exhibit the desired behaviour.

The most representative feature of the model-based approach is that all aspects of the user interface design are represented using declarative models.

For a user interface tool to be considered as a model-based user interface development environment, it must fulfil the following criteria:

1. It must include a high-level, abstract and explicitly represented (declarative) model about the system under development.
2. It must clearly establish a technologically well-supported relation between 1) and the final running user interface.

Currently there exist several model-based user interface software tools, as for instance, HUMANOID [2], MOBI-D [3], MASTERMIND [4], TRIDENT [5], FUSE [6], JANUS [7] and IDEAS our proposed model explained in following sections.

A comparison of IDEAS with different model-based interface development environments is shown in Figure 1. As we can see, IDEAS together with MOBI-D in the academic field and OVID [8] in the industrial field, is a complete environment, i.e. it supports all declarative models from the task analysis to the presentation, whereas most of the environments simply focus on the specification of the first models, above all the domain model, to generate the Graphical User Interface (GUI). In fact, IDEAS includes a new model, the use case model, previous to the task model to explicitly capture the GUI requirements.

Compared with all the environments depicted in figure 1, IDEAS presents three main contributions. Firstly, IDEAS follows a guided and methodological approach. It starts with the requirements gathering by means of widely accepted and proven techniques such as use cases to subsequently refine the conceptual models in every stage of the development. Secondly, whereas most of the environments generate script or C++ files for any User Interface Management System, in IDEAS the GUI is generated in XUL an XML-based markup language, which guarantees the portability of the GUI in different

	Use Cases	Tasks	Domain	User	Dialog	Presentation
UIDE			* C++ Classes			
AME			* OO Model			
JANUS			* OO Model			
GENIUS			* E-R Model		*	
ADEPT		*		*		
MASTERMIND		*	* CORBA IDL			*
MECANO			* OO Model			
MOBI-D		*	* Object Model MIMIC	*	*	*
TRIDENT		*	* E-R Model			
FUSE		*	* Algebraic Specification	*	*	
TADEUS		*	* OO Model	*	*	
OVID		*	* OO Model	*	*	*
IDEAS	*	*	* OO Model	*	*	*

Figure 1. Model-based user interface development environments.

platforms. This is, the existing solutions guide the generation of the GUI according to the final platform where it is going to run. Nevertheless, IDEAS allows the implementation of the GUI in any platform or environment, either a web environment or a PC with a particular operating system, a mobile phone or a PDA.

And lastly, the relationship existing between the GUI and the domain objects is explicitly established by means of "contracts". Most environments either mix directly the user interface objects with the domain objects to generate automatically the GUI or they do not establish explicitly the correspondences between both objects societies.

3. User Interface Development Process with the IDEAS Tool.

Abstraction is one of the basic principles to understand and model the reality. The object oriented paradigm favours this principle as it conceives the object oriented development process as an iterative and incremental rapprochement that progressively allows obtaining a detailed specification of the system.

The user interface development process within IDEAS is tackled following this principle. This process is not plain, but it is structured in multiple levels and multiple perspectives. The vertical structuring shows the reification processes followed from the first and most abstract level passing through the following level to reach finally the system implementation, which constitute the last level. On the other hand, the horizontal structuring shows the different perspectives offered by the different models developed in every one of the vertical levels. Thus, different models are used at the same abstract level to describe the different aspects of the GUI.

Following these ideas, we propose the user interface development process depicted in figure 2. Due to space constrains, we cannot detail the different models proposed, so we will describe briefly this process showing some examples of the implemented tool which, interactively and automatically, supports this methodological approach.

At **requirements level** three models are created: Use Case Model, Task Model and User Model.

The *Use Case Model* captures the use cases identified within the information system as originally proposed by Jacobson [9] and currently used by the UML. Then, for every one of the use cases there will be one or more tasks the user may perform to accomplish the functionality defined by the use case. These tasks will be modelled in the task model.

The *Task Model* defines the ordered set of activities and actions the user has to perform to achieve a concrete purpose or goal. We propose a template based on the one proposed by Cockburn [10] to describe in natural language all these issues.

The *User Model* describes the characteristics of the different types of users. The purpose of this model is to support the creation of individual and personalised user interfaces.

At **analysis level** the *Domain Model* is generated. This model consists of two diagrams. The first one is the Sequence Diagram, which defines the system behaviour. The second one is the Roles Model, which defines the structure of the classes that take part in the associated sequence diagram together with the relationships among these classes, specifying the role of each one of them.

At **design level** the *Dialogue Model* is developed. All the models that have been generated up to now do not contain any graphical aspect of the final user interface. It is from now on that these issues start to be addressed and the way in which the user-system interaction will be performed is especially import nt.

The purpose is to describe the syntactic structure of the user-computer interaction. It establishes when the user can invoke commands, select or specify the input data and when the computer can require data from the user or display the output data. These items are modelled by means of Abstract Interaction Objects (AIO).

At **implementation Level** the *Presentation Model* is created. The Presentation Model describes the concrete interaction objects composing the final GUI, its design characteristics and visual dependencies among them. This model leads to the visualisation of the final GUI according to the final platform style guides. The final GUI generation is performed by using XUL[xul] an XML based language, in order to make it as much independent as possible from the final platform where the application is going to run.

The starting point to generate the GUI in XUL is the Dialog Model developed at design level, which, as stated before, models the structure and the behaviour of the GUI by means of AIOs. These AIOs are translated into the Concrete Interaction Objects offered by XUL.

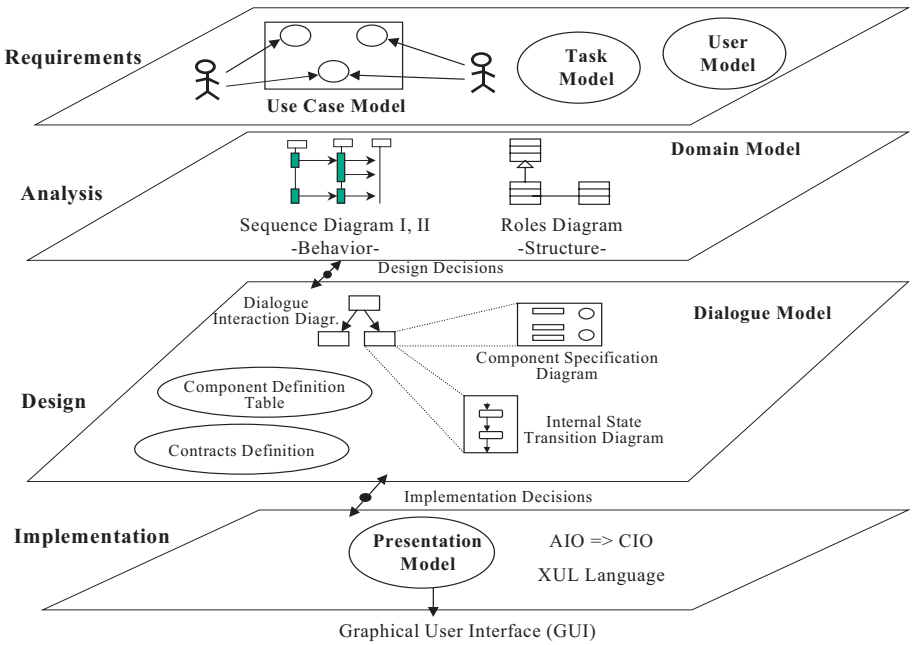


Figure 2. IDEAS: User Interface Development Environment.

Therefore, the GUI structure is generated automatically from the Component Specification Diagram created at design level. The behaviour is currently written manually in JavaScript.

Next we show an over-simplified example developed with the tool that supports the process explained before. The example is based on a Conference Review System case study.

Figure 3 shows the use case diagram for the Conference Review system, showing also the different development stages in which the tool is structured.

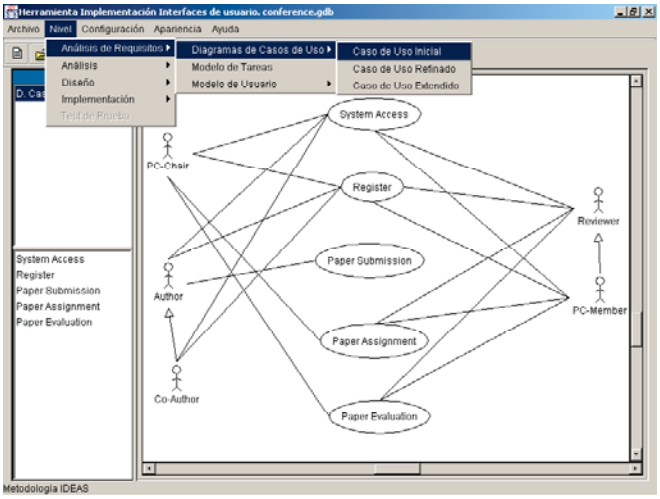


Figure 3. IDEAS Tool: Initial Use Case Diagram.

Following the development process proposed within IDEAS, at design level we finally get the Component Specification Diagram showed in figure 4. This diagram shows the Abstract Interaction Objects the user needs to perform the task of logging in the system.

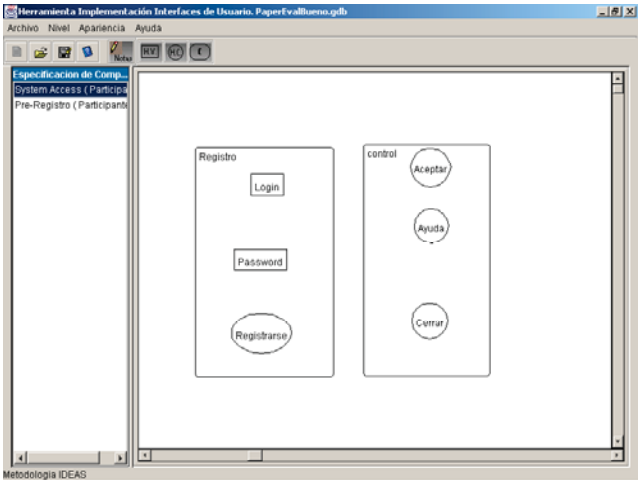


Figure 4. Component Specification Diagram corresponding to the "log in the system" task

Starting from this diagram, the AIO are translated into CIO as depicted in figure 5, generating automatically the final GUI shown in figure 6.

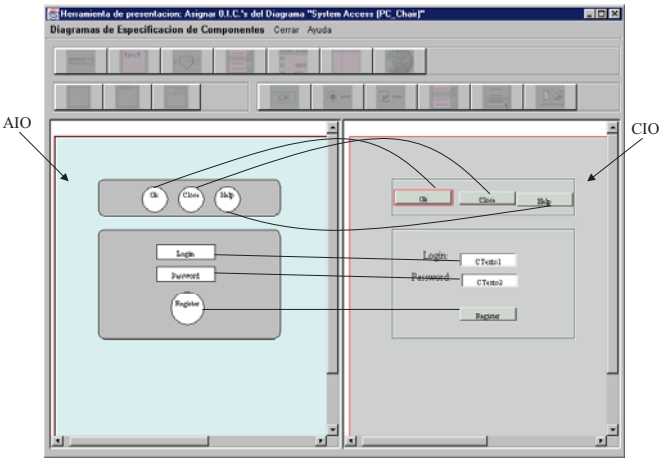


Figure 5. Conversion of AIO into CIO.



Figure 6. Final Graphical User Interface.

4. REFERENCES

[1] Myers, B. A.: User Interface Software Tools. ACM Trans. on Computer-Human Interaction 2 64-103, 1995.

[2] Szekely P. et al: Beyond Interface Builders: Model-Based Interface Tools. In: Bridges between Worlds. Proceedings InterCHI'93, 383-390. Amsterdam, April 1993.

[3] Puerta, A.: The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In CADUI. Namur University Press, 19-36, 1996.

[4] Szekely, P., et al.: Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach. In: Engineering for HCI. London: Chapman &Hall, 120-150, 1996.

[5] Bodart, F. et al.: Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide. In: Design, Specification and Verification of Interactive Systems. Wien: Springer, 262-278, 1995.

[6] Lonczewski, F.: The FUSE System: An Integrated User Interface Design Environment. In CADUI. Namur University Press, 37-56, 1996.

[7] Balzert, H. et al.: The JANUS application Development Environment Generating more than the User Interface. In: CADUI. Namur University Press, 183-205, 1996.

[8] Roberts, D., Berry, D., Isensee, S., Mullaly, J. Designing for the User with OVID. Macmillan Technical Publishing. IBM, 1998.

[9] Jacobson, I., M. Christerson, et al.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

[10] Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, 2001.

Integrating Usability within the User Interface Development Process of Web Applications*

MARÍA LOZANO, PASCUAL GONZÁLEZ, FRANCISCO MONTERO, JOSE PASCUAL MOLINA
Computer Science Department
University of Castilla-La Mancha
Albacete -SPAIN-

ISIDRO RAMOS
Dept. of Information Systems and Computation
Valencia University of Technology
Valencia -SPAIN-

The main goal of information systems development in general, and web oriented applications in particular, is obtaining a system that allows final users to reach their goals and perform the tasks needed to achieve the goals in an efficient and effective way. To achieve this aim it is necessary, not only taking into account system requirements, but also incorporate new techniques that help to capture user needs considering usability criteria. In this way, it is possible to develop intuitive and easy-to-use user interfaces that help the final users to derive the maximum benefit from web applications. With this aim, a new user interface development environment within the object-oriented software development process is proposed in this paper.

Categories and Subject Descriptors: H5.2 [**Information Interfaces and Presentation**]: User Interfaces - *User-centered design*; *Interaction styles*; *Theory and methods.*
General Terms: Human Factors
Additional Key Words and Phrases: Human-computer interaction, object oriented modeling

1. INTRODUCTION

Most of the research developed by industry and academy has led to several object-oriented methods highly adequate for the development of systems. But most of these methods do not include mechanisms (models) for identifying and specifying user needs and requirements as well as testing and validating requirements with end-users before, during and after development. These considerations are especially important in web applications and interactive systems in general, where the user interaction is very high and the user interface is a significant part of the system.

As a result of this weakness, interactive systems developed using such methods can meet all technical requirements, be very robust, and yet be unusable by the end-user. This problem explains a large part of the frequently observed phenomenon whereby large numbers of change requests to modify the services of a web application are made after its deployment. As a solution to this problem we propose a method for integrating a user interface model, taking into account user needs and requirements, in the software life cycle. Besides, this approach to the development process ensures the quality of delivered applications from the point of view of the end user.

* This research is partly supported by the DOLMEN and GEOZOCO projects from the Spanish government commission of Science and Technology with reference CICYT-TIC 2000-1673-C06-06 and CICYT-TIC 2000-1106-C02-02.

Lozano, M., González, P., Montero, F. Molina, J. P., Ramos, I. Integrating Usability within the User Interface Development Process of Web Applications. Proc. of the 2nd International Workshop on Web Oriented Software Technology at ECOOP'02. 2002.

There exists several reasons that justify the complexity of design and implementation of user interfaces [1], but perhaps the main reason is the difficulty of understanding the tasks the user must perform with the system and the characteristics of the different kind of users managing it [2]. In this sense, it seems necessary to define new models that allow collecting user requirements and specifying the different characteristics of the interface associated to a concrete application. These models should allow defining the specification of the interface independently [3], that is, the models used to specify the user interface should be different from those used to specify the application.

In this paper we present IDEAS[1] an object oriented and model-based approach for developing user interfaces within an automatic software production environment. This environment is supported by an object-oriented model called OASIS[2] [4], [5], which covers the software life cycle of analysis, design and implementation within the principles of the object oriented paradigm. So, for better understand our proposal, we first present the state-of-the-art of model-based user interface development environments, and then we finally present IDEAS with an example of use.

2. MODEL-BASED USER INTERFACE DEVELOPMENT ENVIRONMENTS. STATE OF THE ART

In this section we briefly describe the most understanding features of current model-based user interface development environments. In a state of the art report, B. Myers presents a classification of user interface software tools [6]. This classification is based on the way the user interfaces developers can specify the layout and dynamic behavior of a user interface. This classification is the following:

- Language-based Tools: These tools require the developer to program in a special purpose language.
- Interactive Graphical Specification Tools: These tools allow an interactive design of the user interface.
- Model-based Generation Tools: These tools use a high-level specification or model to generate the user interface automatically.

The last classification seems to be the most adequate to tackle the user interface development. The first two support the specification of either the dynamic behavior or the layout of the user interface in an easy way, but not both parts at the same time. And the same as most current tools, they only support the last phase of the user interface life-cycle development, and the abstractions they provide do not have a direct connection with the task analysis results.

On the other hand, model-based user interface development approach and its corresponding tools is an emerging technology that remedies these shortcomings. Interfaces are developed by using specialized design-time tools to build and refine models. Developers specify "what" features the interface should have, rather than write programs that specify "how" to make the computer exhibit the desired behavior.

The main shortcoming of the model-based approach is that building models is difficult. A rich model describing all the features of an interface is complex and non-trivial to specify.

[1] **I**nterface **D**evelopment **E**nvironment within o**AS**is.
[2] **O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems

The most representative feature of the model-based approach is that all aspects of the user interface design are represented using declarative models. This paradigm offers several benefits:

- User-centered development life cycle.
- Centralized user interface specification.
- Design tools for an automated and interactive development.
- Reuse of the user interface designs.

For a user interface tool to be considered as a Model-based user interface development environment, it must accomplish the following criteria:

1. It must include a high-level, abstract and explicitly represented (declarative) model about the system under development.
2. It must clearly establish a technologically well-supported relation between 1) and the final running user interface.

Currently there exist several model-based user interface software tools, as for instance, UIDE [7], HUMANOID [8], MOBI-D [9], MASTERMIND [10], [11], TRIDENT [12], FUSE [13], JANUS [14] and IDEAS [15]our proposed model explained in following sections.

3. IDEAS: INTERFACE DEVELOPMENT ENVIRONMENT WITHIN OASIS

The aim of this research is the integration of a user interface Model within an object oriented software production environment. The user interface specification process is tackled in parallel to the application development, and according to the common principles of Model-based User Interface Development Environments, as exposed in previous section. This development process is depicted in figure 1.

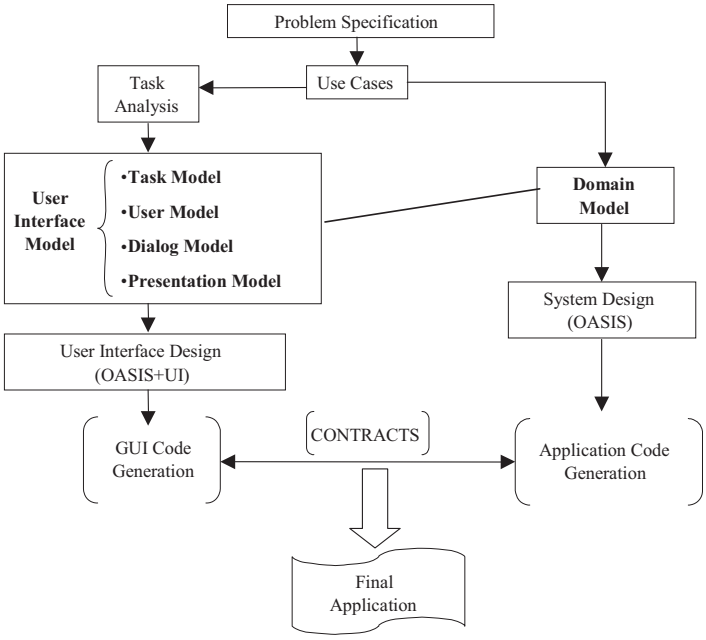


Fig. 1. IDEAS: Interface Development Environment within OASIS.

IDEAS aims to be an automatic user interface development system integrated within the framework of the object oriented model OASIS [5] to support the automatic production of high-quality user interfaces. This is possible due to the fact that this environment is based on declarative models, with the advantages of this approach as stated in the previous section.

It is important to highlight that this user interface development process is independent of the final platform where the Graphical User Interface (GUI) is going to run. It could be a web interface, a personal computer interface or any other platform, as a decision of this kind is only made in the last phase of the methodology, i.e., the same design may be used for different implementations in different devices.

As shown in figure 1, the development process is split into two parts: On the one hand the development of the GUI is performed and on the other hand the application itself is developed.

These two independent and parallel developments take as starting point the same information: a set of use cases providing the requirements of the domain activity to be supported. Starting from this common information, on the one hand a domain model is created, that is, a conceptual object model that describes the objects identified within the system together with the relations among them.

But on the other hand, and in order to develop the GUI, we need additional information apart from the one gathered by the use cases. Use cases are not enough to describe user tasks from the point of view of the final user. Use cases describe the systems in terms of functional requirements, nevertheless, one of their main deficiencies is that they do not capture non-functional requirements, such as reliability, efficiency, maintainability and above all, usability, which undoubtedly are critical factors for final users to accept a computer application. Use cases do not capture user needs either, which are essential to develop user interfaces. For these reasons, task analysis techniques are included at this point. Starting from task analysis, the User Interface Model is built in the way described below. The relationship between use cases and the tasks identified with the task analysis technique is one to many, this is, one use case may correspond to one or many tasks. So, the system functionality described by a use case is equivalent to one or many user tasks.

All in all, the approach we propose is to take a use-case-driven development process adding task analysis techniques to enable the specification of the user interface. These techniques mainly help to define who the users of the system are, what tasks should they perform, what the main goal of those tasks is, and so on.

On the other hand (the right side in figure 1), the design of the system supporting the solution is developed. In many ways these two designs (the user interface and the application) specify two separate but interdependent systems.

In IDEAS, the declarative models composing the user interface model are the following: Task, User, Domain, Dialogue and Presentation Models. In next section, these models will be defined following the different stages of the development process.

3.1 Development Process

The user interface development process proposed in IDEAS is depicted in figure 2.

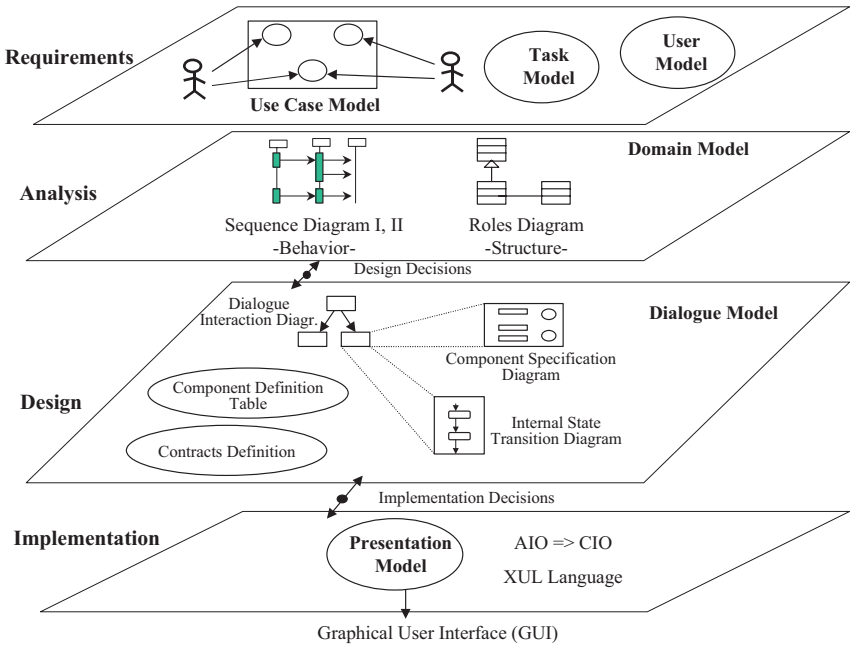


Fig. 2. User Interface Development Process.

At Requirements Level three models are created: the Use Case Model, the Task Model and the User Model.

Firstly, the use cases technique is used. In this first stage and taking into account the use cases, the different kinds of users are identified. Subsequently, each use case is refined and the business rules, entities and actors that participate in it are specified.

The Task Model defines the ordered set of activities and actions the user has to perform to achieve a concrete purpose or goal. Artifacts are essential for task development. They are modeled as objects and represented in the domain model. There exist a strong relation between the task model and the domain model.

According to [16], a task description should include a goal, a non-empty set of actions necessary to achieve the goal, a plan for the selection of actions and a model of the artifact involved in the task development. This description of what a task is and must contain can be used to construct a task definition language. We propose a template (fig. 6), based on the one proposed by [17], to describe in natural language all these issues. The question now is what tasks should be modeled in the task model. The answer is all those tasks the final users have to perform with the system.

The User Model describes the characteristics of the different types of users. The purpose of this model is to support the creation of individual and personalized user interfaces. Firstly, it defines, for each kind of user, the set of tasks he/she can perform. Secondly, for each kind of user in relation with a concrete task, a projection on the actions within the task that he/she can perform is established. And finally, depending on the user's particular characteristics (child, adult, handicapped...), the information and the interaction established by the Dialog Model to show the information contained in the Domain Model is adapted to the user.

At analysis level the Domain Model is performed. This model consists of two diagrams. The first one is the Sequence Diagram, which defines the system behavior. The second one is the Roles Model, which defines the structure of the classes that take part in the associated sequence diagram together with the relationships among these classes, specifying the role of each one of them.

At design level, the Dialogue Model is performed [18]. All the models that have been generated up to now do not contain any graphical aspect of the final user interface. It is from now on that these issues start to be addressed and the way in which the user-system interaction will be performed is especially important. The dialogue model includes the generation of four different kinds of diagrams. Herein and due to space constraints, we highlight the Dialogue Structure Diagram and the Component Specification Diagram. The first diagram specifies the user interface behavior, this is, it represents the windows and dialogues that the user needs to complete all the tasks he/she requires from the system, and the user selections to pass from one window to another. For the generation of this diagram, we have to take into account the sequence diagram obtained in the analysis phase. The second diagram entails establishing, for every one of the windows and dialogues obtained in the first diagram, the set of control and visualization tools needed to give the user the functionality he requires performing the corresponding task. This diagram defines the state of the user interface.

At implementation level the Presentation Model is performed. This model describes the concrete interaction objects (CIOs) composing the final GUI, its design characteristics and visual dependencies among them. These CIOs are defined taking into account implementation decisions, the final implementation platform and according to the style guide followed.The final GUI has a static and a dynamic part. The first one is the standard widgets presentation and the second one shows the data dependent on the application that change at run time. In IDEAS, the final GUI generation is performed by using XUL[3], an XML based language, in order to make it as much independent as possible from the final platform where the application is going to run and so get the maximum compatibility with new web technologies.

This development process considers the three-level software architecture: boundary, control and entity as proposed in the Unified Software Development Process [19], but the scope of our proposal is to specify the boundary classes corresponding with the final graphical user interface. Users interact with these boundary classes (user interface classes) to perform their tasks. The boundary classes will be associated to the corresponding control class/es to achieve the task goal, and the entity classes correspond to long-lived and persistent information. The specification of boundary classes is the goal of user interface design whereas the specification of control and entity classes is the goal of domain analysis model. Thus, we center on boundary classes to represent the application user interface.

4. USABILITY IN USER INTERFACE DEVELOPMENT

The main aim of software construction is to help users to achieve their goals in an easy way. That is, make the system to be easy to learn, easy to manage and useful, i.e. it should contain all the functionality the users need to accomplish their tasks. All in all, the basic goal is to get a computer application with a high grade of "usability" [2].

Computer systems built with usability criteria have several benefits such as productivity improvement, cost and learning time reduction and a notable increment of final user autonomy.

[3] **X**ML-Based **U**ser Interface **L**anguage

Landauer [20] stated that 80% of maintenance costs – what represents the 80% of software development total costs – are due to user-system interaction problems and not to technical problems. Besides, he indicates that 64% of these costs are related with usability problems.

This situation highlights the importance of usability and justifies the need to incorporate usability criteria before, during and after software systems development. This means that computer application development is a multidisciplinary activity that must incorporate human factors and ergonomic techniques with the aim of improving the work conditions of final users. This implies the participation of different kind of experts, not only computer engineers but also experts on disciplines such as psychology, ergonomics and human factors and so on.

4.1. The ISO Definition of Usability

The ISO 9241 standard [21] defines usability as the "extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use".

- effectiveness: The "accuracy and completeness with which users achieve specified goals".
- efficiency: The "resources expended in relation to the accuracy and completeness with which users achieve goals".
- satisfaction: The "freedom of discomfort, and positive attitude to the use of the product".

The components and the relationships between them are illustrated in figure 3.

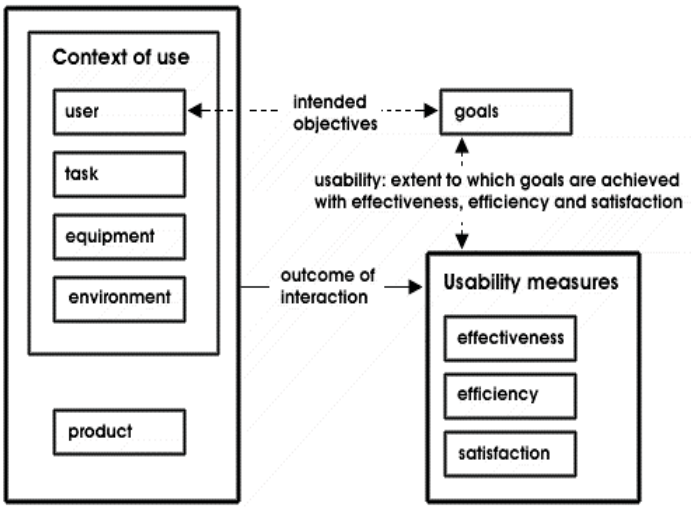


Fig. 3. Usability framework according to ISO/DIS 924111.2.

The standard states that when specifying or measuring usability, the following information is needed:

1. A description of the intended goals.
2. A description of the components of the context of use including users, tasks, equipment and environments. This may be a description of an existing context, or a specification of intended contexts. The relevant aspects of the context and the level of detail required will depend on the scope of the issues being addressed. The

description of the context needs to be sufficiently detailed so that those aspects of the context which may have a significant influence on usability could be reproduced.

3. Target or actual values of effectiveness, efficiency, and satisfaction for the intended contexts.

4.2. Usability Context of Use.

The context of use defined by the standard includes the following factors [22]:

Description of users: Characteristics of the users need to be described. These can include knowledge, skill, experience, education, training, physical attributes, and motor and sensory capabilities. It may be necessary to define the characteristics of different types of user, for example users having different levels of experience or performing different roles.

Description of tasks: Tasks are the activities undertaken to achieve a goal. Characteristics of tasks which may influence usability should be described, e.g. the frequency and the duration of the task.

Detailed descriptions of the activities and processes may be required if the description of the context is to be used as a basis for the design or valuation of details of interaction with the product. This may include description of the allocation of activities and steps between the human and technological resources. Tasks should not be described solely in terms of the functions or features provided by a product or system. Any description of the activities and steps involved in performing the task should be related to the goals which are to be achieved.

Description of equipment: The description of the hardware, software and materials may be in terms of a set of products, one or more of which may be the focus of usability specification or evaluation, or it maybe in terms of a set of attributes or performance characteristics of the hardware, software and other materials.

Description of environments: Relevant characteristics of the physical and social environment need to be described. Aspects which may need to be described include attributes of the wider technical environment (e.g. the local area network), the physical environment (e.g. workplace, furniture), the ambient environment (e.g. temperature, humidity) and the social and cultural environment (e.g. work practices, organizational structure and attitudes).

Usability measures: Usability measures include effectiveness, efficiency and satisfaction. These are measured in user trials of the product.

Goals of trial identify what the projects are trying to find out in their trials. The goal of the user trial may be to help in defining user requirements, to validate that the technology works in real conditions, to measure user attitudes, etc.

All in all, usability means that the people who use the product can do so quickly and easily to accomplish their own tasks. [Dum93].

B. Shneiderman [23] explains that, within the “usability engineering”, one of the basis to reach a high quality level is using iterative design methods. These methods permit developers to use prototypes, getting feedback from user reactions.

J. Nielsen [24] states that the best way to successfully applying usability criteria is putting the maximum effort before the design process of the interface layout starts. With this concern, the most important thing is to make a previous study that allows the designer to “know the users”. This knowledge is based not only on the study of users’ characteristics or individual abilities, but also it is necessary to know the tasks they need to carry out. In this sense, it is very important that users could perform evaluations (usability tests) upon the prototype. With this practice we can obtain improvements in next versions of the product.

4.3. Usability in IDEAS.

For the reasons stated before, our proposal includes high-level models which permit "know the users". For this matter, IDEAS includes use case, task and user models. The first ones permit to describe user tasks accurately. The last one describes the individual characteristics of the different kind of users interacting with the system.

After knowing the users and the tasks they perform, it is time to set the goals or basic usability criteria the system must reach. The definition of these criteria will allow performing the usability test associated to such criteria further on.

In last phases of the development, when designing the presentation model, the designer must take into account different heuristics and style guides to improve the usability of the layouts. Starting from the presentation models, the user interface prototype is generated. Final users evaluate these prototypes with the usability tests prepared previously.

Finally, all the previously stated phases and models are incorporated into an iterative design process. This way of doing permits to perform a feedback process by introducing the information collected in the tests performed by the users in the user interface development. This way, the usability of the designed interfaces improves notably.

In figure 4, this iterative process integrating usability is depicted.

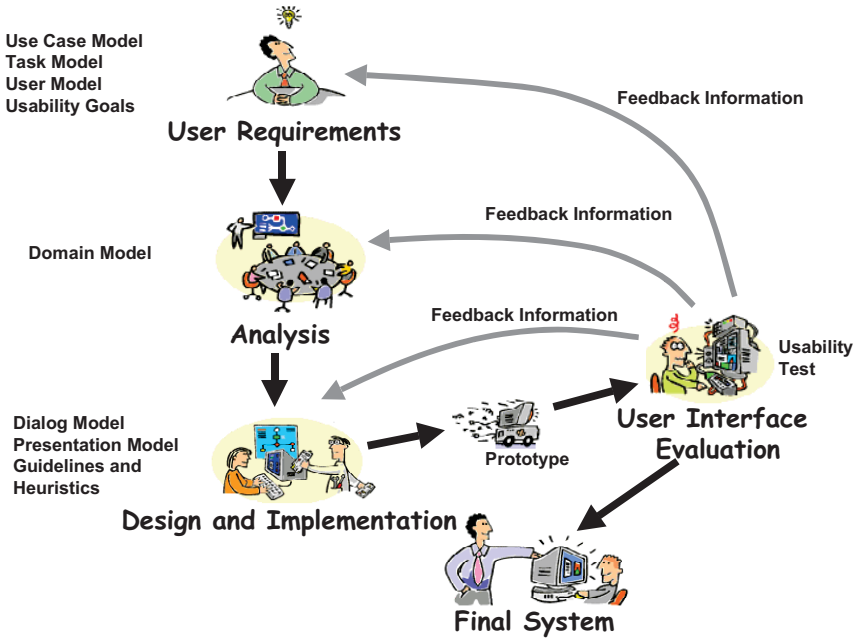


Fig. 4. Iterative development process.

5. EXAMPLE OF USE

To illustrate the proposed methodology, in this section we apply it to the simplified case of a Rental Car. The initial use case model is shown in figure 5.

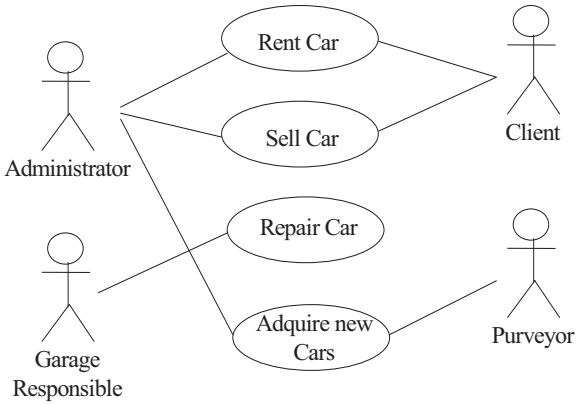


Fig. 5. Use Case Diagram.

The next step is to identify user tasks from use cases. In this case, the "Rent Car" use case corresponds to the task "Rent a Car". In this case the correspondence is one to one. The task model specification follows the template shown in figure 6.

Task Model: Template (Example: Rental Car)

Task: Rent a car.
GENERAL FEATURES:
GOAL: The administrator rents a car to the client.
PRECONDITION: - The car is available.
- The car is in good conditions.
- The client has not more cars than permitted
SUCCESS CONDITIONS: - The administrator checks the car's availability and state.
- The administrator brings the client´s state up to date.
- The client receives the solicited car.
FAILURE CONDITIONS: - The car is not available.
- The client already has remted the maximum number of cars permitted.
PRIMARY ACTORS: Client, Car.
SECONDARY ACTORS: Administrator.
TRIGGER ACTION: The client request to rent a car.
NORMAL SCENARIO:
.The client request to rent a car.
.The administrator checks the state and the availability of the requested car .
.The administrator checks that the client has not rented the top number of cars permitted.
.The administrator gives the client the car.
.The client receives the requested car.
VARIANTS:
.The car is not available.
.The car is not in good state.
.The client already has the top number of cars permitted to rent.
RELATED INFORMATION:
Priority: High
Duration: 15 mins.
Frequency: 10/day.

Fig. 6. Task Model Template.

The Domain Model uses sequence diagrams [25] to model the system behavior and roles model to describe the system structure. This is depicted in figure 7:

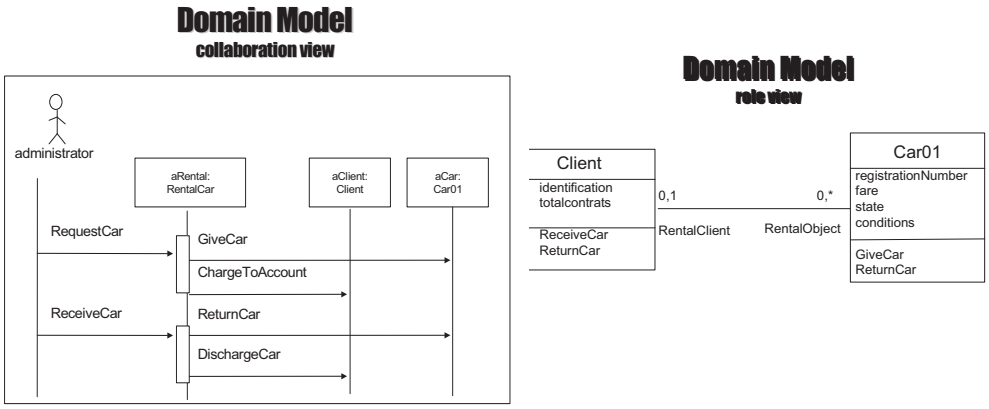


Fig. 7. Domain Model.

The Dialog Model includes the generation of three different kinds of diagrams: the Dialog Structure Diagram, the Component Specification Diagram and the Component Behavior Definition Table. For the Rent Car case study these diagrams are shown in figure 8.

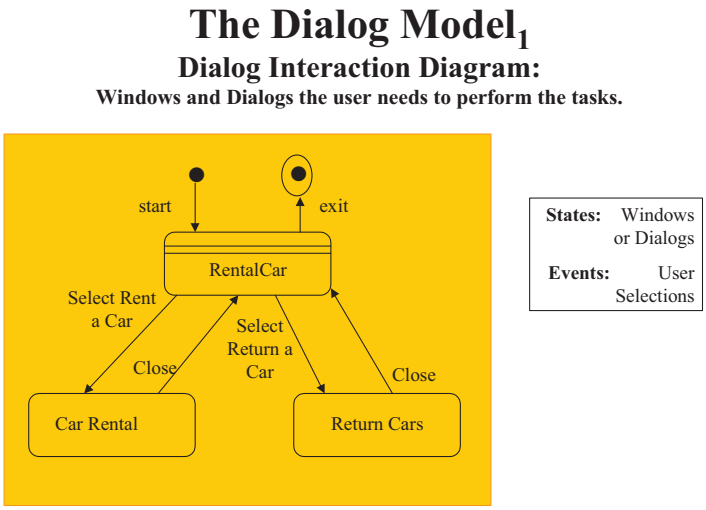
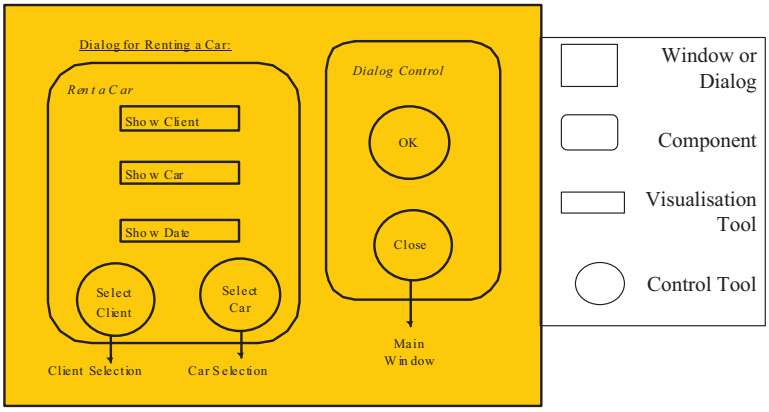


Fig. 8. Dialog Model.

The Dialog Model$_2$

Component Specification Diagram:

Models the contents of every one of the windows and dialogs represented in the previous diagram.



The Dialog Model$_3$

Component Functional Model:

Table in which every one of the components and tools represented in the previous diagram are described in detail.

Component	Control Tool	Visualisation Tool
Rent a Car	***Select_Client***: Allows the selection of a Client or the introduction of a new one. ***Select_Car***: Allows the selection of a car from the database.	***Show_Client***: Shows the client selected through the control tool. ***Show_Car***: Shows the car selected through the control tool. ***Show_date***: Shows the system date corresponding to the rental date.
Dialog Control	***Ok*:** Allows the user to introduce a new rental in the system. ***Close*:** Close the dialog and returns the user to the main window.	———

Fig. 8. Dialog Model (Cont.)

Finally, the Presentation Model is built from the dialog model taking into account style guides. This is, the abstract interaction objects modeled in the dialog model are now translated into the concrete interaction objects conforming the GUI.

6. CONCLUSIONS

In this paper we propose a user interface development environment within the object-oriented software development process. This development process is performed applying use cases and task analysis techniques, from which a user interface model is built. This model consists of five declarative models that describe different aspects involved in user interface construction.

It has been also described the way to incorporate usability criteria in order to achieve high usable user interfaces. Finally, we have stated an iterative design process that permits the incorporation of the consecutive improvements the users introduce in the design process.

And finally, remark the more and more importance that is getting the user interface, above all in interactive systems, as a means of interacting with a system and taking the most profit of it. This is the main motivation of this research.

7. REFERENCES

[1] B.A. Myers: Why are Human-Computer Interfaces Difficult to Design and Implement. Tech. Report CMU-CS-93-183, CMU, Jul 1993.

[2] J. Gould, C. Lewis: Designing for Usability – Key Principles and What Designers Think. Commun. ACM, Vol 28, 1985, pp.300-311.

[3] W. Swartout, R. Blazaer: The Inevitable Intertwining of Specification and Implementation. Commun ACM, Vol 25, No. 7, Jul 1982, pp.438-440.

[4] O. Pastor, I. Ramos. OASIS 2.1.1.: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach. Octubre 95 (3 ed.)

[5] P. Letelier, I. Ramos, P. Sanchez, O. Pastor. OASIS versión 3.0: A Formal Approach for Object Oriented Conceptual Modelling. SPUPV-98.4011. Edited by the Technical University of Valencia, Spain, 1998.

[6] B. A. Myers: User Interface Software Tools. ACM Transactions on Computer-Human Interaction 2 (1995), 1, 64-103

[7] J. Foley, P. Sukaviriya: History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In F. Paterno (ed.): Interactive Systems: Design, Specification and Verification. Berlín: Springer, 1995, 3-14.

[8] P. Szekely et al: Beyond Interface Builders: Model-Based Interface Tools. In: Bridges between Worlds. Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 383-390.

[9] A. Puerta: The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 19-36.

[10] R. Neches et al.: Knowledgeable Development Environments Using Shared Design Models. In: Proceedings of the 1993 International Workshop on Intelligent User Interfaces (Orlando, Enero 1993). New York: ACM Press, 1993, 63-70.

[11] P. Szekely, et al.: Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach. In: Engineering for Human-Computer Interaction. Proceedings of the IFIP Conference on Engineering for Human-Computer Interaction (Yellowstone Park, Agosto 1995). London: Chapman &Hall, 1996, 120-150

[12] F. Bodart et al.: Towards a Systematic Building of Software Architectures: teh TRIDENT Methodological Guide. In: Design, Specification and Verification of Interactive Systems. Wien: Springer, 1995, 262-278.

[13] F. Lonczewski: The FUSE System: An Integrated User Interface Design Environment. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 37-56.

[14] H. Balzert et al.: The JANUS application Development Environment Generating More than the User Interface. In: Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 183-205.

[15] M. Lozano: A Methodological Approach for the Specification and Development of Object Oriented User Interfaces. Ph.D. Thesis. Department of Information Systems and Computation. Valencia University of Technology. 2001.

[16] G. Storrs: The Notion of Task in Human-Computer Interaction. In: People and Computers X. Proceeding British HCI'95 (Huddersfield UK, Agosto 1995). Cambridge: Cambridge University Press, 1995, 357-365.

[17] A. Cockburn: Structuring Use Cases with Goals. Journal of Object-Oriented Programming, 10(5) y 10(7), 1997.
http://members.aol.com/acockburn/papers/usecases.htm

[18] M. Lozano, I. Ramos, P. González. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development. 34th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-USA 2000. Santa Barbara, CA. 30 July - 3 August, 2000.

[19] I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process. Addison Wesley. 1998.

[20] Landauer, T.K. "The Trouble with Computers: Usefulness, Usability and Productivity. MIT Press, 1995.

[21] ISO/DIS 9241: Ergonomic Requirements for Office Work with Visual Display Terminals (VDT's). Geneva: International Standards Organization. 1998.

[22] Concejero, P. Clarke, A., Ramos, R. "ACTS Usability Evaluation Guideline" Report of USINACTS Usability in Acts. June, 1999.

[23] B. Shneiderman. Designing the User Interface. Strategies for Effective Human-Computer Interaction. Addison Wesley, 1998.

[24] J. Nielsen. Usability Engineering. Morgan Kaufmann, 1993.

[25] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language. Addison Wesley. 2000.

Designing Web Sites by Using Design Patterns

Francisco Montero, María Lozano, Pascual González, Isidro Ramos

LoUISE Research Group
Computer Science High School
University of Castilla–La Mancha
Campus Universitario
02071 – Albacete – Spain
http://www.info-ab.uclm.es/louise
{fmontero, mlozano, pgonzalez}@info-ab.uclm.es

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46071 Valencia - Spain
iramos@dsic.upv.es

Abstract

This paper contains a resumed collection of patterns for designing web sites. Traditionally, the Web is mainled seen as a medium used to exchange information and that is the main reason why most web sites are designed like a book in which you can jump back and forth. Usability criteria should be considered when we are developing a web site [Nie00]. This set of patterns is established under these criteria.

Introduction

The World Wide Web has rapidly become the dominant Internet tool, combining hypertext and multimedia to provide a network of multidisciplinary resources. It is important to make sure that all parts of a web site are useful. A user will come to a site expecting to be able to perform a particular task, or read a particular piece of information. When we are designing a web site we want to make sure that the user can find that resource quickly and easily. If they can't find the information quickly then they may leave our site, and proceed to another site where they can find the resource. The Web is a new medium and requires a new approach [Nie00].

We should begin by asking, as in any user interface design process, Who are the users? and What are the tasks? But answer these questions is not easy. Anybody can visit our web site, kids, seniors, older or disabilities people. All information and resources should be accessible to them [Con01]. Everybody should be able to navigate with no problem. The idea is that, no matter what you're doing, there's a user-centred way of doing it. Users should be considered throughout the web site design process. Usability should not be an afterthought. Testing and fixing a web site *after* it has been built is inefficient and unlikely to produce good results.

Shneiderman [Shn98] commented that “It will take a decade until sufficient experience, experimentation, and hypothesis testing clarify issues” and warned that meanwhile “the paucity of empirical data to validate or sharpen insight mean that some guidelines are misleading”. Nevertheless, many sets of web design guidelines have been published. There are many guidelines [Guides] that can be used to improve the design of our web sites. Most of these recommendations for web site designers are however not based on research but on intuition. They are based in the designer’s experience. Traditionally, interface design experiences are gathered with guidelines but patterns can be used too. The concept of a pattern language has been developed by Christopher Alexander and his colleagues in architecture and urban design [Ale77, Ale79]. In brief, a pattern language is a network of

Copyright © 2002, Francisco Montero. Permission is granted to copy for the SugarloafPLoP 2002 Conference. All other rights reserved.

Montero, F. Lozano, M. D., González, P., Ramos, I. Designing Web Sites by Using Design Patterns. Proc. of SugarLoafPLoP 2002. 2002.

patterns of varying scales; each pattern is embodied as a concrete prototype, and is related to larger scale patterns, which it supports, and to smaller scale patterns which support it. The goal of a pattern language is to capture patterns in their contexts, and to provide a mechanism for understanding the non-local consequences of design decisions [Eri97].

A Web Design Pattern Language

This pattern language describes the usability-desired result unlike a prescriptive pattern language or guidelines. The patterns in this language are grouped into three levels.

- **Web site** level, people expect a web site to provide information, be interactive and fulfill their requirements. web sites have to be visually pleasing, download quickly, be helpful, easy to use and explore, in addition to presenting a professional appearance. A web site is made up of a collection of documents, images, sounds and other files. These usually reside in a unique place on the Internet. We can find this collection of web pages because they have an address.
- **Web page** level, a web page is a single page on any specific web site. A page is specially important, the home page, due to it identifies the web site.
- **Ornamentation** level. There are several elements that usually are on a web page. These elements give useful support to user for performing his/her tasks. These elements improve usability of the web site.

This taxonomy of three levels is based on [Ale77]. The Alexander's language contains 253 patterns split into three broad categories, towns, buildings and construction. The first 94 patterns are collected together under the heading of "Towns". However, they cover far more than just urban planning and begin by examining the responsibilty of the designer from a global perspective. Moving on from the study of areas and collections of buildings, Alexander examines the buildings which make up the urban landscape and suggests over 100 patterns which define his position on what constitutes good building design. Alexander finishes his collection of patterns by defining the right way to construct buildings. The patterns in this final section of the language show how the need for careful thought and creative effort are required throughout the design process. The main objective of Alexander was improving *a quality without name* and planning cities, towns and buildings. Alexander wanted to create structures that are good for people and have a positive influence on them by improving their comfort and their quality of life. When we are developing a web site that's our goal too. We are interested in designing web sites with quality. But what is quality?. Quality is perhaps usability, understandability, learnability, operability, adaptability, accessibility, etc. We think that quality is everything mixed together.

The proposed web pattern language will be introduced by using a particular example. The patterns in the entire collection are depicted graphically in Figure 1 and summarised at the end of this section by using three tables, one of them for each level of patterns.

A problem is presented by each pattern and, under a context and a set of forces, a solution is proposed [Mes94]. Problems are associated with user requirements and solutions look for improving usability web site. Usability is [ISO/IEC 9126-1] learnability, understandability and operability and these elements are improving by providing navigation, functionality, control, language, feedback, consistency, error prevention and visual clarity facilities.

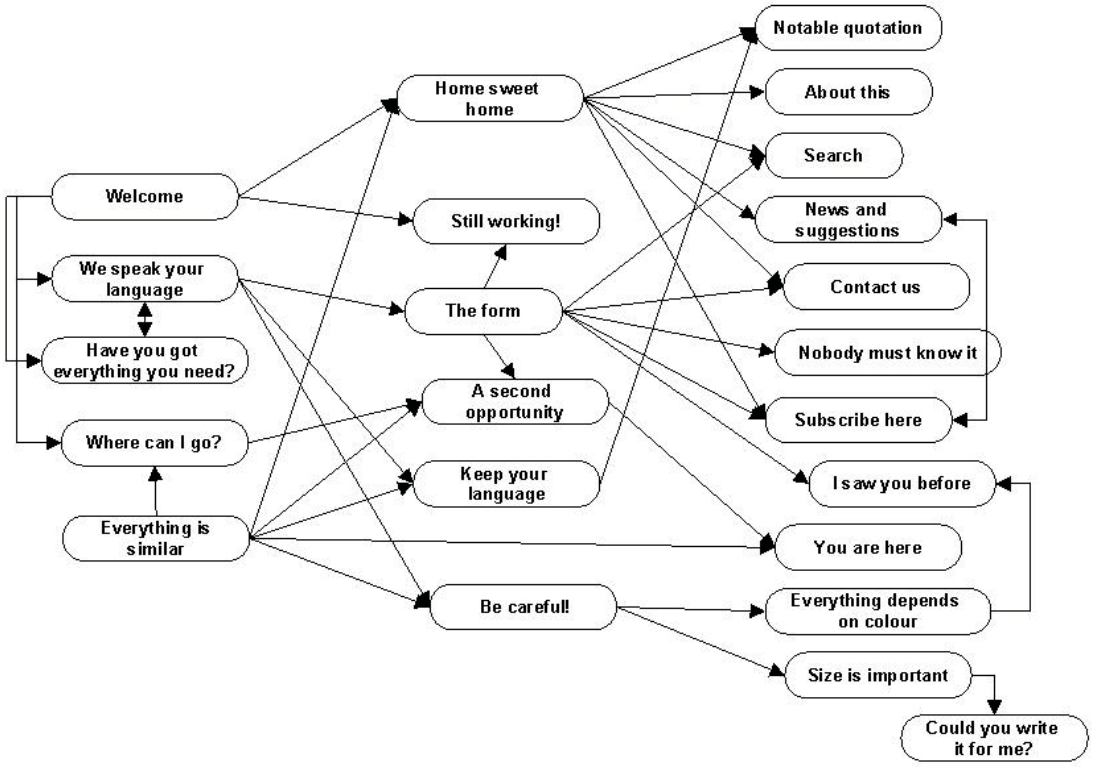


Figure 1. Proposed pattern language

Figure 1 shows the proposed pattern language. It has a structure of a network, in the left column there are patterns of web site level, in the centre column there are patterns of web page level and, finally, in the right column there are patterns of ornamentation level.

The following tables summarise the patterns in this pattern language for reference purposes. These patterns could be integrated on a methodology to develop user interfaces like IDEAS [Loz01]. There are patterns at requirements level, like these, that can be used in the beginning of an usability-based iterative life cycle. So patterns can be used to improve a participatory design, evaluate web site under usability criteria and facilitate communication between stakeholders involved in web site development.

Web site patterns		
Problem	**Solution**	**Pattern name**
How does the user know where he is?	*Set up a reception point where user finds information about the web site*	Welcome
How does the user know where he can go and what will he find there?	*Web site musts provide the needs mechanisms that allow any user to move from one place to another places*	Where can I go?
How can the user do a useful use of the web site and access information at your own pace?	*Speak user's language is "design for all"*	We speak your language
How does the user know	*Web site should be designed*	Everything is similar

where he is? How does the user know that he is visiting the same web site?	*by using the same criteria: colours, fonts, navigation location and layout*.	
How does the user know if he needs anything else for visiting a web site?	*The user should be informed of what he need for visiting a web site*.	Have you got everything you need?

Web page patterns		
Problem	**Solution**	**Pattern name**
How does the user know where the user is?	*Provide a checkpoint where the user feels like at home*.	Home sweet home
How can the user access the content of the web page in a simple and proper way?	*Keep your language clean and inoffensive*.	Keep your language
How does the user know when his operations have finished or what is their current state?	*Show the user a status information of some kind, indicating how far along the process is in real time*.	Still working
How can the user visit the web site to his/her own pace?	*Provide return approaches*.	A second opportunity
How can the user provide preformatted information?	*Provide appropriate "blanks" to be filled in, which clearly and correctly indicate what information should be provided.*	The form
How can the user visit the web site without be interrupted or disoriented by unnecessary effects?	*Design for all people, technology and knowledge level in a possible manner*.	Be careful!

Ornamentation patterns		
Problem	**Solution**	**Pattern name**
How does the user know what the main feature of the web site owner is?	*Include a tag line that explicitly summarise what the site or company does*.	Notable quotation
How can the user get a suitable print of information?	*Provide a text version of web pages directly printable*	Could you write it for me?
How can the user have got access to additional and periodic information?	*Provide an approach to user cans book on-line*	Subscribe here
How can the user get additional information on products or documents?	*Include a "Contact Us" link on the homepage*	Contact us
How can the user find specified information?	*Offer a search engine*	Search
How does the user know	*Use an approach to*	I saw you before

where he/she has been?	*maintain state variables on the Web, like cookies*	
How can the user access to the content of web site in a suitable way?	*Use suitably colours*	Everything depends on colour...
How can the user access the content of web site in a suitable way?	*Provide on-demand*	Size is important
How can provide private information in secure way?	*Implement security in web site*	Nobody must know it!
How does the user know what are the news in the web site?	*Include at homepage news and suggestions sections*	News and suggestions
How does the user know where he/she is?	*Include location references in web site*	You are here
How does the user know who the owner of web site is?	*Include a link to an "About Us" section*	About this

In the next section, we will describe some of these patterns by using them to solve problems related with the creation of a web site where usability degree should be high. This task helps us to introduce some of these patterns with more detail than this section.

When each pattern is introduced a last field, examples and implementation details, provides references (urls) and brief considerations on how this pattern can be recognised when we are navigating across the Web.

Example of use: Applying the Web Pattern Language

Suppose than you have to promote an institution, a business or a research group. An interesting possibility could be the creation of a web site where users can find out about services, products, latest news supplied by that institution.

You are a beginner designer and know that having a good web site is important, but you have not got enough experience in web design. There are guidelines, experiences and knowledge gathered in web site design field but it is difficult to use and in some occasions can be contradictory. However, a language is the most powerful tool of communication, but in a language there are words, sentences and relationships between words and sentences. On the other hand, patterns are a way of documenting design expertise. A pattern language can be useful in two senses: firstly as a tool to document the experience and secondly, as a tool to communicate between stakeholders of the project (end-user included).

How could I use this collection?
1. Read the resumed list of patterns.
2. Scan down the list, and find the pattern, which best describes the overall scope of the project or the problem that you want to solve.
3. Read the starting pattern. Tick all of the low order patterns and ignore all the high order patterns.
4. Turn to each pattern and now tick only relevant low order patterns.
5. Keep going like this, until you have ticked all the patterns you want for your project.

6. Adjust the sequence by adding your own material where you haven't found a corresponding pattern.
7. Change any patterns where you have a personal version, which is more relevant.

As an example, and after of reading the list of patterns, we could select some of them, for instance, Welcome, Home sweet home, Notable quotation, About this, Search, News and suggestions, Contact us, or Subscribe here. These patterns are related like shown figure 1 and they are of different levels. Then we can read them in more detail.

Welcome

Motivation:

When a user arrives at a web site, like he/she arrives at a city, town or any important building needs to know where he/she is, what can he do there, and what he need for visiting that web site.

Problem:

How does the user know where he/she is? How does the user know where he can go? Who does the site manage? What is the purpose of the site?

Forces:

- Users need know where they are
- User wants to know where they can go next
- A complex web site can be very disorienting for users
- Users who are familiar with the structure and content of a web site should be able to jump straight to the space where they want to go

Solution:

Set up a reception point where the user finds information about the web site. From this welcome point, user will be able to enter to home page (Home sweet home). User's information, such as language or monitor size should be gathered to the provision of web site's services to user (We speak your language). In its defect, the user should be informed about the best conditions for the visiting web site (Have you got everything you need?). User finds information about content (About this) and owner (Contact us) of the web site in this page. Welcome and homepage is the same in many occasions.

Consequences:

Provide improvements on the navigation, functionality and feedback

Examples & implementation details: http://www.aosa.es, http://www.alanismorissette.com These web sites, and many others on the Web, have got a initial page where users are received. These web pages have as main features their low-load time, offer the possibility to customise the language or browser properties, and provide information on who, what, when and where the user can find on the web site.

Home Sweet Home

Motivation:

A web site can be achieved by random way, but always must have a point of reference. When an user arrives at a web site, like he/she arrives at a city, town or any important building needs to know where he/she is, what he can do there, and what he need for visiting that web site. The homepage is an essential component of a web site. Questions such as: who?, what?, when? and where? should have answer on it.

Problem:

How does the user know where the user is? How does the user know where he will go? Who does the site manage? What is the purpose of the site?

Forces:

- Users need know where they are
- User wants to know where they can go next
- A complex web site can be very disorienting for users
- Users who are familiar with the structure and content of a web site should can jump straight to the space where they want to go

Solution:

Provide a checkpoint where the user feels like at home. Homepage is a place where the user can go back if he is disoriented. Its layout puts important information at top (News and suggestions), includes logos (Notable quotation), search approaches (Search) and information contact (Contact us, About this, Subscribe here) .

Consequences:

Provide improvements on the functionality, control and navigation

Examples & Implementation details: http://www.apple.com, http://www.ieee.org Any web site has a homepage. It is a specific page that introduces distinctive features. It has links to different sections of the web site, such as news, contact, about us, search etc. References to homepage should be included in every pages of the web site (A second opportunity).



Notable quotation

Motivation:

When you are designing a web site you should provide information about its purpose.

Problem:

How does the user knows what is the main feature of the web site owner?

Forces:

- Users have hurry
- Users don't read web pages, they have a look at pages

Solution:

Include a tag line that explicitly summarise what the site or company does. Its should be brief, simple and to the point. Include a short description of the site in the window.

Consequences:

Provide improvements on visual clarity, functionality and feedback

Examples & Implementation details: http://www.coolhomepages.com, http://www.bbva.es These web pages has images or taglines that implements this pattern. A tagline is a short phrase that communicates the "who" and "why' of your Web site.
The following elements create effective taglines: subject + audience + organization.



About this

Motivation:

All business web sites need to provide a clear way to find information about the company no matter how big or small the company is.

Problem:

How does the user know who the owner of web site is?

Forces:

- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know who is behind the service

Solution:

Include a link to an "About Us" section that gives users an overview about the web site owner and links to any relevant details about your products, services, company values, business proposition, management team, and so forth.

Consequences:

Provide improvements on functionality and feedback.

Examples & Implementation details: http://www.sunspot.net, http://www.ireland.com This pattern is implemented by adding a page or a section where information about owner of the page can be found. Normally a link to this section is situated in the homepage.

Search

Motivation:

Search is one of the most important elements of a homepage (Home sweet home), and it is essential that users be able to find it easily and use it effortlessly.

Problem:

How can the user find specified information?

Forces:

- User wants to know if the searched information is on the web site
- User doesn't read web site. He/she has a look at it.

Solution:

Offer a search engine. Give users an input box on the homepage to enter search queries, instead of just giving them a link to a search page [Nie02]. Search on the homepage should search the entire site default. [see Wel01]

Consequences:

Provide improvements on functionality and control.

Examples & Implementation details: http://www.paginasamarillas.es, http://www.microsoft.com This pattern is implemented by providing a search form. Search forms are the user interface of the search engine. It can consist on a very simple form with just a text field and a button, it can be a page and add a lint to it in your navigation. Advanced search capabilities can be worth adding. An advanced search page with options for phrases, multiple fields, special collections or zones, and date ranges allows them to perform more precise searches.



News and suggestions

Motivation:

Users want to know if there are new features in the web site. Users admit suggestions and want to know offers and promotions.

Problem:

How does the user know what the news in the web site are?

Forces:

- User doesn't read web site. He/she has a look at it
- Users have hurry

Solution:

Include news and suggestions sections at the homepage where users will have rapid access to new services offered by Web site

Consequences:

Provide improvements on functionality and navigation

Examples & Implementation details: http://www.microsoft.com, http://www.terra.es This pattern is implemented by placing latest news or suggestions in an outstanding place in the homepage (Subscribe here).

Contact us

Motivation:

All business web sites need to provide a clear way to contact with web site owner.

Problem:

How can the user get additional information on products or documents?

Forces:

- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know how is behind the service

Solution:

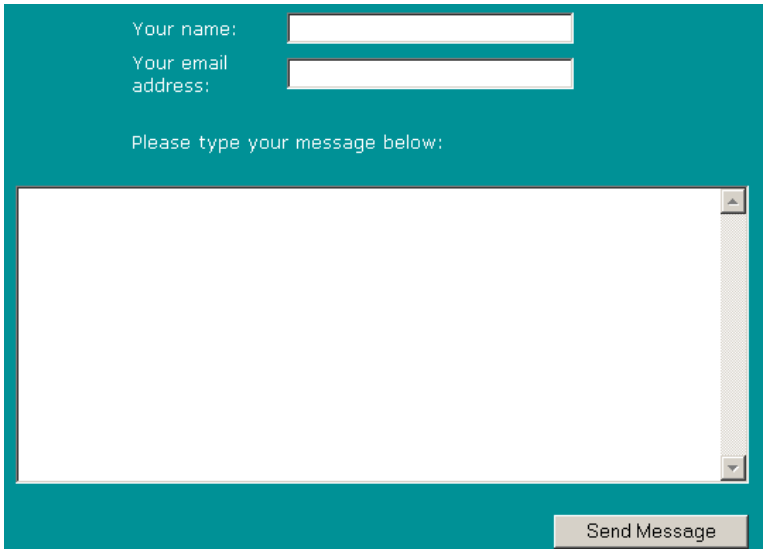
Include a "Contact Us" link on the homepage that goes to a page with all contact information for your company (About this).

Consequences:

Provide improvements on feedback.

Examples & Implementation details:

http://www.intel.com, http://www.lucent.com This pattern is implemented by including a page or a section where user can find contact information, in many occasions this information is included at the bottom of all the pages of the web site. In others cases, a form is provided to the user. This form contains features like a text area or text fields for the user can provide his email and other comments.



Subscribe here

Motivation:

Users want not to visit a web site everyday, they prefer to be informed when new products or news arrive.

Problem:

How can the user have got access to additional and periodic information?

Forces:

- User has hurry
- User wants to be informed

Solution:

Provide an approach where users can book on-line by providing an e-mail. So, the web site owner can send information to registered users about news and suggestions (News and suggestions).

Consequences:
Provide improvements on feedback.

Examples & Implementation details: http://www.prenhall.com, http://www.sun.es This pattern is implemented by using a simple form where user usually only have to provide an email. In other occasions is necessary provide more information related with the user's profile and preferences, so it is possible provide personalised information. Unsubscribe option should be provided too.



In this moment, we have a departure point to start the development of the web site. With a language, the redaction of pattern includes references to others patterns of low order. If in this moment, our beginner designer identifies the need of getting information from the user, could read The form pattern and related ones – Still working, Search, Contact us, Subscribe here, Nobody must know it and I saw you before. Next, some of them are explained

The form

Motivation:
The user has to provide information, usually short answers to questions

Problem:
How can the user provide preformatted information?

Forces:
- The user needs to know what kind of information to provide
- Users generally do not enjoy supplying information this way
- It should be clear what is required, and what is optional
- The user has hurry

Solution:
Provide appropriate "blanks" to be filled in, which clearly and correctly indicate what information should be provided [Tid98]. Search, Subscribe here, Contact us are examples of forms. In occasions, a form fills a complete page. The user needs know if his/her submit was correctly processed (Still working). In some situations, we need to get confidential information of the users then must to provide additional security (Nobody must know it!)

Consequences:
Provide improvements on the functionality

Examples & Implementation details: http://www.iomega.com, http://www.iberia.es These web pages and some others like them, where the user can provide information implement this pattern. An HTML form is a section of a document

containing normal content, markup, special elements called *controls* (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally complete a form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a web server, to a mail server, etc.)



Particular examples of this patterns are Contact us, Subscribe here.

Still Working

Motivation:

Web sites are places where users can download information, images, files or applications, but this downloading can take a lot of time, create significant delays or be accomplished in different ways.

Problem:

How does the user know when his/her operations have finished or what is their current state?

Forces:

- The user wants to know how long they have to wait for the process to end
- The user wants to know how fast the progress is being made, especially if the speed varies
- Sometimes its impossible to tell how long the process is going to take

Solution:

Show the user a status information of some kind, indicating how far along the process is in real time. Images, files and any element that the user can download should have got information about size, so users can know how long have to wait for the download process. Images and text should be downloaded on-demand (Size is important).

Consequences:

Provide improvements on the functionality, feedback and error prevention

Examples & Implementation details:

http://www.google.com, http://www.acrobat.com Many web pages needs load a plug-ing to get a correct visualisation, a progress bar is used to provide such information. Sometimes a task running within a web site might take a while to complete. A web page with usability provides some indication to the user about how long the task might take and how much work has already been done. If you don't know or don't want to indicate how complete the task is, you can use a cursor or an animated image to indicate that some work is occurring. If, on the other hand, you want to convey how complete the task is, then you can use a progress bar like this one (http://java.sun.com):

Motivation:

There is a plethora of plug-ins for sound, animation and all kinds of things. But you can't assume that anyone is going to have them, or can use them with their particular computer set-up.

Problem:

How can the user visit a web site without getting confused, being interrupted or being disoriented?

Forces:

- Users generally have not got the plug-in that he/she needs
- It should be clear what is required to visit the Web site
- Users that can visit Web site is unknown
- Everybody is disability in one way or another

Solution:

Be careful with using disoriented components. For example, you can use readable font size, consider monitor size, use well-designed headings, limit number of frames, limit use of animated gifs, flash, applets, music, rollovers, reduce user's workload, not use blink or marquee elements, limit maximum page size (**Size**, **Colour**). Use style sheets to control layout and presentation.

Consequences:

Provide improvements on visual clarity, control, functionality and navigation.

Examples & Implementation details:

The fundamental design of the Web is based on having the page as the atomic unit of information, and the notion of the page permeates all aspects of the Web [Nielsen, 00].

http://www.garbage.com, http://www.geocities.com/webterocks are bad examples, antipatterns in this respect. Frames pose many problems, for instance, navigation does not work with frames since the unit of navigation is different from the unit of view. Frames sites are either hard or impossible for search engines to index. If you use frames, people will have a hard time finding your Web site. Frames cause printing issues on older browsers, which tend to print the frame clicked in last, or if not default to top-left, which is generally not the frame you want to print.

The BLINK and MARQUEE elements are not defined in any W3C specification and should not be used".

Internet is a great search engine, and Macromedia Flash is a great visual impact tool but search engines cannot read the text within a Flash 'Movie' or within any other image files like GIF's or JPEG's.

Ornamentation Level

This section introduced decoration features of a Web site. These features provide improvements on general usability of any Web site. They are related with using of **Colour**, **Size**, security (**Secret**) and providing location references (**Location**, **Contact us**) and information (**Subscription**, **Recognize**, **Novelty**).

Tagline

Motivation:

When you are designing a web site you should provide information about the purpose of it.

Problem:

How can the user know the purpose of the web site?

Forces:

- Users are in a hurry
- Users don't read web pages, they have a look at pages

Solution:

Include a tagline that explicitly summarise what the site or company does. Its should be brief, simple and to the point. Include a short description of the site in the window.

Consequences:

Provide improvements on visual clarity, functionality and feedback

Examples & Implementation details: http://www.coolhomepages.com, http://www.bbva.es These pages has images or taglines that implements this pattern. A tagline is a short phrase that communicates the "who" and "why' of your Web site. The following elements create effective taglines: subject + audience + organization.



Print

Motivation:

Most people read online text differently from how they read printed texts rather than reading word by word, most people quickly scan blocks of online text. When reading text on pages within a Web site, most people also move quickly among pages

Problem:

How can the user get a suitable print of information?

Forces:

- Readers appreciate short chunks that can be located quickly
- Most users either save documents to disk or print them out

Solution:

Provide a text version of web pages directly printable, or offer a downloadable, formatted version of the document to be printed [Lyardet 00].

Consequences:

Provide improvements on functionality and control.

Examples & Implementation details: http://www.borland.es, http://www.sport.es This pattern is used when content of long documents is broken up into smaller chunks and linked. Then providing one large page for printing, a file to download (.pdf, .ps, .doc) or the ability to print all of a sectioned document in one step is useful to the user.



Subscription

Motivation:

Users want not to visit a web site everyday, they prefer to be informed when new products or news arrive.

Problem:

How can the user be informed with meaningful information for him? How can the user have got access to periodic information?

Forces:

- User is in a hurry
- User wants to be informed

Solution:

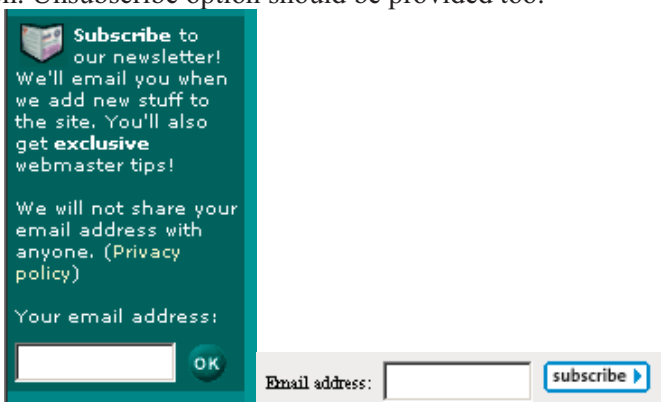
Provide an approach to user can book on-line by providing an email. So Web site owner can send information to registered users about **Novelties**. The user should be sure that your email address is not shared with anyone (**Secret**).

Consequences:

Provide improvements on feedback.

Examples & Implementation details:

http://www.prenhall.com, http://www.sun.es This pattern is implemented by using a simple **Form** where user usually only have to provide an email. In other occasions is necessary provide more information related with preferences of the user and your profile, so is possible provide personalised information. Unsubscribe option should be provided too.



Contact us

Motivation:

All business Web sites need to provide a clear way to contact with Web site owner.

Problem:

How can the user get additional information on products or documents?

Forces:

- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know how is behind the service

Solution:

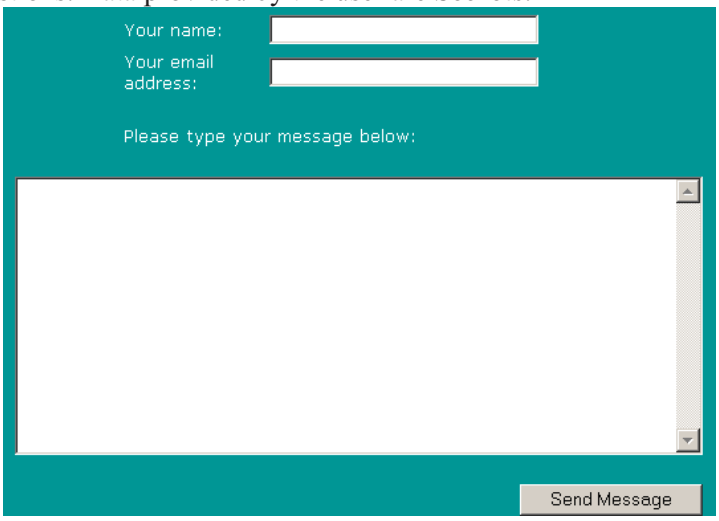
*Provide a **Form**, a place or a link in the web site where the user can get additional information about the web site owner and his poducts.*

Consequences:

Provide improvements on feedback.

Examples & Implementation details:

http://www.intel.com, http://www.lucent.com This pattern is implemented by including a page or a section where user can find contact information, in many occasions this information is included in bottom of all pages of the web site. In others cases, a form is provided to the user. This **form** contains features like a textarea or textfields in order to the user can provide his email and questions. Data provided by the user are **Secrets**.



Search

Motivation:

Search is one of the most important elements of a Homepage, and it's essential that users be able to find it easily and use it effortlessly.

Problem:

How can the user know if a web site can provide specified information that he wants?

Forces:

- User wants to know if the searched information is on the Web site
- User doesn't read web site. He/she has a look at it.

Solution:

Provide a search engine or overview page. Give users an input box on the **Homepage** to enter search queries, instead of just giving them a link to a search page [Nielsen 02]. Search on the homepage should search the entire site default. [see Welie's patterns 01]

Consequences:

Provide improvements on functionality and control.

Examples & Implementation details:

http://www.paginasamarillas.es, http://www.microsoft.com. This pattern is implemented by providing a search form. Search forms are the user interface of the search engine. It can consist of a very simple **Form** with just a text field and a button, it can be a page and add a lint to it in your navigation. Advanced search capabilities can be worth adding. An advanced search page with options for phrases, multiple fields, special collections or zones, and date ranges allows them to perform more precise searches.



Recognize

Motivation:

When a user comes back to a Web site he needs know what places he has visited, what documents he has downloaded and if there are modifications from last visit.

Problem:

How does the user know where he/she has been?

Forces:

- User does not want to loose his time
- User wants to receive personalised information

Solution:

Keep information about user actions, visited places, logins, etc, for instance by using cookies. Since HTTP is a non-persistent protocol, it is impossible to differentiate between visits to a web site, unless the server can somehow mark a visitor.

Consequences:

Provide improvements on feedback and error prevention

Examples & Implementation details:

http://www.kinkos.com, http://www.americanairlines.com. This pattern is implemented by using cookies. Web cookies are simply bits of software placed on your computer when you browse Web sites, so the web site will recognise the user's computer when he comes back to visit again. Cookies have some beneficial things. For example, when you log on or purchase online to certain sites, did you ever notice that when you return again you do not have to sign on the next time? That's because it stored your password and id on your machine in a cookie. So user's workload is reduced.

Colour

Motivation:

Many web designers overlook the importance of colour when designing a web site. Colour should be one of your first concerns when it comes time to start your web site design.

Problem:

How can the user access to information in a suitable way?

Forces:

- Web browsers can only see 256 colours
- People reading light characters on dark backgrounds for long periods reported less visual fatigue

Solution:

Provide information by using suitable colours in fonts, backgrounds and image. For example, the default colours for Web page links are blue for non-visited links and purple for visited links. Ensure that foreground and background colour combinations provide sufficient contrast when viewed by someone having colour deficits or when viewed on a back and white screen. You should use yellow and red colours sparingly in your Web site itself. Only use them in areas where you want the visitor to focus on. Do not make large parts of your web site with bright colour.

Consequences:

Provide improvements on feedback, functionality, consistency and visual clarity

Examples & Implementation details:

http://www.biblioteca.uclm.es, http://www.trashclub.com. Initially, one of the limiting aspects of designing for the Web can be the 216-colour palette. The idea behind the Netscape-created colour-set was to maintain a consistent appearance for Web pages viewed on a Windows, Macintosh, or Unix machine.

To create a consistent Web site, it's best to base the site's colours on the Websafe ones. Luckily, most HTML editors now have that palette built in.

This pattern is implemented by choosing colour combinations where readability and visual clarity is improved and ensuring that all information conveyed with colour is also available without colour for example from context or markup. Backgrounds should not detract from readability. The first reference is an example of bad use of colour.

Size

Motivation:

Design for the WWW is a balancing act between the graphics "wow" and the real time "now". The more graphically intense a site the longer it can take to download.

Problem:

How can the user access to information in a suitable way?

Forces:

- With a 28.8k connection, your computer can receive, on average, 2K per second. No one wants to wait 30 seconds just to see your site logo
- Developing fixed-size Web pages is a fundamentally flawed practice

Solution:

Provide information by using suitable sizes in images, fonts, and pages. Animations, images, long files should be provided if the user really wants it (on-demand). Page length, scrolling vs. paging needs, font size are important aspects.

Consequences:

Provide improvements on control, consistency and visual clarity

Examples & Implementation details:

http://www.number-10.gov.uk, http://www.tagheuer.com This pattern is implemented when thumbnails are used, when width and height attributes of the <IMG> tag are used, using standard fonts, or when large volume of information is broken into chunks and written in several pages avoiding scrolling. Organize documents so they may be read without style sheets is useful.

Secret

Motivation:

If user provides private information, he/she will need to have the right to expect confidentiality. Rapid advances in communication technology have accentuated the need for security in the Internet.

Problem:

How can the user be sure that information which he provides is protected?

Forces:

- Users want to security
- Users do not need to know technical aspects

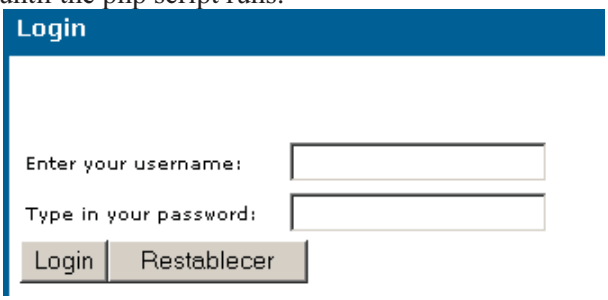
Solution:

Provide security needed mechanisms (access and privacy) and inform to the user of security conditions and terms of use. Users should be registered (**Subscription**) and so access to private sections on the Web site is allowed, but only a login and password is not sufficient sometimes [Yoder98].

Consequences:

Provide improvements on feedback and control

Examples & Implementation details: http://www.bankofamerica.com, http://www.cdnow.com This pattern is implemented with login form where we'll ask the user for their username and password by using php or asp. But unless your form is located on a secure server, the information is transmitted in clear text, and encryption won't occur until the php script runs.



Include links where users can read web site Privacy Policy Statement | Terms of Use

Novelty

Motivation:

Users want to know if there are new features in the Web site. Users admit suggestions and want to know offers and promotions.

Problem:

How can the user know novelties and latest news or suggestions of a web site?

Forces:

- User doesn't read web site. He/she has a look at it
- Users are in a hurry

Solution:

Provide novelties of the web site in a clear and intuitive manner where users will have rapid access to new services offered by web site

Consequences:

Provide improvements on functionality and navigation

Examples & Implementation details: http://www.microsoft.com, http://www.terra.es This pattern is implemented by placing latest news or suggestions in an outstanding place in the **Homepage**.

Location

Motivation:
When a user arrives at a Web site, like he/she arrives at a city, town or any important building needs to know where he/she is.

Problem:
How does the user know where he/she is?

Forces:
- Users need know where they are
- A complex Web site can be very disorienting for users

Solution:
Provide feedback information about location of the user in the web site.

Consequences:
Provide improvements on navigation, consistency and feedback

Examples & Implementation details: http://java.sun.com, http://www.acrobat.com This pattern is implemented by placing references in all web pages of the web site: using titles and breadcrumbs. Usually complex Web site includes a sitemap.



About this

Motivation:
All business Web sites need to provide a clear way to find information about the company no matter how big or small the company is

Problem:
How does the user know which is the purpose of the site?

Forces:
- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know who is behind the service

Solution:
Provide a place or a link where the user can get information about the web site's content.

Consequences:
Provide improvements on functionality and feedback.

Examples & Implementation details: http://www.sunspot.net, http://www.ireland.com This pattern is implemented by adding a section where information about owner of the page can be found. Normally a link to this section is situated in the **Homepage**.

Summary

The following tables summarise the patterns in this pattern catalogue for reference purposes. These patterns could be integrated on a methodology like a checklist to develop user interfaces like IDEAS [Lozano 01]. There are patterns of requirements, like these, that can be used in beginning of an usability-based iterative life cycle. So patterns can be used to improve a participatory design, evaluate web site under usability criteria and facilitate communication between stakeholders involved in Web site developing.

Problem	Solution	Pattern name
How does the user know where he is?	*Supply a reception place where conditions of user access to web site can be evaluated*	Welcome
How does the user know where he can go and what he will find there?	*Provide meaningful links to the different pages of the web sites*	Indication
How can the user visit the web site at his own pace?	*Provide information of a suitable way by taking into account users*	Polyglot
How does the user know that he is in the same web site?	*Provide an uniform aspect of the web site (colours, sizes, distribution, etc.)*	Similarity
How does the user know that he can visit the web site without problems?	*Provide tools or information needed to visit the web site of sutiable manner*	Ready

Problem	Solution	Pattern name
How does the user know where he is?	*Provide a reference point of the web site*	Homepage
How can the user access the content of the Web page in a simple and proper way?	*Provide information by using a simple language and jargon is avoided*	Polite
How does the user know when his operations have finished or the finished state of them?	*Provide feedback of the user action (sendings, loadings, downloadings, etc.)*	Busy
How can the user be sure of his actions?	*Provide elements for undo/redo, backing and clearing*	Second chance
How can the user provide information to the web site owner?	*Provide appropriate "blanks" to be filled in, which clearly and correctly indicate what information should be provided*	Form
How can the user visit a web site without getting confused, being interrupted or being disoriented?	*Be careful with using disoriented components (frames, animated gifs, floating, windows, banners, applets, flash, etc.)*	Danger!

Problem	Solution	Pattern name
How can the user know the purpose of the web site is?	*Provide a slogan or image that identifies the web site and its purpose*	Tag Line
How can the user get a suitable print of information?	*Provide information on several ways and formats and give the possibility of printing or downloading wide documents*	Print
How can the user be infomred with meaningful information for him?	*Provide a Form where the user can get information that he wants automatically*	Subscription
How can the user request for additional information about the content of the web site?	*Provide a Form, a place or a link in the web site where the user can get additional information about the web site owner and his products*	Contact us
How can the user know if a web site can provide the information he wants?	*Provide a search engine or overview page*	Search
How does the user know where he/she has been?	*Keep information about user actions, visited places, logins, etc.*	Recognize
How can the user access to information web site in a suitable way?	*Provide information by using suitable colours in fonts, backgrounds and image.*	Colour
How can the user access to information web site in a suitable way?	*Provide information by using suitable sizes in images, fonts, and pages*	Size
How can the user be sure that information that he provides is protected?	*Provide security mechanisms (access and privacy) needed to protect user data and the web site and inform to the user of security conditions*	Secret
How can the user know novelties and latest news of a web site?	*Provide suggestions and news of the web site in a clear and intuitive manner*	Novelty
How can the user know where he is, or what is the section that he is visiting?	*Provide feedback information about location of the user in the web site*	Location
How can the user get additional information about web site owner?	*Include a link to an "About Us" section*	About this

Acknowledgements

We would like to thank our shepherd Serge Demeyer for his valuable suggestions and comments on this paper. This paper is his paper too.

This work is supported in part by the Spanish CICYT TIC 2000-1673-C06-06 and CICYT TIC 2000-1106-c02-02 grants.

References

[Alexander 77] Christopher Alexander. “A Pattern Language”, Oxford University Press, 1977.

[Alexander 79] Christopher Alexander. “The Timeless Way of Building”, Oxford University Press, 1979.

[Constantine 01] Constantine Stephanidis, Anthony Savidis. “Universal Access in the Information Society: Methods, Tools and Interaction Technologies.” Springer-Verlang. 2001.

[Erickson 97] Thomas Erickson, “Supporting Interdisciplinary Design: Towards Pattern Languages for Workplaces”. 1997. http://www.pliant.org/personal/Tom_Erickson/Patterns.Chapter.html

[Evans 98] Mary Evans. “Web Design: An Empiricist’s Guide”. University of Whasington, Seatle, Washington. 1998.

[Lozano 01] María Lozano, Isidro Ramos, Pascual González. “User interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development”. IEEE 34th International Conference on TOOLS USA. 2001.

[Lyardet 00] Fernando Lyardet, Gustavo Rossi. Web Usability Patterns.2000.

[Meszaros & Doble 94] Gerard Meszaros, Jim Doble, “A Pattern Language for Pattern Writing”, in Martin, Riehle, Buschmann, Pattern Languages of Program Design 3. Reading, Mass: Addison-Wesley, 1994.

[Nielsen 00] Jakob Nielsen, “Designing Web Usability: The Practice of Simplicity”. New Riders Publishing. 2000.

[Nielsen 02] Jakob Nielsen, Marie Tahir. “Homepage Usability: 50 Websites deconstructed ”. New Riders. 2002.

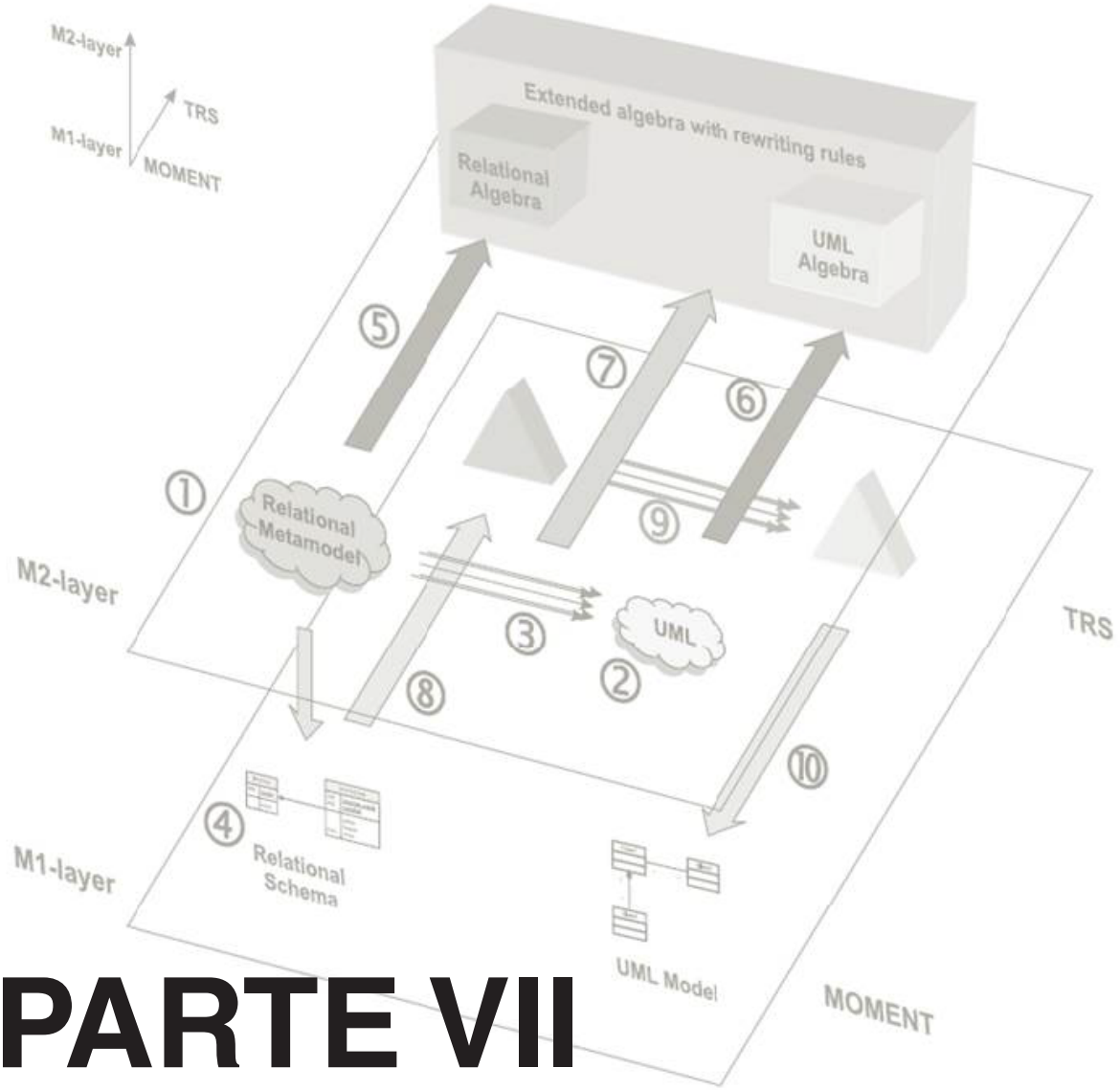
[Shneiderman 98] Ben Shneiderman. “Designing the User Interface: Strategies for Effective Human-Computer Interaction”. Addison Wesley. 1998.

[Tidwell 98] Jenifer Tidwell. “Common Ground: A Pattern Language for Human-Computer Interaction”. http://www.mit.edu/~jtidwell/. 1998/99

[Welie 01] Martijn van Welie. Interaction design patterns. http://www.welie.com/. 2001.

[Yoder 98] Joseph Yoder, Jeffrey Barcalow. “Arquitectural Patterns for Enabling Application Security”. PloP’97 D-4 book. 1998.

[Guidelines] Yale C/AIM WWW Style Manual
http://info.med.yale.edu
Apple’s Web Design Guide
http://applenet.apple.com
IBM Web Design Guidelines
http://www.ibm.com/IBM/HCI/guidelines



PARTE VII

GESTIÓN DE MODELOS

PRÓLOGO DE JOSÉ ÁNGEL CARSÍ

GESTIÓN DE MODELOS (2001-2006)

PARTE VII

Creo que fue en el año 1992 cuando oí hablar por primera vez del profesor Isidro Ramos. ¡El director del DSIC nos iba a dar clase de Ingeniería de la Programación en 5º curso de carrera! Nos contó una formalización algebraica del modelo orientado a objetos, que me costó unos 3 años de entender. Claro, suspendí esa asignatura en Junio aunque en Septiembre, y por los pelos, la aprobé.

Cosa curiosa: aunque el tema era exigente, fue en su grupo donde fui a terminar haciendo mi PFC sobre el núcleo de un motor de un sistema gestor de bases de datos orientados a objetos para una cosa que se llamaba OASIS. Fui dirigido por el profesor José Hilario Canós, en lo que luego me enteré era una pequeña parte de su tesis. Entre los recuerdos de entonces están las reuniones semanales de grupo en las que, sobre todo el profesor Isidro Ramos, nos contaban artículos que habían sido publicados fuera. Recuerdo tomar muchas notas y tener que repasarlas varias veces y preguntar las dudas que tenía para entrever lo que se decía en esas reuniones.

Aunque acabé trabajando fuera de la universidad, un día me llamaron por teléfono porque había una beca para un proyecto llamado «OOAP-Object Oriented Assitant Prototyper» en el que una parte era algo parecido a lo que había hecho en mi PFC. Y ahí fui.

Cuando llegó el momento de empezar a realizar la tesis, tuve la suerte de tener como co-directores a los profesores Isidro Ramos y José Hilario Canós. Fueron muchas las reuniones, paciencia, discusiones y presentaciones con las que finalmente consiguieron que la acabara en el 1999. Mi tesis consistió en dar soporte a la evolución para aplicaciones especificadas haciendo uso de OASIS a través de la introducción del concepto de metaobjeto en OASIS.

Por entonces, teníamos una buena relación con CARE, una empresa que estaba construyendo una herramienta CASE industrial para especificar requisitos de aplicaciones en OASIS

y generar las mismas en Java, C++, o Visual Basic, con soporte para motores de bases de datos como SQL Server, Oracle y otros. Ellos tenían el problema de la evolución de las aplicaciones, y nosotros sabíamos cómo hacerlo. El proyecto ADAM surgió de forma natural.

Los primeros artículos de esta parte son trabajos relacionados con lo que se hizo en ADAM: el generador automático de planes de migración, el comparador automático de esquemas conceptuales orientados a objetos, los patrones de generación de código para construir el plan de migración, y otros. Para poder realizar estos trabajos se contrató a 3 personas: Josep Silva, Jennifer Pérez y Víctor Anaya. Josep poco tiempo después de acabar su PFC aceptó una beca en otro grupo del departamento y en la actualidad es profesor del mismo. Víctor salió al mercado laboral y pasó de una empresa a otra. Jenny continuó mucho tiempo más en el grupo acabando su tesis doctoral sobre modelos arquitectónicos orientados a aspectos utilizando técnicas de MDA y hasta la fecha está en la Universidad Politécnica de Madrid.

Como extensión natural del trabajo relacionado en ADAM surgió RELS, también financiado por CARE. En RELS se quería resolver el problema de qué hacer con las aplicaciones legadas y cómo construir aplicaciones modernas extrayendo la mayor información posible de ellas, llegando a construir un modelo inicial de la aplicación y migrando los datos. Al igual que en el proyecto anterior, se contrató a 3 persona para que nos ayudaran con las implementaciones: José María Cubel, Fernando Domínguez y Artur Boronat. José María y Fernando fueron contratados por diferentes empresas y hace tiempo que les perdí la pista.

Muchos de los trabajos que se habían realizado en el grupo se basaban en elevar el nivel de abstracción, trabajar con modelos en vez de con código, utilizar técnicas de compilación o de traducción, y si podía ser automática, mejor. Durante mucho tiempo se estuvo trabajando en algo que no tenía nombre todavía. Hasta que en el 2001 Philip Bernstein lo acuñó. ¡Estábamos trabajando en Gestión de Modelos!

Artur continuó mucho tiempo con nosotros. El tema que le atraía era justamente el que aborda esta sección de artículos: la gestión de modelos. Algunos de los trabajos iniciales intentan definir una plataforma genérica para la gestión de modelos, que llamamos MOMENT, basando la semántica en RDF. Trabajos posteriores abordan la semántica mediante sistemas de reescritura de términos mediante CafeOBJ aunque al final se optó por MAUDE por ser un sistema mucho más completo y robusto que permitió la formalización algebraica del núcleo de MOMENT. Otros artículos muestran metáforas gráficas para la definición de modelos, soporte para la trazabilidad automática, motores de transformaciones, fusión de modelos, definición precisa de la semántica de OCL, y la definición de operadores complejos de modelos haciendo uso de un DSL, entre otros. Artur acabó su tesis sobre gestión de modelos y en la actualidad es profesor en la universidad de Leicester.

En esta etapa se consiguieron cosas realmente interesantes, como por ejemplo la primera implementación funcional del estándar QVT, la integración de MAUDE en el entorno de trabajo de ECLIPSE y disponer de un entorno completo que permitía trabajar con modelos de manera independiente del metamodelo (MOMENT) utilizando como interfaz de usuario el mismo ECLIPSE.

En este periodo también nos presentamos a un concurso público organizado por la conselleria de obras públicas de la Generalitat Valenciana en el que para optar se debía presentar una herramienta CASE completa que permitiera definir aplicaciones con UML 2.0 y generara el código y los scripts de bases de datos. Lo hicimos en menos de un mes con la ayuda de MOMENT. Al concurso nos presentamos de la mano de CapGemini y lo perdimos por 1 punto en

aspectos que tenían que ver con la atención al público. Concretamente, con la cantidad de horas de atención al público que la empresa iba a dedicar a los usuarios de la herramienta. ¡Por los pelos!

Como parte de la validación de MOMENT, y gracias a una visita que nos realizó el profesor Hans-Dieter Ehrich —que nos explicó el problema y nos puso en contacto con gente de la universidad de Braunschweig—, se construyó una herramienta que extrae información de bases de datos como TRANSPATH acerca de los caminos de señales que recorren las moléculas en su paso por las células para llegar al núcleo y construye simuladores en redes de Petri coloreadas que los biólogos pueden usar para comprender su funcionamiento.

Diferentes personas nos ayudaron con las implementaciones necesarias que terminaron en forma de PFC. Julián Pedrós, Luis Hoyos, Pascual Queralt, José Iborra y Abel Gómez compartieron con nosotros muchas reuniones y discusiones, y todas llegaron a buen puerto. Julián, Luis, Pascual y José hace tiempo que salieron al mercado laboral y les perdí la pista. Abel es otro de los que conseguimos 'engañar' para que se quedara con nosotros y realizara una tesis en líneas de producto software dirigidas por modelos.

El último y más reciente trabajo de esta parte aborda la especificación algebraica del lenguaje natural para el procesamiento del mismo. Un trabajo realizado con investigadores de la universidad de Alicante. Esta sección aborda mucho trabajo, tiempo y dedicación… y todo ello habría sido imposible sin la persona que nos enseñó, empeñó su tiempo, esfuerzo y paciencia con todos nosotros.

Muchas gracias Isidro por estar con nosotros.

Pepe Carsí

Un Generador Automático de Planes de Migración de Datos*

J. A. Carsí, I. Ramos, J. Silva, J. Pérez, V. Anaya
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46071 Valencia - España
{pcarsi, iramos, jsilva, jeperez, vanaya}@dsic.upv.es

Abstract.– This paper presents a solution for the data migration problem in information systems that must evolve to adapt to new requirements. The solution uses the information provided by the conceptual schemas, which represent the evolution that the information system has undergone, in order to automatically generate data migration plans. The plan offers the maximal flexibility permitting the user to act always with the maximum possible freedom in its construction and execution. Thus, a solution to a seldom addressed problem in Software Engineering is offered, and a tool that supports it is presented. This tool has been developed by the Information Systems and Computing Department of the Universidad Politécnica de Valencia in collaboration with the software development firm CONSOFT.

Keywords: data migration plan, migration language, patterns, migration expressions, automatic generation, comparison criteria, comparison algorithm.

1. INTRODUCCIÓN

La primera ley de la ingeniería de sistemas de Bersoft [4] dice *"Sin importar en qué momento del ciclo de vida del sistema nos encontremos, el sistema cambiará y el deseo de cambiarlo persistirá a lo largo de todo el ciclo de vida."*. Es un hecho destacado por numerosos investigadores y profesionales que los sistemas informáticos deben evolucionar para adecuarse a los siempre cambiantes requisitos del entorno. Las estadísticas indican que entre el 65% y el 75% de las fuerzas vivas relacionadas con el mundo de la informática [7] y aproximadamente el 80% de los gastos totales del software [23] se dedican al mantenimiento del software existente.

Idealmente, a la hora de introducir una modificación en una aplicación informática se debe seguir un proceso (por ejemplo el ISO/IEEE 12207 [5]) en el cual sea identificada la razón del cambio, se analice cómo afecta el cambio a introducir en la aplicación[1], para posteriormente realizar los cambios allí donde sean necesarios, finalizando con una exhaustiva fase de pruebas que demuestre que la modificación no ha introducido errores. Todo el proceso anterior debe estar bajo un estricto control de cambios que recoja todos los documentos que se generan y almacene las diferentes versiones.

Artículo recibido el 25 de febrero de 2002.

* Este trabajo ha sido financiado con fondos del Proyecto CICYT del programa FEDER con ref. TIC 1FD97-1102.

[1] Tanto en los documentos de análisis, diseño, como en el código.

ISSN 1665-238X.

Lamentablemente, muchas empresas simplifican el proceso anterior reduciéndolo, en la mayoría de los casos, a un análisis de la modificación para posteriormente aplicarla directamente al código sin excesivas pruebas y menor documentación.

En la actualidad, numerosas herramientas CASE son capaces de generar todo o parte del código que implementa una aplicación a partir de la información que se introduce en los modelos del método que se esté utilizando. Así por ejemplo, herramientas como Rational Rose [13], Together [20] o System Architect [19], son capaces de generar esqueletos de programas en lenguajes de programación como Java, C++, VBasic y *scripts* en SQL para generar las bases de datos necesarias a partir de la información estructural que se introduce mediante dichas herramientas en los diagramas de clases. Otras herramientas CASE como OO-Method/CASE [10] u Oblog/CASE [14], son capaces de generar aplicaciones completas a partir de la información de modelado que se introduce en sus modelos gracias a su sólido soporte formal. En el caso de OO-Method es el lenguaje de especificación formal OASIS, actualmente por su versión 3.0 [6].

Todas las herramientas CASE anteriores no ofrecen un buen soporte al paso del tiempo. La introducción de un nuevo requisito en las aplicaciones en funcionamiento, siguiendo algún método de gestión de cambios, consiste en introducirlo en los modelos pertinentes para posteriormente y aprovechando las capacidades generadoras de las herramientas CASE, regenerar la aplicación y el esquema de la base de datos acorde con el modelo del sistema actualizado. Finalmente, existen dos esquemas conceptuales del sistema de información, uno inicial con la definición del sistema antes de introducir los nuevos requisitos y uno final en el que ya se han introducido. Además, existen dos bases de datos, una inicial con toda la información que se ha generado mientras la aplicación ha estado en funcionamiento, y una final que satisface los nuevos requisitos del sistema, sin información. El problema es ¿cómo trasvasar la información de la base de datos inicial a la final?

La solución ofrecida en la siguiente, a partir de la información disponible en los modelos generados con las herramientas CASE, por comparación de los esquemas que definen el sistema original y el sistema evolucionado que recoge los nuevos cambios, es posible generar de manera automática planes de migración de datos cuya ejecución migre y transforme la información almacenada en la base de datos inicial a la final.

De esta forma, se da solución al problema de la evolución de los sistemas de información que han de cambiar para adaptarse a los nuevos requisitos de una manera sencilla y elegante. Dicha solución genera de manera automática los planes de migración de los datos y toda la documentación necesaria para documentar el cambio, agilizando en gran medida el tiempo necesario para adaptarse a los cambios.

El artículo está dividido en cuatro secciones. Tras la introducción, en el segundo apartado se presentan las diferentes fases en las que se ha dividido la herramienta de generación de planes de migración de datos. En el tercer apartado se presenta algunos trabajos relacionados con el presente. Y finalmente, en el cuarto apartado se presentan las conclusiones y los trabajos futuros.

2. Fases del Generador Automático de Planes de Migración de Datos

La Figura 1 muestra las fases en las que se ha divido el generador de planes de migración:

1. Un comparador de esquemas *conceptuales aprovecha la información generada por la herramienta CASE para comparar la estructura de los modelos y calcular las diferencias que hay entre ambos.*
2. Un generador de planes *utiliza las semejanzas y diferencias detectadas por la primera fase para construir de manera automática por aplicación de diversos patrones un plan de migración en un lenguaje OO independiente del sistema gestor de bases de datos (SGBD) que se esté utilizando.*
3. *Finalmente, el plan de migración que se ha construido en la fase anterior debe ser ejecutado para que se realice la migración de los datos. Un* traductor *realiza esta tarea compilando el plan a paquetes DTS*[2] *que pueden ser ejecutados en SQL Server.*

2.1. Comparador de Esquemas Conceptuales

La primera fase de la migración de datos [17] consiste en determinar los orígenes y los destinos de los datos para realizar la migración. Para ello, se realiza un proceso comparativo a través del cual se determina qué modificaciones han sufrido los elementos de los esquemas conceptuales; y de esta manera, conocer entre qué elementos debe realizarse el trasvase de información. La comparación de esquemas conceptuales tiene como entrada los dos esquemas entre los que se va a realizar la migración; y como salida, qué elementos de los esquemas han sido eliminados, qué elementos han sido añadidos, y finalmente qué elementos han sufrido modificaciones pero deriva uno del otro.

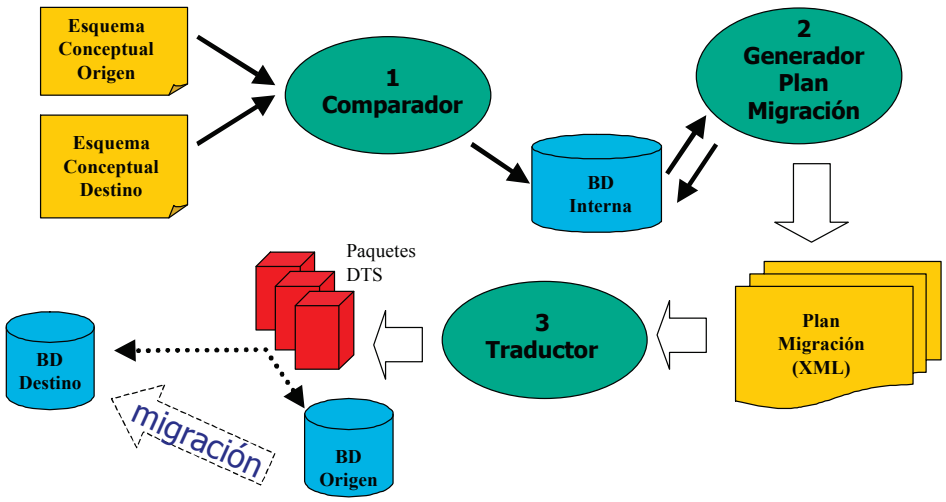


Figura 1. Fases del generador automático de planes de migración de datos.

El trabajo realizado para obtener un algoritmo capaz de automatizar la comparación de esquemas conceptuales, se ha dividido en dos secciones independientes:

1. Un estudio para determinar qué tipo de algoritmos son adecuados y presentan un acercamiento mayor al problema tratado. Finalmente, se ha diseñado un algoritmo específico de comparación de esquemas conceptuales.

[2] Data Transformation Services.

2. La determinación de qué información deben utilizar los algoritmos para comparar. Dicha información constituye lo que se ha denominado 'criterio de comparación', el cual es parcialmente independiente del algoritmo utilizado.

El análisis algorítmico realizado ha obtenido como resultado un algoritmo capaz de generar automáticamente las diferencias existentes entre dos esquemas conceptuales OASIS [18]. Dichas diferencias vienen dadas en términos de inserciones, modificaciones y borrados. El algoritmo está basado en técnicas de programación dinámica y de teoría de grafos, debido a que aprovecha la estructura de árbol de los esquemas conceptuales. Para procesar los esquemas conceptuales, se divide el espacio de comparación en cuatro clases distintas de elementos (clases, atributos, relaciones de agregación y relaciones de especialización) como se muestra en la Figura 2. Dicho fraccionamiento reduce el espacio de búsqueda limitando el número de comparaciones y consecuentemente, el tiempo de procesamiento del algoritmo.

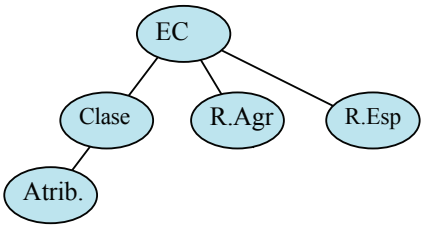


Figura 2. Estructura de árbol para la representación de esquemas conceptuales.

El algoritmo considera durante el proceso de comparación información semántica, siendo capaz de detectar modificaciones complejas como pueden ser *Inline* (unión de varias clases en una sola) o *Duplicate* (duplicado de la definición de una clase en varias) [16].

El algoritmo de comparación es capaz de utilizar información de distinta naturaleza para comparar los elementos de los esquemas conceptuales. Dicha información puede ser el nombre de los elementos, su identificador, fechas de creación, etc. Al aplicar el algoritmo de comparación utilizando diferentes criterios de comparación se pueden obtener resultados distintos dependiendo del contexto en el que sea utilizado.

Se llama *criterio de comparación* a la información utilizada para determinar si dos elementos de dos esquemas diferentes proviene uno del otro o no. El algoritmo propuesto es capaz de considerar en la actualidad cinco criterios de comparación distintos junto con las posibles combinaciones entre ellos. Para llegar a dicha selección de criterios de comparación se ha llevado a cabo un experimento utilizando siete criterios distintos para comparar una muestra de veinte esquemas conceptuales. A partir de dicho experimento se han obtenido estadísticas representativas del comportamiento de cada criterio ante una gran diversidad de situaciones.

El algoritmo propuesto junto con el diseño de los criterios de comparación que es capaz de utilizar son la primera fase de la herramienta que ha sido desarrollada para la generación de planes de migración. Se han elaborado los modelos conceptuales de la comparación de esquemas y a partir de los mismos se ha generado automáticamente el código de la aplicación utilizando la herramienta CASE OO-Method, la misma a la que da soporte de evolución. El resultado es una herramienta capaz de obtener de manera automática las diferencias existentes entre dos esquemas conceptuales cualesquiera.

Para la validación de la herramienta se han realizado diversas pruebas de comparación con esquemas conceptuales industriales, obteniéndose tasas medias de acierto cercanas al 98%. La

herramienta genera como salida el conjunto de correspondencias entre los elementos del esquema destino y origen así como aquellos para los que no se ha encontrado su correspondiente.

2.2. Generación de un Plan de Migración de Datos

A partir de las correspondencias generadas en la fase inicial del proceso de migración, de los esquemas conceptuales y del *orden de migración* [1] calculado a partir de las dependencias (relaciones) entre elementos del esquema conceptual destino, la segunda fase [11] genera de forma automática la primera versión de un *plan de migración de datos*.

Un plan de migración está formado por un conjunto de cambios expresados en un lenguaje de migración, cuya posterior ejecución permite migrar los datos de una BD origen a una BD destino de forma adecuada. La creación de un plan de migración completo debe realizarse incorporando cada uno de los cambios necesarios para que pueda hacerse dicha migración.

Cada uno de los posibles cambios que pueden realizarse en las bases de datos, se expresa mediante un lenguaje pivote declarativo. Este *lenguaje de migración* especifica los cambios basándose en el modelo orientado a objetos, debido a que la información que maneja son esquemas conceptuales orientados a objetos OASIS. De esta forma, se permite especificar el plan de migración de forma correcta y próxima al usuario de la aplicación, independientemente del SGBDR en el que se vaya a ejecutar finalmente.

El plan de migración se construye gracias a la composición de diferentes elementos: expresiones de migración, cambios y módulos de migración. El nivel de abstracción más elevado de esta composición de elementos lo constituyen los módulos de migración, mientras que el gránulo más pequeño son las expresiones de migración.

Una *expresión de migración* es el concepto general para hacer referencia a las diferentes expresiones que permite especificar el lenguaje de migración diseñado. Cada una de ellas ayuda a especificar unos determinados elementos de un esquema conceptual, tiene una connotación semántica diferente y unas restricciones sintácticas claramente definidas.

Un *cambio* contiene el conjunto de expresiones de migración necesarias para expresar las modificaciones de un determinado elemento del esquema conceptual origen y los filtros que se le han de aplicar a su población, para poderlo migrar correctamente.

Un *módulo de migración* esta asociado a una clase, una relación de agregación o una relación de especialización del esquema conceptual destino. Los módulos representan unidades de migración, cuya composición mediante agregaciones y especializaciones constituyen el plan de migración. Un ejemplo de dicha composición es el módulo de migración de una agregación, que está compuesto por los módulos de migración de clase de sus clases compuesta y componente.

El proceso de generación de un plan de migración de datos se ha dividido en dos partes, cada una de ellas genera y parte de una información distinta:

1. La primera parte consiste en generar automáticamente tanto la estructura del plan de migración de datos, como cada una de las expresiones que lo componen. Para ello, se define un módulo de migración vacío por cada elemento del esquema conceptual destino con el orden y la composición que indique el orden de migración. Después, se calculan automáticamente las expresiones de cada

uno de los cambios asociados a un módulo de migración mediante el uso de *patrones de expresiones de migración*. Posteriormente, cada una de las expresiones generadas se incluye en el módulo adecuado y en el lugar correcto basándose en los *patrones de migración*. El resultado de este subproceso automático es una primera versión de un plan de migración completo.

2. La segunda parte tiene como objetivo la validación del plan de migración que se ha generado automáticamente por la primera parte. Para ello, el analista necesita interaccionar con la herramienta de migración de datos, no solo para validar el plan de migración propuesto, sino también para modificarlo cuando no exprese lo que él desee.

Por este motivo, se ha diseñado una interfaz gráfica de usuario en la que se muestran las correspondencias entre los elementos del esquema conceptual final (EC') y los del esquema conceptual original (EC). Se han contemplado varias representaciones visuales para mostrar los elementos que forman parte de ambos esquemas y finalmente se ha optado por una visualización en forma arborescente. Por otro lado, mediante un código de colores y de símbolos se muestran las diferencias entre ambos esquemas, además de indicar los cambios, mediante expresiones asociadas a los elementos (ver Figura 3).

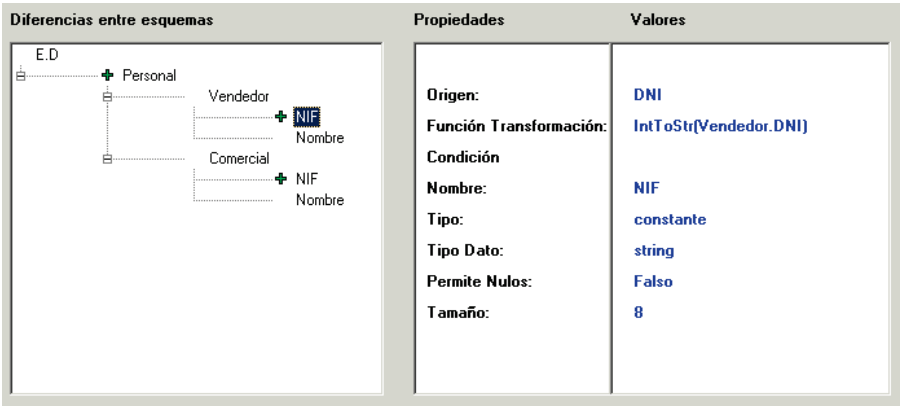


Figura 3. Visualización de diferencias entre dos esquemas conceptuales.

Los cambios se muestran desde la perspectiva del esquema conceptual final, ya que el objetivo es llenar de información la base de datos final a partir de la base de datos inicial. Esta visualización muestra en el primer nivel del árbol las clases y relaciones del esquema final en el orden en que se han citado. Si se despliegan cualquiera de estos elementos, se encuentra a un segundo nivel de anidamiento el elemento o elementos del esquema conceptual origen del que provienen sus datos, que serán de su mismo tipo o compatibles. Finalmente, en el tercer nivel de anidamiento del árbol se encuentran los atributos de la clase destino de la que cuelgan, la cual aparece en el primer nivel del árbol. Se ha de destacar que en este último nivel del árbol aparece en la parte derecha de la pantalla de exploración el atributo origen del cual se extraen los datos para el atributo destino seleccionado. Dicho atributo origen pertenece a la clase del esquema conceptual inicial del que cuelga el atributo destino. La representación arbórea de la figura 3 muestra que los objetos de la clase *Personal* provienen de los objetos de las clases *Comercial* y *Vendedor*. También da a conocer que los valores de los atributos *NIF* y *Nombre*, de la clase *Personal* provienen de los atributos *DNI* y *Nombre* de las clases *Comercial* y *Vendedor*, cuya especificación no aparece en la estructura arborescente, sino en las propiedades que aparecen en la ventana de exploración asociada al árbol de visualización.

Por otro lado, siempre que se siga la estructura del árbol a la hora de validar la información, la complejidad de ésta se reduce, ya que primero se validan las clases y luego los atributos. De esta manera, se tiene la certeza de que los atributos que se asocien a un atributo del esquema final, han de pertenecer a la clase del esquema inicial que se ha asociado a la clase del esquema final a la que pertenece dicho atributo.

Otro punto a considerar es la visualización de las operaciones realizadas en el esquema conceptual inicial que lo hicieron evolucionar al esquema conceptual final. Las posibles operaciones que pueden sufrir los elementos de un esquema conceptual son: inserción, modificación y borrado. No obstante, al visualizar las diferencias desde la perspectiva del esquema conceptual final, solamente se consideran la inserción y modificación de elementos. Los borrados no se visualizan en el árbol, por no existir en el esquema conceptual destino.

Finalmente, una vez completadas las dos partes de esta fase del proceso de migración se obtendrá un plan de migración de datos validado por el analista. Dicho plan será compilado por la tercera fase de la herramienta que hará efectiva la migración. Con el objetivo de que la fase de ejecución tenga el mayor conocimiento posible del proceso de migración, en lugar de proporcionarle el plan de migración de forma textual, se estructura en formato XML. De este modo, se facilita la lectura del plan de migración y el proceso de traducción.

2.3. Traductor del Plan de Migración

Una vez validado el plan de migración sintáctica y semánticamente, se procede a traducirlo a paquetes de código [2]. El objetivo de la ejecución de dichos paquetes es la migración de la información entre bases de datos relacionales. Dicha migración puede realizarse entre bases de datos heterogéneas (Oracle, Access, dBase, SQLServer, etc). Para realizar dicha tarea no basta con la simple generación de sentencias SQL, ya que la expresión de transformaciones complejas sobre los datos origen no es viable al no disponerse de un lenguaje imperativo.

La solución adoptada es la generación de código interpretado por herramientas específicas de migración. Entre las distintas posibilidades que existen, se ha optado por DTS (Data Transformation Services de Microsoft®), herramienta que acompaña a SQL Server. Entre las ventajas que proporciona generar código para DTS está la capacidad de migrar información entre multitud de fuentes de datos (ficheros, bases de datos jerárquicas, bases de datos relaciones heterogéneas, etc), la posibilidad de permitir expresar transformaciones complejas sobre los datos haciendo uso de lenguajes *script*, y las opciones de copia rápida (Bulk Copy) y de carga rápida (Fast Load Option), que optimizan el rendimiento de la migración. Pero a pesar de todas las facilidades ofrecidas, un proceso de migración necesita de la implementación de miles de líneas de código. El traductor implementado se encarga de ahorrar ese trabajo.

La entrada principal del traductor la constituye el plan de migración. Junto a éste, para traducirlo a paquetes de código se necesita de información adicional que no está disponible en él por cuestiones de rendimiento y evitar duplicidad. En el plan de migración pueden aparecer expresiones de navegación y atributos derivados origen, ya sea en los filtros poblacionales, en las condiciones o en las funciones de transformación, que hacen necesaria la consulta de los esquemas conceptuales para obtener dicha información.

La salida del traductor es un conjunto de paquetes DTS. Un paquete DTS [15, 22] contiene un conjunto de conexiones asociadas a fuentes de datos de dónde leer y salvar la información. A su vez el

paquete posee un conjunto de tareas que pueden ser de distinto tipo. Las únicas que se utilizan en este trabajo son la de bombeo de datos y la de consulta guiada por datos. La primera para la migración de clases y especializaciones, ya que únicamente se realizan inserciones de tuplas. La segunda para las agregaciones y asociaciones, ya que realizan modificaciones de tuplas existentes. Una tarea de bombeo de datos tiene asociada dos conexiones, una para leer datos y otra para depositarlos. La lectura de información se puede realizar mediante una sentencia de consulta SQL. Dicha información es asignada y transformada en el cuerpo en *script* de una función de transformación que toda tarea de bombeo de datos posee. La forma de asignar valores en el cuerpo *script* es:

$$D_i = f\_trans(O_1, ..., O_n)$$

donde D_i es el atributo i-esimo de la tabla destino $\forall i$: $1 <= i <= m$, siendo m el número de campos de la tabla destino. *f_trans* es la función de transformación que se aplica sobre los operandos $O_1, ..., O_n$, donde O_k es un campo origen o una expresión introducida por el usuario.

El traductor lee el plan de migración y los esquemas conceptuales y extrae la información necesaria para generar paquetes automáticamente. La información extraída es almacenada en estructuras intermedias en memoria principal. Estas estructuras guardan, de una forma adecuada la información, de los esquemas conceptuales y del plan de migración, para la generación de paquetes. Las estructuras con la información del plan de migración mantienen referencias a aquellas que poseen la información de los esquemas para facilitar el acceso y almacenamiento de la información requerida. Las estructuras poseen procedimientos adecuados para deducir información con la finalidad de generar los paquetes. Estos procedimientos son llamados por el traductor para generar el conjunto de paquetes DTS de salida. En ello se resuelve la diferencia semántica entre el contexto OO del plan de migración, y el contexto relacional de los paquetes de código.

En el proceso de traducción se establecen correspondencias semánticas entre las secciones del plan de migración orientado a objetos generado y evaluado por el analista en la segunda fase, y los paquetes DTS. Dichas correspondencias son necesarias para definir patrones de generación de código. La forma de cada patrón depende de la representación que en base de datos se hace de los elementos del esquema conceptual.

La siguiente tabla muestra las correspondencias entre las secciones de un plan de migración y un paquete DTS:

PLAN DE MIGRACIÓN	CÓDIGO DTS
Módulo de migración	Paquete
Submódulo de migración	Tarea
Filtro poblacional	Condición WHERE de la consulta de la tarea
Funciones de transformación	Función en lenguaje script de la sección de transformación de una tarea.
Condiciones sobre atributos	Sentencias condicionales indicadas con un lenguaje script en la sección de transformación de la tarea

A partir del conjunto de las correspondencias y dependiendo del tipo de módulo de migración (depende del elemento del esquema conceptual destino al que está asociado) se expresa un patrón que indica la forma de generar código de forma automática a partir de cada módulo de entrada.

Existen dos modos de funcionamiento del traductor:

- *Modo intérprete: el traductor genera un conjunto de paquetes, tantos como clases, relaciones de agregación y asociación haya en el esquema conceptual evolucionado. La ejecución de paquetes del mismo tipo es independiente, pero se han de ejecutar los paquetes que migran la información de clases antes que los que provocan la migración de agregaciones y asociaciones. La migración de las especializaciones se hace junto a la de clases, de forma intrínseca. El usuario puede observar el correcto resultado de la migración parcial consultando los estados intermedios alcanzados para evaluar si es o no correcto el proceso.*

- *Modo compilado: el traductor genera un único paquete capaz de migrar toda la base de datos origen de una vez. El orden de migración lo impone el propio traductor mediante relaciones de precedencia entre tareas del paquete.*

3. TRABAJOS RELACIONADOS

En la actualidad, son los SGBD los que dan soporte a la evolución de los datos, ya sean relacionales como ORACLE [9] o SQL-Server [8] u orientados a objetos como O2 [3], Poet [12] o Versant [21] ya que éstos son las que almacenan la información que se genera con las aplicaciones. Muchos SGBD permiten la migración de datos mediante sus herramientas ETL (Extract, Transform & Load), bien ejecutando de forma individualizada sentencias SQL o bien codificando *scripts* que posteriormente serán ejecutados sobre las bases de datos. Estas herramientas no ofrecen un soporte automático a la generación de *scripts* para la migración de los datos, tal y como propone nuestra solución. Las herramientas ETL sólo proporcionan un entorno de interacción amigable para la migración de datos entre las bases de datos origen y destino. Por este motivo, no consiguen paliar el gran número de personas implicadas en el proceso de migración de datos y el alto coste temporal que desencadena. En todas estas soluciones el usuario ha de concebir la migración a un nivel de abstracción bajo, donde es difícil expresar transformaciones complejas. A su vez, se carece de integración entre las distintas fases del proceso de evolución, por lo que se requiere de personal especializado en las distintas fases de desarrollo de software. Las soluciones existentes en el mercado obligan a implementar miles de líneas de código en un lenguaje específico, con el fin de dar solución a un problema de migración concreto, requiriendo gran cantidad de tiempo y recursos en las etapas de codificación, consiguiendo soluciones de baja calidad.

Un enfoque más próximo a nuestra solución, debido a que es un proceso automático y se basa en la evolución del esquema, es la herramienta TESS que presenta Barbara Staund Lerner en su artículo [16]. TESS analiza el código asociado a los esquemas relacionales tras su previa traducción a un lenguaje independiente. Sin embargo, nuestra aproximación se basa la comparación de los esquemas conceptuales OO independientes del SGBD en el que se almacena la información, ahorrando el proceso de traducción para conseguir mayor abstracción.

Respecto a la visualización de las diferencias entre esquemas conceptuales, existen herramientas como *Visual Differencing* proporcionada por Rational Rose [13] que permite visualizarlas. La visualización es en forma árbol mostrando la unión de los esquemas que se están comparando. Las diferencias encontradas se muestran desde el punto de vista del esquema origen y el criterio de comparación es único, no se puede aplicar a subconjuntos del esquema y no es modificable de forma manual. Esta herramienta realiza la comparación entre esquemas conceptuales, pero su resultado no es utilizado posteriormente, y por lo tanto, no persigue el mismo objetivo que el presente trabajo, utilizar

las diferencias entre esquemas para obtener de forma automática las expresiones que constituirán el plan de migración de datos.

4. CONCLUSIONES

Se ha presentado una solución al problema de migración de los datos en las aplicaciones para adecuarse a los nuevos requisitos que surgen con el paso del tiempo. Además, se ha mostrado la arquitectura de una herramienta que, basándose en una herramienta CASE existente, aprovecha la información generada por ésta para dirigir el proceso de migración de los datos de forma automatizada.

De forma resumida, las fases en las que se ha dividido la herramienta para migrar la información consiste en: primero, comparar la información estructural de los esquemas inicial y evolucionado que definen el sistema de información; segundo, generar por aplicación de unos patrones una primera versión del plan de migración que debe de ser validada por el analista de la aplicación; para finalmente, tras un proceso de traducción a paquetes DTS, aplicar el plan generado a las bases de datos entre las que se desea migrar la información.

El proceso de modificación de las aplicaciones se simplifica gracias a la utilización de la herramienta descrita en este artículo. El proceso consiste en analizar el cambio, introducir la información de los nuevos requisitos en los modelos adecuados, regenerar el código de la aplicación y la base de datos mediante la herramienta CASE y, finalmente, validar el plan de migración de datos que es propuesto por la herramienta de migración.

Pruebas experimentales realizadas han demostrado que la utilización de esta herramienta ha conseguido reducir el número de personas involucradas y el tiempo dedicado en la modificación de las aplicaciones alrededor de un 80%.

REFERENCIAS

[1] S. Abad, J.A. Carsí, I. Ramos, "Obtención de un Orden de Migración para los Elementos de un Esquema Conceptual Orientado a Objetos", *Taller de Evolución, VI Jornadas de Ingeniería del Software y Bases de Datos*, Almagro (Ciudad Real), Nov. 2001.

[2] V. Anaya, *Generación de Módulos de Trasformación para la Migración de Información entre Bases de Datos a Partir de Migración*, Proyecto Fin de Carrera, Facultad de Informática, Universidad Politécnica de Valencia, Sep. 2001.

[3] Ardent Software, "O2", http://www.ardent.com/.

[4] E.H. Bersoff, V.D. Henderson, S.G. Siegel, *Software Configuration Management*, Prentice-Hall, 1980.

[5] IEEE, *IEEE Std 12207. Standard for Software Maintenance*, IEEE Computer Society Press. 1992.

[6] P. Letelier, P. Sanchez, I. Ramos, O. Pastor, *OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*, Servicio de Publicaciones, Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721-663-0, 1998.

[7] J.R. McKee, *"Maintenance as a Function of Design"*, *Proc. AFIPS National Computer Conf.*, Las Vegas, pp. 187-93.

[8] Microsoft® Corporation, "SQL Server", http://www.microsoft.com/sql.

[9] Oracle® Corporation, "Oracle", http://www.oracle.com.

[10] O. Pastor, E. Insfrán, V. Pelechano, J. Romero, J. Merseguer, "OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods", *Conference on Advanced Information Systems Engineering (CAiSE '97)*. LNCS 1250, Springer-Verlag 1997, ISBN: 3-540-63107-0, Barcelona, pp. 145-159, Jun. 1997.
[11] J. Pérez, *Generación de un Plan de Migración de Datos para Esquemas Conceptuales OASIS Generados con OO-Method/CASE*, Proyecto Fin de Carrera, Facultad de Informática, Universidad Politécnica de Valencia, Sep. 2001
[12] POET Software, "POET", http://www.poet.com.
[13] Rational Software, "Rational Rose", http://www.rational.com/products/rose/.
[14] A. Sernadas, J.F. Costa, C. Sernadas, "Object Specifications Through Diagrams: OBLOG Approach", INESC Lisboa, 1994.
[15] D. Siebold, Visual Basic Developer's Guide to SQL Server, Sybex, 2000.
[16] B. Staund Lernerm, "A Model for Compound Type Changes Encountered in Schema Evolution", *ACM Transactions on Database Systems (TODS),* Mar. 2000, Vol. 25 No. 1.
[17] J. Silva, *Comparación de Esquemas Conceptuales OASIS para la Obtención de las Diferencias Producidas por un Proceso de Evolución*, Proyecto Fin de Carrera, Facultad de Informática, Universidad Politécnica de Valencia, Sep. 2001.
[18] J. Silva, I. Ramos, J.A. Carsí, "Comparación Automática de Esquemas Conceptuales Orientados a Objeto", *Ingeniería Informática, Revista electrónica del DIICC, Edición número 7*; ISSN : 0717 – 4195, Ene. 2002.
[19] System Architect, http://www.popkin.com/products/sa2001/systemarchitect.htm.
[20] TogetherSoft Corporation, "Together", http://www.togethersoft.com/.
[21] Versant Object Technology, "Versant", http://www.versant.com/.
[22] Wrox®, "Recursos del Programador de la Editorial Wrox", http://p2p.wrox.com/archive/sql_server/2001-03/.
[23] E. Yourdon, "RE-3: Re-engineering, Restructuring and Reverse Engineering", *American Programmer*, 2 (4), pp. 3-10, 1989.

Comparación automática de esquemas conceptuales orientados a objeto.

J.Silva, J.A.Carsí, I.Ramos
{ jsilva | pcarsi | iramos }@dsic.upv.es

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46071 Valencia – España

Resumen

En este trabajo se realiza un estudio algorítmico de cómo comparar esquemas conceptuales orientados a objeto en un contexto de migración de datos. Se presenta un algoritmo de comparación de esquemas basado en la comparación de árboles que aprovecha información semántica para optimizar la eficiencia de la comparación. Se calcula a partir de dos esquemas cualesquiera, y de manera automática, su plantilla de equivalencias y diferencias en términos de inserciones, modificaciones y borrados. El presente trabajo forma parte del desarrollo de una herramienta para la migración automática de bases de datos, que está siendo desarrollada en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia.

Palabras clave: Comparación de esquemas, Algoritmos de comparación, Anclajes, Criterios de comparación, Recorrido de árboles, Segmentación de esquemas.

1. Introducción

Muchos sistemas y procesos pueden ser representados como estructuras de árbol, y en ocasiones se da la necesidad de realizar una comparación entre dichas estructuras con el fin de determinar la posible interoperación, las diferencias existentes, o simplemente el grado de similitud entre dichos sistemas. Un caso particular se da en la ingeniería del software cuando lo que queremos comparar son esquemas conceptuales de sistemas de información orientados a objeto.

La comparación de esquemas conceptuales tiene un especial interés en la migración automática de bases de datos. Idealmente, la evolución del software se realiza a nivel de esquema conceptual, y a partir del mismo se generan automáticamente las aplicaciones y bases de datos del sistema de información. En ocasiones ocurre que las bases de datos del sistema original contienen información, y dicha información debe ser traspasada al nuevo sistema, este hecho es conocido como migración de datos. El principal inconveniente en la migración de datos es que las bases de datos entre las que se quiere realizar la migración suelen ser muy distintas, y en ocasiones incompatibles.

En una migración de datos, es fundamental conocer exactamente el grado de evolución sufrido por el esquema y la base de datos asociada que va a ser migrada. Se hace indispensable conocer qué modificaciones han sufrido los elementos del esquema origen, así como qué inserciones y borrados se han producido durante el proceso de evolución. También es una tarea importante determinar qué información se debe usar para identificar el origen y el destino de la migración. Por ejemplo, si en la BD del sistema origen se dispone de la tabla A, y en el destino también se dispone de la tabla A, ¿qué nos asegura que la información de A en el origen debe ser traspasada a la tabla A del destino? Puede haber ocurrido que la tabla A del origen se llame B en el destino, y esa información es determinante para la migración.

En este contexto, adquiere especial relevancia un algoritmo capaz de automatizar el proceso de comparación de esquemas conceptuales y que además no se limite a la mera comparación de

Silva J.F., Carsí J.A., Ramos I. Comparación automática de esquemas conceptuales orientados a objeto, Revista Ingeniería Informática - Revista electrónica del Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción - Chile. 2002.

árboles, sino que aproveche información semántica de los esquemas durante la comparación para resolver posibles ambigüedades. En el contexto de migración de datos, este trabajo se corresponde con la primera fase a realizar por la herramienta de migración.

En el apartado 2 se presenta el problema de la comparación de esquemas conceptuales y su estrecha similitud con la comparación de árboles simples. Posteriormente, en el apartado 3, veremos cómo han tratado diversos autores el problema de la comparación, y mostraremos el hecho de que los algoritmos presentados hasta el momento no son adecuados para tratar el problema en contextos de migración de datos. En el apartado 4, se presenta un algoritmo que resuelve el problema de la comparación de esquemas conceptuales, y que es capaz de generar automáticamente las correspondencias que servirán de base para la generación automática del plan de migración [Per01]. Finalmente, en el apartado 5 se extraerán las conclusiones fruto del estudio realizado.

2. Presentación del problema

Se pretende obtener un algoritmo capaz de automatizar la comparación de esquemas conceptuales con el fin de migrar las instancias almacenadas entre las bases de datos de dichos esquemas. Dicho algoritmo, tendrá una doble utilidad; por un lado servirá para medir el grado de evolución existente entre dos esquemas; puesto que comparar implica medir, podremos medir aspectos como la complejidad ciclomática de los servicios, el número de clases, el número de atributos de las clases, nivel de herencia, etc. Por otro lado, cuando los esquemas conceptuales que se comparan sean modelos de un mismo sistema, el algoritmo servirá para identificar la evolución individual que ha tenido cada uno de los elementos de dichos esquemas, y de esta manera indicará los orígenes y los destinos de los datos a una herramienta de migración.

Algunas herramientas comerciales son capaces de identificar la evolución entre los esquemas sin necesidad de comparar, sino que guardan una traza de las operaciones de evolución realizadas. Esta solución tiene el problema de ser poco flexible puesto que obliga a que los dos esquemas que se comparan sean dependientes entre sí, es decir, uno sea la evolución del otro. El hecho de plantear una comparación de esquemas, permite que los esquemas que se comparan sean independientes entre sí, e incluso que tengan distinto formato. Además, la solución de la traza de servicios no es adecuada en un contexto de evolución de datos, puesto que, en la práctica, la mayoría de las migraciones que se desarrollan son realizadas entre sistemas antiguos , que no tienen un EC asociado, y que por lo tanto no existe una traza de evolución entre el modelo antiguo y el nuevo. En estos casos, se ha propuesto modelar el sistema legado a partir de su base de datos mediante un proceso de ingeniería inversa, y posteriormente realizar el proceso de comparación. Se extrae la conclusión de que existen casos, en los que es necesario aventurarse en un proceso de comparación, y por lo tanto es necesario un algoritmo específico de comparación de EC.

El algoritmo ha de ser capaz de generar automáticamente las diferencias existentes entre dos esquemas cualesquiera. Dichas diferencias vendrán dadas en términos de inserciones, modificaciones y borrados. Desde el punto de vista de la teoría de grafos, el problema que se nos plantea es un problema de comparación de grafos acíclicos no-dirigidos, es decir, de árboles. En nuestro caso, el objetivo que se persigue es migrar las instancias almacenadas de los elementos de los esquemas, por lo tanto sólo es necesario considerar aquellos elementos que tienen repercusión directa sobre la persistencia de los datos. Es necesario estudiar cuáles son esos elementos: Clases, Atributos y Relaciones de Especialización y Agregación [Car99]. A efectos de comparación, un esquema conceptual será considerado como un árbol de tres niveles de profundidad. Ver figura 1.

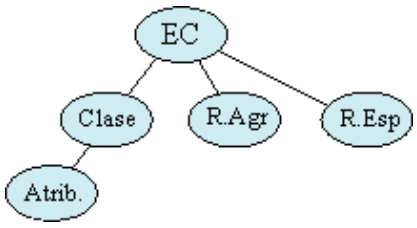


FIGURA 1: Representación de esquemas conceptuales en árboles.

En la figura 1 podemos ver que cada elemento del esquema conceptual queda representado como un nodo del árbol; siendo el nodo raíz el propio esquema conceptual.

Debemos fijar ahora una meta en la comparación, ¿qué esperamos obtener de la comparación de esquemas conceptuales? En un principio, podríamos pensar que lo más interesante es encontrar un conjunto de transformaciones (inserciones, modificaciones y borrados) del esquema conceptual origen, que nos lleven al esquema conceptual destino; y que además, esas transformaciones sean mínimas, es decir, queremos pasar de un árbol a otro realizando el menor número de transformaciones posibles. Nos encontraremos entonces ante un problema de optimización.

Realmente, no queremos un camino mínimo que nos lleve de un árbol a otro; lo interesante es tratar de identificar correctamente qué evolución ha tenido cada uno de los elementos del esquema conceptual inicial, sin importarnos el coste que pueda tener la transformación entre árboles. En este caso, el proceso algorítmico a utilizar es radicalmente diferente, y lo importante es identificar cada uno de los elementos de cada árbol y establecer una función biyectiva entre aquellos elementos identificados como correspondientes (dos elementos pertenecientes a dos esquemas distintos son correspondientes, un elemento se deriva del otro, si existe una correspondencia entre ambos elementos). A dicha función biyectiva la llamamos 'correspondencia'; a continuación se presenta la definición formal de correspondencia:

Correspondencia entre árboles: Intuitivamente, denota cómo una secuencia de operaciones transforman un árbol T en otro T', sin importar el orden en que se han aplicado las operaciones. Formalmente, una correspondencia entre árboles es expresada mediante la tripleta (M, T, T') donde T es el árbol inicial, T' el final, y M un conjunto de pares de enteros (i, j) que satisfacen:

1) $1 \le i \le |T|$, $1 \le j \le |T'|$;

2) Para cada par (i_1,j_1) y (i_2,j_2) en M se cumple:

a) $i_1 = i_2$ si y sólo si $j_1 = j_2$

b) $i_1 < i_2$ si y sólo si $j_1 < j_2$

c) $T[i_1]$ es un antecesor (descendiente) de $T[i_2]$ si y sólo si

$T'[j_1]$ es un antecesor (descendiente) de $T'[j_2]$.

Correspondencia entre elementos: Llamaremos correspondencia entre elementos (o simplemente correspondencia) a cada pareja (i,j) ☐ M de una correspondencia entre árboles.

Los objetivos principales que fijamos para este trabajo, son los siguientes:

1. Obtener un algoritmo de comparación de esquemas conceptuales que trate de identificar correctamente qué evolución ha tenido cada uno de los elementos del esquema

conceptual inicial.

2. Que el algoritmo sea capaz de establecer una correspondencia para cada elemento de los esquemas conceptuales.

3. Que el algoritmo no se limite a la comparación de árboles, sino que sea específico para EC; que utilice información semántica para comparar.

4. Que los EC a comparar no tengan por qué ser dependientes entre sí.

A continuación, vamos a realizar un estudio algorítmico que solucione el problema atendiendo a cada uno de los objetivos establecidos. Abordaremos en primera instancia, el problema del camino mínimo entre árboles, y seguidamente, el problema de la identificación de elementos entre esquemas conceptuales.

3. Estado del arte: El problema 'Árbol a Árbol'

El problema de la obtención de diferencias entre esquemas, es tratado actualmente por diferentes herramientas comerciales utilizando una traza de las operaciones de evolución. Este es el caso de VDIFF de Rational Rose. El principal problema que plantean estas herramientas es que los EC deben ser dependientes. Además, el hecho de que deba existir una traza de evolución entre los esquemas obliga a que los esquemas sean del mismo formato o pertenezcan a la misma aplicación. Estos problemas desaparecen con un proceso de comparación.

No se han encontrado en la bibliografía algoritmos específicos de comparación de esquemas, sin embargo sí existen diversos algoritmos para comparar árboles etiquetados, en los que se plantea el problema de encontrar la distancia mínima entre los mismos. Aunque en la mayoría de los casos no tendrá sentido plantear la comparación de esquemas como un problema de optimización, puede haber situaciones en las que sea interesante conocer el camino mínimo entre dos árboles. En cualquier caso, necesitaremos del conocimiento de estos algoritmos para el desarrollo de los presentados en el siguiente apartado; y además nos establecerán una cota inferior en el paso de un árbol a otro (estos algoritmos encuentran el camino mínimo entre los dos árboles).

Muchas estructuras y procesos pueden modelizarse como árboles etiquetados, y en ocasiones puede ser muy útil conocer cuál es la forma de pasar (o transformar) de un árbol a otro con el menor número de cambios posibles. Un caso particular de este problema ocurre cuando los árboles son de profundidad dos. Este es el caso de tratar de comparar dos cadenas de caracteres donde cada carácter está representado por un nodo que guarda una determinada posición entre los nodos hoja. Este problema fue abordado y resuelto por diversos autores. Sankoff [San72] y Wagner y Fisher [Wag74] presentaron un algoritmo capaz de computar una secuencia de operaciones de edición de mínimo coste. Dicho algoritmo presentaba un coste computacional O(n*m), siendo m y n el número de hojas de cada uno de los árboles. Posteriormente, Wong y Chandra [Won76]; y Aho, Hirschberg y Ullman [Aho76] probaron que para muchos modelos de computación el algoritmo de Sankoff es óptimo. Otro algoritmo capaz de solucionar el problema, también con coste óptimo, fue presentado en 1977 por Selkow [Sel77] en Knoxville.

Una generalización del problema, es considerar que los árboles pueden tener cualquier profundidad. En este caso, el problema aumenta su complejidad en un factor cuadrático con relación a la misma. Vamos a presentar a continuación un algoritmo que resuelve el problema y que está atribuido a Kuo-Chung Tai [Kuo79].

Partimos de la siguiente información:

- **r**: Función arbitraria de coste que asigna un valor no negativo a cada operación de edición ($a \rightarrow b$).

- **D(T, T')**: Distancia mínima entre dos árboles; formalmente, min{r(S) / S es una secuencia de operaciones que transforma T en T'.
- Coste(M): Representa el coste asociado a una correspondencia M. Dicho coste es función del número de inserciones, modificaciones y borrados.
- **MIN_M(i + 1, j + 1)**: min{coste(M) / M es una correspondencia de T(1:i+1) a T'(1:j+1), y (i+1, j+1) ∈ M}.
- **E[s: u: i, t: v: j]**: min{coste(M) / M es una correspondencia de T(s: i) a T'(t: j), y (s, t) ∈ M, y ningún descendiente de T[s] (T'[t]) en el camino de T[s] (T'[t]) a T[u] (T'[v]) es tocado por una línea de M}.

El algoritmo se compone de tres pasos:

1. Calcular E[s: u: i, t: v: j] para todo s, u, i, t, v, j, donde

   $1 \le i \le |T|$, $1 \le j \le |T'|$,

   T[u] (T'[v]) está en el camino de T[1] (T'[1]) a T[i] (T'[j]),

   T[s] (T'[t]) está en el camino de T[1] (T'[1]) a T[u] (T'[v]);

2. Calcular MIN_M(i, j) para todo i, j, donde $1 \le i \le |T|$, $1 \le j \le |T'|$;

3. Calcular D(i, j) para todo i, j, donde $1 \le i \le |T|$, $1 \le j \le |T'|$.

Dados dos árboles T y T' el algoritmo propuesto por Kuo-Chung Tai, computa la distancia entre ambos árboles con un coste asociado O(V * V' * L2 * L'2), donde V y V' son los números de nodos de T y T' respectivamente, y L y L' son los niveles de profundidad máximos de cada árbol.

4. Análisis algorítmico basado en información semántica

El algoritmo de Selkow encuentra el camino mínimo entre dos árboles de profundidad dos, y el de Kuo-Chung Tai, calcula el mínimo para árboles de cualquier profundidad. Tenemos, por tanto, una cota inferior del coste de un algoritmo de comparación de esquemas conceptuales. Podríamos pensar que utilizar el algoritmo de Kuo-Chung Tai para comparar EC es la solución al problema; sin embargo, ese algoritmo no fue pensado para comparar EC, sino árboles simples, y desaprovecha mucha información de los esquemas que pueden mejorar el resultado de la búsqueda.

En el marco de la comparación de esquemas conceptuales para la generación automática de planes de migración, se dispone de mucha información semántica que puede ser aprovechada por los algoritmos de comparación. Del mismo modo, la información que los algoritmos generan durante la comparación de dos elementos simples puede ser reutilizada en futuras comparaciones del resto de elementos del mismo esquema. En este apartado, vamos a analizar cómo se puede aprovechar toda esa información para desarrollar un algoritmo de comparación de esquemas conceptuales.

El concepto de distancia entre dos árboles, puede contraponerse a la medida de similitud entre árboles. En un contexto de evolución, dicha distancia puede servir para medir el grado de evolución que ha existido entre dos esquemas conceptuales, los cuales representaremos como árboles. No obstante, el cálculo del camino mínimo entre dos árboles, no es una buena solución para identificar la evolución sufrida por esquemas conceptuales; en la figura 2 se muestra un ejemplo muy simple que deja al descubierto el inconveniente que se plantea.

Una comparación de esquemas que busque el camino mínimo entre los dos árboles, daría como solución el cambio de nombre de la clase 'Coche' por 'Moto'; cuando realmente, la clase 'Coche'

ha sido eliminada, y la clase 'Moto' creada, puesto que se trata de clases distintas. En este caso, no debería existir una correspondencia entre coche y moto, puesto que no deben migrarse las instancias de uno a otro. El hecho de que la búsqueda de un conjunto mínimo de operaciones no sea adecuada para comparar esquemas conceptuales, se debe a que la evolución de esquemas conceptuales sigue patrones de evolución aleatorios, que no tienen porqué coincidir con caminos mínimos. Mientras que los algoritmos de optimización encuentran la forma más rápida de pasar de un árbol a otro, los algoritmos del siguiente apartado tratarán de encontrar métodos alternativos para comparar esquemas conceptuales que intenten identificar qué evolución real han tenido las instancias de las clases. Por otro lado, los algoritmos utilizados hasta el momento no tenían en cuenta información semántica del modelo, ya que es posible aprovechar las propiedades de los esquemas para mejorar el proceso de comparación.

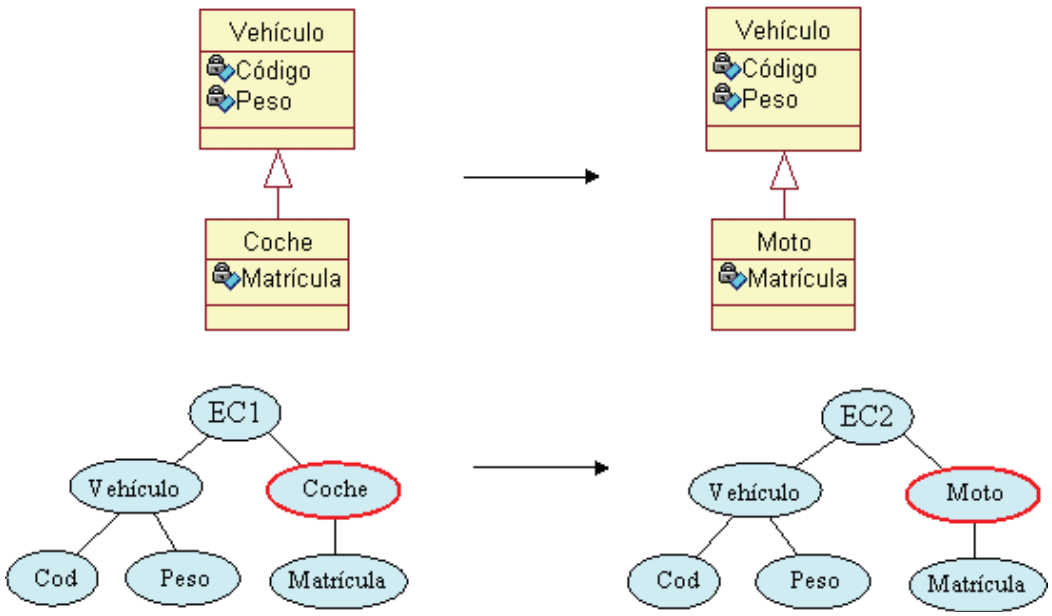


FIGURA 2: Comparación de esquemas conceptuales 'Árbol a Árbol'.

La aproximación 'Árbol a Árbol', puede ser adecuada cuando los esquemas conceptuales no tienen población asociada; pero en un contexto de evolución y migración de datos, los esquemas conceptuales están asociados a una población que debe evolucionar a la par que el esquema. En este tipo de contextos, la aproximación 'Árbol a Árbol' no suele ser adecuada porque no tiene en cuenta la evolución de población.

Para solucionar el problema, debemos buscar métodos alternativos de comparación, que tengan en cuenta la migración de población, y que traten de fijar como meta identificar qué hizo el analista con cada elemento de los esquemas cuando llevó a cabo la evolución. Necesitamos encontrar un algoritmo que compare los esquemas aprovechando, no sólo su estructura de árbol, sino también sus propiedades de esquemas conceptuales. Además, es muy importante que el algoritmo de comparación utilice criterios que traten de identificar qué evolución pretende el analista.

No hay que olvidar que una de las premisas iniciales era que ambos árboles debían estar etiquetados, la cuestión es ¿qué información contienen las etiquetas de los nodos? En un esquema conceptual la información disponible para comparar es muy rica (identificadores, nombres, relaciones con otros elementos, número de atributos en el caso de las clases, etc.) y dicha información puede determinar en gran medida el éxito de la comparación. En [Sil01] se analizan con detalle los diversos criterios de comparación de esquemas conceptuales.

El hecho de comparar dos esquemas conceptuales como si fueran sólo árboles desaprovecha mucha información semántica. Un algoritmo de comparación de esquemas conceptuales debe

aprovechar su estructura de árbol, pero también el hecho de que no es meramente un árbol, sino que se trata de un esquema conceptual. Esta idea puede aportar información muy rica a la hora de comparar. Los algoritmos del apartado anterior desaprovechaban esta información semántica y por ello trataban a todos los nodos del árbol de la misma forma. A priori, podríamos pensar que para comparar los elementos de los esquemas conceptuales deberemos comparar todos con todos, como hacían los anteriores algoritmos... ¿todos con todos? En un árbol-esquema conceptual, podemos distinguir cuatro tipos de ramas:

- Clases
- Atributos (subrama de una clase)
- Relaciones de Agregación
- Relaciones de Especialización

Es evidente, que no tiene sentido comparar clases con atributos, sino que habrá que comparar elementos de la misma naturaleza, esto es clases con clases, atributos con atributos, relaciones de agregación con relaciones de agregación y finalmente relaciones de especialización con relaciones de especialización.

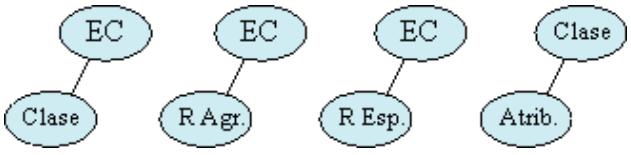
Partiendo de esta idea, podemos mejorar sustancialmente los algoritmos generales de comparación de árboles; para ello vamos a utilizar una técnica de fragmentación, con el fin de partir los árboles en subárboles comparables.

4.1 Mejora de la propuesta 'Árbol a Árbol' mediante fragmentación en subárboles

Dados dos árboles etiquetados cualesquiera, siendo V y V' sus números de nodos, y siendo L y L' sus respectivas profundidades, el algoritmo de Kuo-Chung Tai calcula el camino mínimo en términos de Inserciones, Modificaciones y Borrados con un coste:

Si representamos un esquema conceptual en forma de árbol, tal y como aparece en la figura 1, el coste aplicando el algoritmo de Kuo sería: Coste1 = V·V'·4·4 = 16·V·V' $\in$ O(V·V')

Este coste puede mejorarse si aprovechamos la idea de que el árbol está representando un esquema conceptual, y cada subárbol tiene unas características distintas. Por lo tanto, si fragmentamos el árbol podríamos reducir L y L':



El coste sería: Coste2 = (V1·V'1) + (V2·V'2) + (V3·V'3) + (V4·V'4) $\in$ O(V·V')

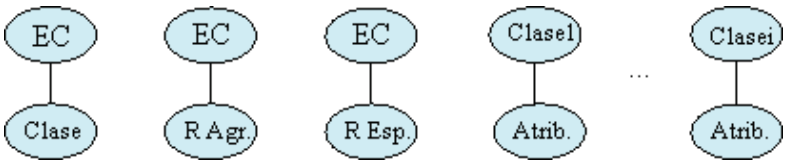
Como V = V1 + V2 + V3 + V4 y V' = V'1 + V'2 + V'3 + V'4

entonces

Coste1 = V1·V'1 + …+ V1·V'4 + V2·V'1 + …+ V2·V'4 +…+ V4·V'1 +…+ V4·V'4 > Coste2

Se demuestra que coste1 > coste2.

Podemos mejorar el coste todavía más si continuamos fragmentando los subárboles:



El coste sería:

Coste3 = (V1·V'1) + (V2·V'2) + (VC1·VC'1) + … + (VCn·VC'n) $\in$ O(V·V') < Coste2

Se demuestra que coste1 > coste2 > coste3.

Otro tipo de información semántica que conviene tener en cuenta es el orden de comparación. No tiene sentido comparar atributos, sin antes haber identificado las clases (*nodos padres en el árbol*) que contienen esos atributos. La comparación de esquemas conceptuales ha de seguir un orden lógico, dicho orden no es un orden prefijado, sino que depende del criterio de comparación que vaya a aplicarse. Por ejemplo, supongamos que queremos comparar dos esquemas con un criterio de comparación basado en OID, el orden lógico de comparación sería el siguiente:

- Análisis de clases (parte 1).
- Análisis de atributos.
- Análisis de clases (parte 2).
- Análisis de relaciones.

El primer paso será identificar qué clases del esquema conceptual inicial han sido borradas, y qué clases del final han sido insertadas; el resto de clases consideraremos que permanecen invariantes (*Pueden haber sido modificadas, pero son clases correspondientes*). Este es a lo que llamamos el análisis de clases (parte 1). A continuación, procederemos a analizar los atributos de cada clase invariante (sólo de las invariantes, porque los atributos de las inserciones son todos nuevos, y los de los borrados han sido todos eliminados) para identificar los insertados, modificaciones, borrados e invariantes. Esta parte constituye el análisis de atributos. A partir del análisis de atributos podremos determinar qué clases han sufrido modificaciones, y cuáles son realmente invariantes. Finalmente podremos analizar las relaciones entre clases, puesto que ya tendremos totalmente identificadas las mismas.

Cuando aplicamos la información semántica anterior, nos damos cuenta de que ya no tenemos un árbol que comparar, sino muchos de profundidad uno. Podríamos pensar que el algoritmo de Selkow se ajusta perfectamente al problema, pero no es así. Dicho algoritmo tiene en cuenta a la hora de comparar el orden de los nodos del árbol, y este hecho a nosotros nos es totalmente indiferente; esto nos permite relajar todavía más las restricciones originales.

A continuación, vamos a presentar dos versiones del algoritmo que resuelve el problema:

Partimos de la siguiente información:

A, B: Listas con los elementos del árbol T y T' respectivamente.

Criterio(Ai, Bj)**:** Devuelve CIERTO si Ai y Bj son elementos iguales según algún criterio de comparación preestablecido.

Emparejar(Ai, Bj)**:** Introduce Ai y Bj en una lista de parejas de elementos, y los saca de las respectivas listas de nodos A y B.

Borrado(Pa): Inserta en la lista de elementos borrados el elemento Pa.

Inserción(Pa): Inserta en la lista de elementos insertados el elemento Pa.

Ordenar(A): Ordena la lista A.

Algoritmo (versión 1):

Procedure (A, B);

(Asumimos que A y B tienen como

subárboles $A_1,...,A_m$; $B_1,...,B_n$ respectivamente)

For i = 1, 2, …, m **do**

For j = 1, 2, …, n **do**

Begin

If Criterio(Ai, Bj) = CIERTO **then**

Emparejar(Ai, Bj);

Exit; /* Sale del bucle interno */

End if;

End;

End Procedure.

Algoritmo (versión 2):

Procedure (A, B);

(Asumimos que A y B tienen como

subárboles $A_1,...,A_m$; $B_1,...,B_n$ respectivamente)

Ordenar(A);

Ordenar(B);

$P_a = A_{i;}$ $P_b = B_{j;}$ /* Punteros a los elementos de los vectores */

While((P_a < Length(A_i)) **and** (P_b < Length(B_j))

If Criterio(Pa, Pb) = CIERTO **then**

Emparejar(Pa, Pb);

$P_a = P_a+1;$ $P_b = P_b+1;$

Else If P_a < P_b **then**

Borrado(Pa);

$P_a = P_a+1;$

Else If $P_a > P_b$ **then**

Inserción(Pb);

$P_b = P_b+1;$

End if;

End While;

/* Completar resto de la lista en inserciones o borrados */

End Procedure.

Al finalizar el algoritmo dispondremos de tres listas. La primera lista contendrá todos los elementos que han sido eliminados del esquema conceptual inicial; la segunda lista contendrá todos los elementos que han sido insertados en el esquema conceptual final; y la tercera lista contendrá todos los elementos que se mantienen tras la evolución entre los dos esquemas. Es fácil darse cuenta de que el algoritmo es totalmente dependiente del criterio de comparación elegido, lo cual es lógico, porque el criterio adoptado determinará cuándo dos elementos son iguales; Este hecho es muy importante, dado que no todos los criterios son igual de buenos, ni existe un criterio que sea siempre mejor que los demás.

La primera versión del algoritmo tiene un coste asociado O(m*n) (*Estos costes no incluyen el coste asociado al criterio de comparación*); dicho coste es mejorado por la segunda versión. Ha sido presentado porque en él se ve claramente la dependencia del coste respecto del criterio de comparación utilizado. En cualquier caso, siempre debemos tener en cuenta que el número de elementos a comparar no superará el centenar en la mayoría de los casos, con lo que la diferencia de coste se ve sustancialmente disminuida. La segunda versión del algoritmo tiene un coste asociado O(n*log(n)) (*Realmente, el coste sería n*log(n) + m*log(m) + n + m : las dos primeras componentes se derivan de la ordenación de las listas (utilizando por ejemplo 'mergesort' o 'quiksort'); la tercera y cuarta componentes del coste proceden del recorrido de las listas.*), dado que ordena las listas antes de su tratamiento.

Es importante hacer notar que el criterio de comparación puede tener una complejidad elevada si utiliza la información que va obteniendo el algoritmo durante la comparación. Aunque este tipo de criterios presenta mayores costes y su complejidad ciclomática es de un orden superior (debido a su interdependencia con el propio algoritmo de comparación), este será en la mayoría de los casos el criterio de comparación apropiado, puesto que por termino medio, presenta una mejoría sustancial en sus tasas de evaluación. En [Sil01] puede encontrarse un análisis completo sobre este tipo de criterios.

El algoritmo propuesto es el núcleo de una herramienta de comparación automática de esquemas conceptuales que ha sido desarrollada para un entorno de migración automática de bases de datos. En ese contexto, la herramienta es capaz de identificar automáticamente los orígenes y los destinos de la migración.

5. Conclusiones

La comparación de esquemas conceptuales es un proceso mucho más rico que la comparación simple de árboles. Los algoritmos optimizados de comparación de árboles pueden utilizarse para comparar esquemas; pero también pueden mejorarse utilizando información semántica:

- Existen cuatro tipos de elementos comparables distintos en un esquema conceptual. Sólo

deben compararse elementos pertenecientes a la misma clase de equivalencia.
- Cada elemento de un esquema conceptual es único y está perfectamente identificado: La primera comparación exitosa del algoritmo podrá cortar la búsqueda de ese elemento.
- A diferencia de la comparación simple de árboles, los algoritmos de comparación de esquemas conceptuales pueden aprovechar dinámicamente la información semántica que van generando durante el propio proceso de comparación.
- El algoritmo de comparación debe estar regido según un orden lógico de comparación.
- El criterio de comparación determinará en gran medida el éxito final de la comparación.

El objetivo final de la comparación no es tanto la búsqueda de caminos mínimos, como la identificación de la evolución de cada elemento. Finalmente, la comparación de esquemas conceptuales no es un problema de optimización.

6. Bibliografía

[San72] D. Sankoff, Matching sequences under deletion/insertion constraints, Proc. Nat. Acad. Sci. 1972, 4-6.

[Wag74] R.A.Wagner y M.J. Fisher, The string to string correction problem, J. Assoc. Comput. 1974, 13-16.

[Won76] C.K. Wong y A.K. Chandra, Bounds for the string editing problem, J. Assoc. Comput. 1976, 168-173.

[Aho76] A.V. Aho, D.S. Hirschberg y J.D. Ullman, Bounds on the complexity of the longest common subsecuence problem, J. Assoc. Comput. 1976, 1-12.

[Sel77] S. Selkow, The tree-to-tree editing problem. Dep. Com. Sci., University of Tennessee, Knoxville, 1977.

[Kuo79] Kuo-Chung Tai, The tree-to-tree correction problem, Dep. Com. Sci., North Carolina State University, Raleigh, 1979.

[OOM98] Ramos I., Pastor O., OO-Method: Una metodología OO para la producción automática de software., DSIC, UPV, 1998.

[Gra98] Grau A., Computer-Aided Validation of formal conceptual models, Tesis Doctoral, Institute For Software, Information Systems Group, Technical University of Braunschweig, 1998.

[OAS98] Letelier P., Ramos I., Sánchez P. Pastor O., Oasis 3.0: Un enfoque formal para el modelado conceptual orientado a objeto, SPUPV-98.4011, 1998.

[Ana99] Anaya R., Desarrollo y gestión de componentes reutilizables en el marco de Oasis, Tesis Doctoral, Facultad de Informática, Universidad Politécnica de Valencia, Octubre 1999.

[Car99] Carsí J.A., OASIS como marco conceptual para la evolución del software, Tesis Doctoral, DSIC, UPV, 1999.

[Mor00] Moreno M., García F., Polo M., Medición de la calidad del software en el ámbito de la especificación de requisitos , Universidad de Salamanca, Departamento de informática y automática 2000.

[Sta00] Staudt B., A model for compound type changes encountered in schema evolution, University of Massachusetts, Amherst, ACM Transactions on Database Systems, Vol. 25, No. 1, March 2000, Pages 83–127.

[Dol00] Dolado J.J., Fernández L., Medición para la gestión en la ingeniería del software. RA-MA, 2000.

[Var00] Varas M., Pradenas J., Hacia la definición de métricas de calidad para esquemas conceptuales de bases de datos., revista electrónica del DIICC, edición nº 6, 2000.

[Sil01] Silva J., Ramos I., Carsí J.A., Análisis teórico/experimental de criterios de comparación de esquemas conceptuales a efectos de la generación automática de planes de migración, enviado a las JISBD,2001.

[Per01] Pérez J., Anaya V., Silva J., Carsí J.A, Ramos I., Generación Automática de un plan de migración entre poblaciones de esquemas conceptuales orientados a objetos, Distributed Objects Languages Methods & Environments, DOLMEN, Sevilla 2001.

[Cal01] Calero C., Definición de un conjunto de métricas para la mantenibilidad de bases de datos relacionales, activas y objeto-relacionales, Tesis Doctoral, Universidad de Castilla-La Mancha, 2001.

Electronic Notes in Theoretical Computer Science 74 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume74.html 13 pages

Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas

⋆

J.Pérez [1], I.Ramos [2], V.Anaya [3], J.M.Cubel [4], F.Domínguez [5], A.Boronat [6], J.A.Carsí [7]

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46071 Valencia - Spain

Abstract

This paper presents a solution and a methodology to recover legacy databases of most DBMS using formal-method based techniques. These formal methods (terms rewriting systems) are applied during the data reverse engineering process and allow for an automatic approach. This automatic approach reduces the time invested and the number of people involved in the data reverse engineering and data migration processes. This solution is being implemented in the RELS (Re-Engineering of Legacy Systems) tool. The RELS tool is under development in the Department of Information Systems and Computation of the Valencia University of Technology in collaboration with the industrial partner CARE-Technologies. RELS is used together with the model compiler Sosy Technology® of CARE-Technologies and provides a complete solution to the re-engineering proccess.

Key words: Re-engineering, Reverse engineering, Terms rewriting system (TRS), Data reverse engineering, Rewriting rules, Algebraic expressions, ADT(Abstract Data Type).

⋆ This work has been financed by the project CICYTof the program SIGLO with ref. TIC2000-1673-C06-01.

[1] Email: jeperez@dsic.upv.es
[2] Email: iramos@dsic.upv.es
[3] Email: vanaya@dsic.upv.es
[4] Email: jcubel@dsic.upv.es
[5] Email: fdominguez@dsic.upv.es
[6] Email: aboronat@dsic.upv.es
[7] Email: pcarsi@dsic.upv.es

©2002 Published by Elsevier Science B. V.

Pérez J., Ramos I., Anaya V., Cubel J.M, Domínguez F., Boronat A., Carsí J.A. Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. Proc. of the Workshop on Software Evolution Through Transformations (SET'02) at ICGT2002. 2002.

1 Introduction

Technologies have a dynamic nature and they are continually evolving. For this reason, the systems must be able to adapt to new requirements. However, their development tools, DBMS or programming languages rapidly become obsoletes. As a consequence, these information systems become *Legacy Systems*. The recovery of legacy systems is a big problem for companies due to the cost of technological adaptation. In this paper, we propose a methodology and a tool to reduce this expense.

Our work is focused in legacy databases. For this reason, we apply *Data reverse engineering (DBRE)* to recover the obsolete databases. Data reverse engineering is a part of the Reverse engineering process. "Data Reverse Engineering deals with the tasks of understanding legacy databases and extracting their design specifications (domain semantics)" [2].

Nowadays, existing CASE tools are able to generate applications following the paradigm of automatic code generation. These CASE tools are called model compilers. They automatically generate the application code and the database schema starting from the conceptual schema of an information system.

Our tool recovers a legacy database obtaining an equivalent UML-like OASIS OO conceptual schema using formal methods and then, automatically generates a new SQL/Server database from the schema obtained. Finally, the data from the legacy database are correctly migrated to the new one (see Figure 1).

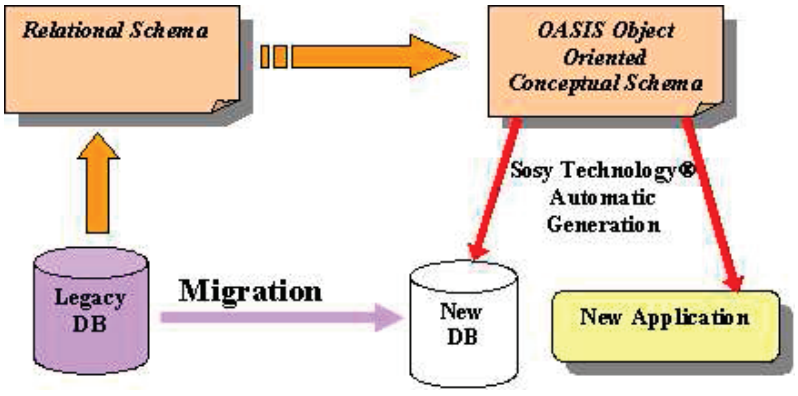


Fig. 1. Re-engineering process of RELS

This work is being developed in the Department of Information Systems and Computation of the Valencia University of Technology in collaboration with the industrial partner CARE Technologies. The data reverse engineering process and the data migration process followed by our approach will reduce the time invested and the number of people involved in the data evolution process. This maintenance improvement is due to the automatic tasks performed by the three phases of the tool. Despite the fact that these phases are

performed automatically, the results can be freely modified by the analyst. In this case, the process is semi-automatic. The three phases involved are the following:

1.- An OASIS OO conceptual schema is obtained by applying a data reverse engineering process in order to recover a legacy database. The relational and OO conceptual schemas are represented as terms which belong to two different term languages of two different ADTs. The correspondences between terms are specified using term rewriting rules. This paper is focused in this first phase.

2.- The rewriting rules applied in the first phase and the patterns used by Sosy Technology® to generate the new relational database are used to generate a data migration plan which is specified using a declarative language (a language which expresses properties of WHAT to do and not HOW to do it).

3.- The data migration plan is translated to *Data Transformation Service* packages of Microsoft whose execution automatically migrates data from the old database to the new database. These packages are finally executed by *SQL Server*.

Our work emphasizes the use of formal methods because they allow us to provides the user an automatic recover solution that is used by an industrial model compiler to improve the software maintenance process.

The structure of the paper is the following: Section 2 describes the data reverse engineering process. Section 3 presents how we mix the formal methods with current technologies in our implementation. Section 4 exemplifies the process of recovering a legacy database by means of an example. Section 5 provides a brief summary of related work. Finally, section 6 presents the future work and conclusions.

2 Data Reverse Engineering Process of RELS

The Data Reverse Engineering Process of RELS takes the relational model of the legacy database as input and generates an object-oriented model which is equivalent with the previous relational one. These models are represented as terms of the ADTs (Abstract Data Types) defined for the relational and object-oriented models.

The automatic generation of the object-oriented model is produced by applying the rewriting rules representing the correspondences between the relational elements [8] and the object-oriented elements [9]. These rewriting rules are applied by a term rewriting system (TRS) ([4]). The data reverse engineering process of RELS is constituted by three steps (see Figure 2):

[8] Elements that can be defined in a relational model
[9] Elements that can be defined in a object-oriented model

1.- The reading of the relational schema of a legacy database and its representation as a term of the defined relational ADT.

2.- The translation of the relational term into an object-oriented term using a TRS that applies the rewriting rules. The obtained object-oriented term which represents the object-oriented schema is compliant with the defined object oriented ADT.

3.- The storage of the object-oriented schema and the applied rewriting rules that will be used in the second phase of RELS.

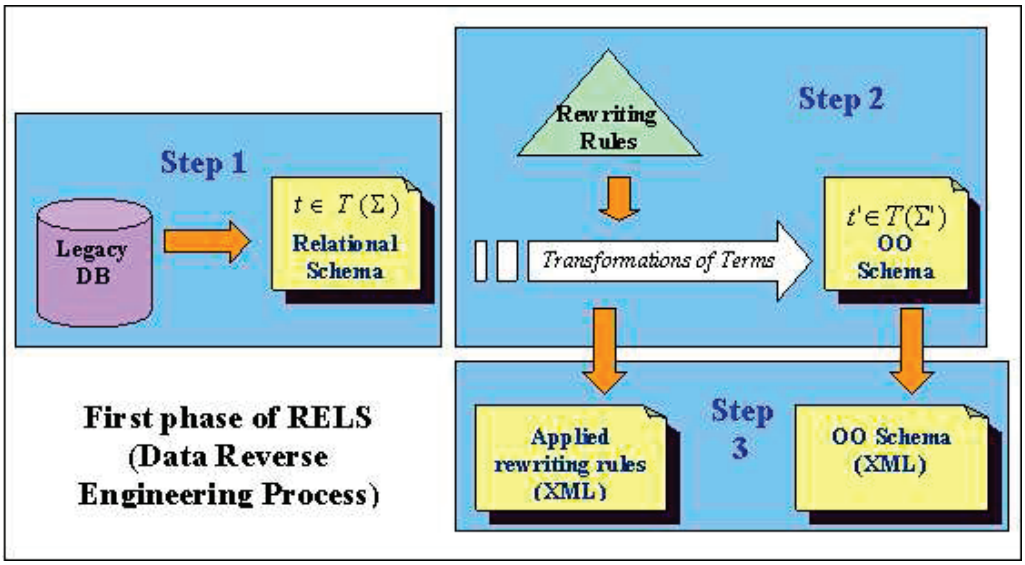


Fig. 2. Data Reverse Engineering Process of RELS

2.1 Relational Model

The relational conceptual schema of the legacy system is captured from the stored database of a specific DBMS. This structure is obtained by using a set of operations that allows it to read a relational schema specification. The relational conceptual schema is represented as an algebraic term which is based on the syntactic and semantic rules provided by an ADT. For this reason, an ADT which expresses the relational model is proposed. This ADT is composed by a group of specification modules which contain operations (constructors or functions) and axioms that determine which relational elements are considered and how they can be combined to express a correct relational schema. A specification module is associated with any relevant element of the relational model. There is a natural compositional order between elements of the relational model (for example, tables are composed by columns), so there is a compositional relation between the specification modules of the relational ADT. Therefore, a well structured ADT is obtained.

In each specification module, a set of constructors, functions and axioms

are provided. The constructors are applied to form a term. A module, which is composed of other modules, combines all the subterms and carries out the term that expresses the relational conceptual schema.

Once the relational conceptual model term is obtained, some user interaction may be required. Some legacy systems are very old and were constructed using DBMS or other repository forms which did not allow for the definition of constraints (either integrity or reference constraints). These constraints were implemented by code, so they are not explicitly presented in the legacy database structure. User interaction may be necessary to provide additional information to obtain a complete relational conceptual schema. Users can also express the formulae of derived attributes. The subprocess of enriching the conceptual schema is done over the relational term, but our tool has an interface that hides the algebraic notation, providing a friendly graphical user interface.

```
SPEC   m-rel
USING  table + column + data_type + nut_nul + ...
SORTS  m-rel
OPS    create_database: --> m-rel
       add_table: table m-rel --> m-rel
       add_column: column data_type not_null table m-rel --> m-rel
       add_ctr_pk: col_list table m-rel --> m-rel ...
ENDSPEC
```

Fig. 3. Part of the Relational ADT

2.2 Object-Oriented Model

After applying the rewriting rules to the relational conceptual schema term, an object-oriented term is obtained. This OO term is expressed by means of the constructor operations and by using the axioms provided by the ADT that represent the OASIS OO Model. The OO ADT specification modules correspond to OO elements and properties of these elements. A module that is composed of other modules expresses how to generate terms that combine the rules of its components.

Hence, the m-oo (OO model specification module) provides the rules to generate OO conceptual schema terms. The OO conceptual schema term is then translated to a XML document that can be interpreted by the SOSY Technology® model compiler.

2.3 Correspondences between the Relational Model and the Object-Oriented Model

A detailed study has been done to obtain the correspondences between the relational ADT and the object-oriented ADT. We have determined for each element of the relational ADT, the different elements of the Object-Oriented ADT that can be equivalent. This has allowed us to define the transformations

```
SPEC   m-oo
USING  class + aggregation + specialization + ...
SORTS  m-oo
OPS    create_schema: --> m-oo
       add_class: class m-oo --> m-oo
       add_identif: attr_list class m-oo --> m-oo
       add_ctt_att: attribute data_type bool class m-oo --> m-oo
       add_vbl_att: attribute data_type bool class m-oo --> m-oo
       add_aggregation: aggregation class class rol rol nat nat
                        nat nat bool bool bool bool m-oo --> m-oo ...
ENDSPEC
```

Fig. 4. Part of the Object Oriented ADT

(set of rewriting rules) to convert a relational term into an object-oriented term.

Our TRS is finite and non-confluent, because we can obtain several object-oriented terms from a relational term (different possible representations). The initial term is constituted by subterms of the relational ADT and the following terms (in the rewriting process) include terms of the object-oriented ADT. However, subterms of both ADTs coexist in the intermediate terms. Those intermediate terms belongs to the mixed ADT m-rel-oo.

The application of the rules is automatic, but the user's knowledge is necessary in order to validate whether the applied rule is semantically correct. This is due to the fact that an element of the relational model can be represented by several elements of the OO model. For this reason, the RELS' proposal of applying a specific rewriting rule can be changed by the user. This change is controlled by the tool; the user can only modify the proposed rewriting rule by using another rule from the set of rewriting rules that are syntactically correct in the following rewriting step. We take into account the criterion that legacy databases were usually designed to improve the efficiency of access to the data. This reduces the interactions required by the users.

3 Implementation

The data reverse engineering process of RELS uses several technologies in order to develop the different phases. This tool is under development using Visual Studio .NET because of its advantages and is one of the programming platforms for the future.

Moreover, an API has been defined to read the relational schema. This API has been created based on ADOX Object Model ((Microsoft ActiveX Data Objects Extensions for Data Definition Language and Security). From the services provided by the API, the application extracts the necessary information to obtain the relational term. RELS uses this API because it provides independence from the DBMS. As a result, RELS reads the relational schema from most legacy DBMS.

The relational and the object-oriented ADTs, the set of rewriting rules which can be applied, and the algebraic terms are specified using MAUDE

[7] (an OBJ dialect). MAUDE also provides the rewriting mechanism that executes the rules in order to transform the relational term into the object-oriented term. We chose HASKELL to implement this phase of the data reverse engineering process because is a functional language and is supported by Visual Studio .NET.

Finally, the outputs of this phase are stored as XML documents to be used by the second phase of RELS. Thus, there are two different XML DTDs to generate each document:

1.- The Sosy Technology® XML DTD: The generated object-oriented term is specified in a XML document which is compliant with this DTD.

2.- The rewriting rules XML DTD: The applied rewriting rules in a specific data reverse engineering process are stored in a XML document that follows this DTD.

The XML document that contains the object-oriented term is used by

```
SPEC   m-rel-oo
USING  m-rel + m-oo + ...
SORTS  m-rel-oo
SUBSORTS
       m-rel < m-rel-oo
       m-oo < m-rel-oo
OPS    ...
FOR ALL
       c, cr, cs, c1, c2, c3, c4, c5, c6, c7, c8: column ...
AXIOMS create_database() --> create_schema()
       add_column(c, d, n, t, m) -->
          add_vbl_att(c, d, n, cat("AGGR-",t, "-1"), m) IF aggregation(cat("AGGR-",t),m)
       add_column(c, d, n, t, m) -->
          add_vbl_att(c, d, n, cat("AGGR-",t, "-2"), m) IF aggregation(cat("AGGR-",t),m)
       add_column(c, d, n, t, m) -->
          add_ctt_att(c, d, n, cat("AGGR-",t, "-1"), m) IF aggregation(cat("AGGR-",t),m)
       add_column(c, d, n, t, m) -->
          add_ctt_att(c, d, n, cat("AGGR-",t, "-2"), m) IF aggregation(cat("AGGR-",t),m)
       add_unique(cr, t, add_unique(cs,t,add_column(c1, d1, n1, t, add_column(c2, d2, n2, t,
       add_column(c3, d3, n3, t, add_column(c4, d4, n4, t, add_column(c5, d5, n5, t,
       add_column(c6, d6, n6, t, add_column(c7, d7, n7, t, add_column(c8, d8, n8, t,
       add_table(t,m)))))))))) -->
          add_unique(cr,t, add_unique(cs, t, add_column(c1, d1, n1, t, add_column(c2, d2, n2, t,
          add_column(c3, d3, n3, t, add_column(c4, d4, n4, t, add_column(c5, d5, n5, t,
          add_column(c6, d6, n6, t, add_column(c7, d7, n7, t, add_column(c8, d8, n8, t,
          add_class(t, m)))))))))
       add_unique(cr,t, add_unique(cs,t, add_column(c1, d1, n1, t, add_column(c2, d2, n2, t,
       add_column(c3, d3, n3, t, add_column(c4, d4, n4, t, add_column(c5, d5, n5, t,
       add_column(c6, d6, n6, t, add_column(c7, d7, n7, t, add_column(c8, d8, n8, t,
       add_table(t,m)))))))))) -->
          add_unique(cr,t, add_unique(cs,t, add_column(c1, d1, n1, t, add_column(c2, d2, n2, t,
          add_column(c3, d3, n3, t, add_column(c4, d4, n4, t, add_column(c5, d5, n5, t,
          add_column(c6, d6, n6, t, add_column(c7, d7, n7, t, add_column(c8, d8, n8, t,
          add_aggregation(cat("AGGR-",t), cat(t,"-1"), cat(t,"-2"), cat("rol",t,"-1"),
          cat("rol", t,"-2"), 1, 1, 1, 1, false, true, false, false, add_class(cat(t,"-2"),
          add_class(cat(t,"-1"),m))))))))))))) IF notnull(cr,cs,t) ...
ENDSPEC
```

Fig. 5. Part of the mixed m-rel-oo ADT

Sosy Technology® model compiler, and the XML document of the applied rewriting rules is used by the second phase of RELS.

4 An Example

In this section we present an example of recovering a legacy database. The example takes a part of the database to exemplify the process followed by the RELS tool. The legacy database belongs to a car insurance company and the example is focused on the relation between the insurance policy and the car.

The insurance company manages its insurance policies in the following way:

- An insurance policy insures only one car and at least one.
- A car, which is stored in the database, must be insured by one insurance policy and only one.

The relations between cars an insurances policies are stored in one table in the legacy database. The SQL relational schema which specifies this table is presented in the figure 6.

```
CREATE TABLE car_insurance_policy(
  insurance_num  int        NOT NULL  UNIQUE,
  enrol_date     date       NOT NULL,
  deadline       date       NOT NULL,
  type           char(4)    NOT NULL,
  price          int        NOT NULL,
  licence        int        NOT NULL  UNIQUE,
  make           char(10)   NOT NULL,
  model          char(10)   NOT NULL,
  PRIMARY KEY (insurance_num, licence))
```

Fig. 6. The SQL relational schema of the example

The relational term, who specifies the relational schema in figure 6, is constructed by RELS taking into account the designed relational ADT. The API, which read the relational schema, is used in the process of term construction. As a result of this process, RELS obtains the following term:

```
add_ctr_pk(insurance_num, licence, car_insurance_policy,
add_ctr_unique(insurance_num, car_insurance_policy,
add_ctr_unique(licence, car_insurance_policy,
add_column(model, string, false, car_insurance_policy,
add_column(make, string, false, car_insurance_policy,
add_column(licence, int, false, car_insurance_policy,
add_column(price, int, false, car_insurance_policy,
add_column(type, char, false, car_insurance_policy,
add_column(deadline, date, false, car_insurance_policy,
add_column(enrol_date, date, false, car_insurance_policy,
add_column(insurance_num, int, false, car_insurance_policy,
add_table(car_insurance_policy, create_database()))))))))))))
```

We must take into account that the table represents in the OO model an aggregation between the *Insurance_policy* class and the *Car* class (see figure 7). The *Car* class is the component of the aggregation and the *Insurance_policy*

class is the composed of the aggregation. As we can see, this information will be provided by the user during the re-writing proccess, which transform the relational term into the OO term.

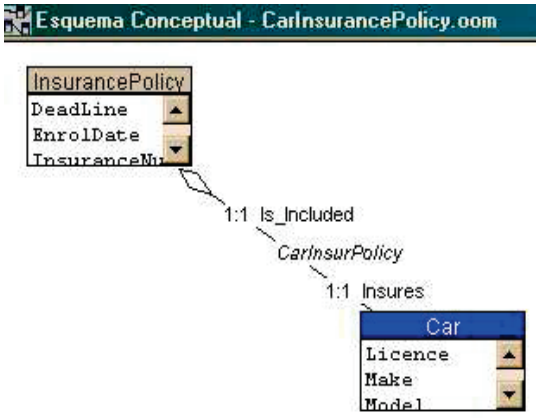


Fig. 7. OO Model

Following is presented the process that applies the rewriting rules in order to translate the relational term into a object oriented term of our example. In this paper, we provides some of the rules which will be presented to the user by means of a graphical user interface (GUI). In this example, we emphasize the changes of the user, it is the moment in which the user change the default rule by another one.

(i) First rewriting rule that it is proposed and applied by RELS.

```
create_database() --> create_schema()
```

(ii) Second rewriting rule that it is proposed by RELS.

```
add_ctr_unique(insurance_num, car_insurance_policy, add_ctr_unique(licence, car_insurance_policy,
add_column(model, string, false, car_insurance_policy, add_column(make, string, false, car_insurance_policy,
add_column(licence, int, false, car_insurance_policy, add_column(price, int, false, car_insurance_policy,
add_column(type, char, false, car_insurance_policy, add_column(deadline, date, false, car_insurance_policy,
add_column(enrol_date, date, false, car_insurance_policy, add_column(insurance_num, int, false,
car_insurance_policy, add_table(car_insurance_policy, m))))))))))) -->
add_ctr_unique(insurance_num, car_insurance_policy, add_ctr_unique(licence, car_insurance_policy,
add_column(model, string, false, car_insurance_policy, add_column(make, string, false, car_insurance_policy,
add_column(licence, int, false, car_insurance_policy, add_column(price, int, false, car_insurance_policy,
add_column(type, char, false, car_insurance_policy, add_column(deadline, date, false, car_insurance_policy,
add_column(enrol_date, date, false, car_insurance_policy, add_column(insurance_num, int, false,
car_insurance_policy, add_class(car_insurance_policy, m)))))))))))
```

(iii) Second rewriting rule that it is applied by RELS. The rule is selected by the user between the rules that can be executed.

```
add_ctr_unique(insurance_num, car_insurance_policy, add_ctr_unique(licence, car_insurance_policy,
add_column(model, string, false, car_insurance_policy, add_column(make, string, false, car_insurance_policy,
add_column(licence, int, false, car_insurance_policy, add_column(price, int, false, car_insurance_policy,
add_column(type, char, false, car_insurance_policy, add_column(deadline, date, false, car_insurance_policy,
add_column(enrol_date, date, false, car_insurance_policy, add_column(insurance_num, int, false,
car_insurance_policy, add_table(car_insurance_policy, m))))))))))))) -->
add_ctr_unique(insurance_num, car_insurance_policy, add_ctr_unique(licence, car_insurance_policy,
add_column(model, string, false, car_insurance_policy, add_column(make, string, false, car_insurance_policy,
add_column(licence, int, false, car_insurance_policy, add_column(price, int, false, car_insurance_policy,
add_column(type, char, false, car_insurance_policy, add_column(deadline, date, false, car_insurance_policy,
add_column(enrol_date, date, false, car_insurance_policy, add_column(insurance_num, int, false,
car_insurance_policy, add_aggregation(AGGR-car_insurance_policy, car_insurance_policy-2,
car_insurance_policy-1, rol_car_insurance_policy-2, rol_car_insurance_policy-1, 1, 1, 1, 1, false, true,
false, false, add_class(car_insurance_policy-2), add_class(car_insurance_policy-1), m))))))))))))
/* IF notnull(insurance_num, licence, car_insurance_policy) */
```

(iv) Third rewriting rule proposed by RELS.

```
add_column(insurance_num, int, false, car_insurance_policy, m) -->
add_vbl_att(insurance_num, int, false, car_insurance_policy-1, m)
/* IF aggregation(AGGR_car_insurance_policy,m) */
```

(v) Third rewriting rule applied by RELS. The rule is selected by the user between the rules that can be executed.

```
add_column(insurance_num, int, false, car_insurance_policy, m) -->
add_ctt_att(insurance_num, int, false, car_insurance_policy-1, m)
/* IF aggregation(AGGR_car_insurance_policy,m) */
```

As a result of this process RELS obtains the object oriented term (see figure 8) and the XML document with the applied rules (see figure 9).

```
add_identif(licence, car_insurance_policy-2,
add_identif(insurance_num, car_insurance_policy-1,
add_unique(licence, car_insurance_policy-2,
add_unique(insurance_num, car_insurance_policy-1,
add_vbl_att(model, string, false, car_insurance_policy-2,
add_vbl_att (make, string, false, car_insurance_policy-2,
add_ctt_att (licence, int, false, car_insurance_policy-2,
add_vbl_att (price, int, false, car_insurance_policy-1,
add_vbl_att (type, char, false, car_insurance_policy-1,
add_vbl_att(deadline, date, false, car_insurance_policy-1,
add_ctt_att(enrol_date, date, false,car_insurance_policy -1,
add_ctt_att(insurance_num, int, false, car_insurance_policy-1,
add add_aggregation(AGR-car_insurance_policy, car_insurance_policy-2,
car_insurance_policy-1, rol_car_insurance_policy-2,
rol_car_insurance_policy-1, 1, 1, 1, 1, false, true, false, false,
add_class(car_insurance_policy-2),
add_class(car_insurance_policy-1),
create_schema())))))))))))))
```

Fig. 8. The object oriented term

For each applied rule, the XML document only includes the subterm of the rule that has been modified. The figure 9 presents the XML document generated for the second rewriting rule (step 3) that has been applied by RELS.

```
<MERToMOO>
   <rule>
      <m-er>
         <table> car_insurance_policy </table>
      </m-er>
      <m-oo>
         <aggregation>
            <name> AGGR-car_insurance_policy </name>
            <composed> car_insurance_policy-1 </composed>
            <component> car_insurance_policy-2 </component>
            <rol_composed> rol-car_insurance_policy-1 </rol_composed>
            <rol_component> car_insurance_policy-1 </rol_component>
            <min_composed> 1 </min_composed>
            <max_composed> 1 </max_composed>
            <min_component> 1 </min_component>
            <max_component> 1 </max_component>
            <dynamic> false </dynamic>
            <inclusive> true </inclusive>
            <id_dependency_composed> false </id_dependency_composed>
            <id_dependency_component> false </id_dependency_component>
         </aggregation>
         <class> car_insurance_policy-1 </class>
         <class> car_insurance_policy-2 </class>
      </m-oo>
   </rule>
</MERToMOO>
```

Fig. 9. The XML document of the third rule applied in the example by RELS

5 Related Work

There are many works that propose database reverse engineering approaches. However, none of them provide a specific industrial tool that offers anything more than just a theoretical result. The common aim of the related works is to obtain an abstract description of the information system with different final purposes (documenting, migrating, etc). Thus, [1] and [6] propose two reverse engineering processes where an entity relationship schema is obtained from a legacy relation schema. This ER schema constitutes the abstract description, but is poorer than an OO model because it cannot be completed by system behaviour. Other works ([3,9,10]) provide OO models from relation schemas. [2] and [8] attempt to reduce the user interaction but require imposing assumptions (3NF relational schemas) and require more information to induce some knowledge (such as instances of legacy database, application source code, etc). We prefer a stronger user interaction because then it is not necessary to impose assumptions or require many inputs and mainly because it is more likely to achieve an accurate model of legacy system.

There are other tools as DB-Main [11] that apply a data reverse engineering process, but their aim is different to RELS because they do not perform a conversion between different models and they do not migrate the data from the legacy database to the new database. These tools recover the conceptual schema from the logic schema in order to obtain traceability between different layers of the database, to create new databases in other DBMS and to reduce

the dependence on the technology.

In [5] the Varlet Database reverse engineering process is explained. The Varlet tool transforms a relational schema into an OO conceptual schema, and migrates the legacy data to the new OO database. Our approach is different: a relational database schema, implementing the persistence layer of an OASIS object society is the migration target. Moreover, in Varlet, the legacy relational schema is enriched with semantic information which are extracted from several sources as the application source code. However, in our approach, this semantic information is given by the user in an interactive way. The last difference between both approaches is the moment in which the transformation rules, for obtaining the desired conceptual schema, are applied. The Varlet approach changes the initial OO conceptual schema produced by the generation process and our approach changes the proposed rules during the generation process.

6 Conclusions and Future Works

This paper presents a solution and a methodology to recover legacy databases of most DBMS using formal-methods based techniques. The presented solution blends formal methods (Abstract Data Types, Term Rewriting System (MAUDE, OBJ)) with current technologies (HASKELL, XML, SQL-Server) following an automatic approach. As a result, it reduces the time invested and the number of people involved in the data reverse engineering and data migration processes. Moreover, this methodology recovers the legacy databases of most legacy DBMS due to the high abstraction level used and the technologies that has been applied. The solution is being implemented in the Department of Information Systems and Computation of the Valencia University of Technology in collaboration with the industrial partner Care-Technologies.

Finally, it is important to note that it is still possible to improve the data reverse engineering process presented. More work is necessary to define the criteria determining the application of the rewriting rules and also in increasing the usability of the user interface. In addition, more work is necessary to take into account the behavior of the legacy system (code) to automatically obtain the information that is not specified in the schema of the database (constraints, derived expressions, etc).

References

[1] Andersson, M., *Extracting an entity relationship schema from a relational database through reverse engineering.* In Proceedings of ER'94, LNCS, 403–419. Springer-Verlag, 1994.

[2] Chiang, R. H. L., Barron, T. M., Storey, V. C., *A framework for the design and evaluation of reverse engineering methods for relational databases*, Data &

Knowledge Engineering **12** (1997), 107–142, Elsevier Science.

[3] Hainaut, J. L., Henrard, J., Roland, D., Englebert, V., Hick, J. M., *Structure Elicitation in Database Reverse Engineering*, WCRE'96 **3** (1996), 131–140.

[4] Haskell, URL: `http://www.haskell.org/`.

[5] Jahnke,J.H., Zundorf,A. *Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment*. Theory and Application of Graph Transformations (TAGT'98), Paderborn, Germany, Technical Report tr-ri-98-201, University of Paderborn,1998.

[6] Johannesson, P., *A Method for transforming Relational Schemas into Conceptual Schemas*, in Proc. Of the 10 th International Conference on Data Engineering, Rusinkiewicz (Ed.), 115–122, Houston, IEEE Press, 1994.

[7] The Maude System, URL: `http://maude.csl.sri.com/`.

[8] Petit, J. M., Toumani, F., Boulicaut, J.F., Kouloumdjian, J., *Towards the Reverse Engineering of Denormalized Relational Databases*, in Proc. of the 12 th International Conference on Data Engineering, New Orleans, Louisiana, USA, IEEE Press, Feb. 1996.

[9] Premerlani, W.J., Blaha, M., *An approach for reverse engineering of relational databases*, Communications of the ACM **37,5** (1994), 42–49.

[10] Ramanathan, S., Hodges, J., *Reverse Engineering Relational Schemas to Object-Oriented Schemas*. Technical Report MSU-960701, Mississipi State University, Mississipi, 1996.

[11] DB-Main, URL:
`http://www.fundp.ac.be/recherche/unites/publications/en/2987.html`.

An algorithm to compare OO-Conceptual Schemas♦

J.Silva, J.A.Carsí, I.Ramos

Department of Information Systems and Computation

Valencia University of Technology

Camino de Vera s/n

E-46071 Valencia – Spain

{josilga | pcarsi | iramos}@dsic.upv.es

Abstract

In this paper, an algorithmic study about how to compare object-oriented conceptual schemas in a data migration context is carried out. An algorithm is presented based on the tree-comparison technique that compares conceptual schemas. The algorithm uses some semantic information in order to optimize efficiency. A template of the equivalences and differences between two schemas is generated in an automatic way. The template shows the results in terms of insertions, updates and deletions. This work is the first phase in a tool which performs the automatic migration of databases starting from the differences detected.

Index Terms: Schema Comparison, Comparison Algorithms, Data Migration, Comparison Strategies, Tree Processing, Evolution.

1. Introduction

Many systems and processes can be represented as tree structures, and sometimes it is necessary to compare these structures to determine the interoperation, the differences, or simply the level of similarity between the systems. This technique is used in software engineering field when it is necessary to compare object-oriented conceptual schemas (CS).

The CS comparison is highly related to the database (DB) migration problem. Ideally, software evolution must be performed at the conceptual schema level, when the applications and databases implementing the information systems could be automatically updated using the evolutioned conceptual schema. Sometimes the database of the original system contains information, and this information has to be migrated to the new system; this fact is known as data migration. The main problem concerning data migration is that the databases involved in the migration process are usually very different and are sometimes incompatible.

Precisely in data migration, it is essential to determine the evolution level of the schema and the associated database to be migrated. It is also indispensable to know the changes in the initial system elements as well as the insertions and deletions that have occurred during the evolution process. Moreover it is very important to determine the information to use for the identification of the origin and the target of the data migrated. For example if Table A exists in the original system DB, and there already exists a Table

♦Work partially funded by the FEDER project "Object-Oriented Software Systems Automatic Generation" (TIC 1FD97-1102).

Silva, J., Carsí, J. A., Ramos, I. An algorithm to compare OO-Conceptual Schemas. Proc. of the International Conference on Software Maintenance (ICSM'2002). 2002.

named A in the target system, how could we determine if the information in the initial Table A must be migrated to the final Table A? It may be that Table A in the origin DB has changed its name to B in the target DB, and this information is vital for the migration.

In this context, it is necessary to design an algorithm wich is able to automate the CS comparison process. In addition, it would be useful for the algorithm to use semantic information about the schemas during the comparison in order to improve accuracy. In other words, the algorithm score is not limited to tree comparison; it has to be able to compare conceptual schemas using the specific conceptual schemas information to resolve possible indeterminism. This work represents the first phase of the migration process in the data migration tool developed [1].

Section 2 provides a review of the CS comparison problem and its profound relation to the general tree-comparison problem. In section 3, we present how the comparison problem is currently dealt with. In this section, we demonstrate that current algorithms are not appropriate for solving the conceptual schema comparison problem in data migration contexts. In section 4, we provide an algorithm that solves the conceptual schema comparison problem and automatically generates all the mappings between the schemas. The conclusions of the article are presented in section 5.

2. The conceptual schema comparison problem

We look for an algorithm to automate the CS comparison in order to migrate the data stored in the DB related with the original CS to the DB automatically generated from the evolutioned CS. The algorithm has a double utility. First, it can measure the evolution level that exists between both schemas. The algorithm measures different features of the schemas such as the cyclomatic complexity of the services, the number of classes, the mean number of attributes of each class, the inheritance level, etc. Second, when the conceptual schemas that are compared are models of the same system, the algorithm can determine the individual evolution of each element in those schemas. In this way, the algorithm will determine the origin and the target of the data and then use them in a migration tool.

Some comercial tools [2, 3] are able to identify the evolution between schemas without a comparison process. These tools keep the trace of the operations that converts the initial schema to the final schema. The problem with this solution is that it is not flexible, because both schemas must be dependent; that is, one of them has to be an evolution of the previous one. However, using a schema comparison process allows the schemas to be independent or to have a different format. Moreover, the operation trace solution is not appropriate in a data migration context, because, in practice, most migrations imply legacy systems. Systems of this kind don't have an associated conceptual schema. Thus, no trace of evolution services exists between the models. In this situation, our approach models the legacy system using an inverse engineering process, and then the comparison process is performed. A specific algorithm for the conceptual schema comparison is then necessary.

The algorithm must be able to automatically generate the differences between two schemas. The differences are expressed in terms of insertions, deletions and updates. In the data migration context, the final objective is to transfer the stored instances between the DBs; then, we only have to consider the elements of the schemas that have a direct repercussion on data persistence. These elements are classes, attributes, aggregation and specialization relationships [2]. With these elements, we represent a conceptual schema like an acyclic and non-directed graph; in other words, like a tree. See Fig. 1.

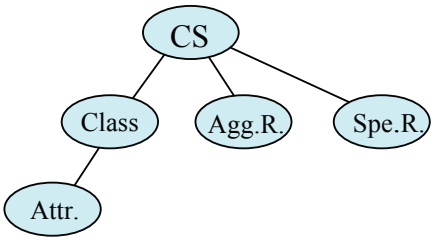


Figure 1: Representation of conceptual schemas as trees.

Figure 1 shows the representation of each element of the conceptual schema as a

node of the tree; the root-node represents the conceptual schema.

In reality, in a data migration context, the obtention of the minimum transformations between two trees is not the main objective. The main objective is to identify, as well as possible, the individual evolution that undergoes each element of the conceptual schemas. The associated cost of this evolution is not very important. With this consideration in mind, the algorithmic approach to be used is radically different in both cases. First, the algorithm has to identify each one of the elements of both trees, and then it has to establish a bijective function between each pair of elements identified as mapped elements (two elements of two distinct trees are mapped elements, if one element is derived from the other). In this paper, the bijective function is called in the paper 'mapping'. The formal definition of mapping is presented below:

Tree Mapping: It intuitively denotes how an operation set transforms a tree T into another tree T', without considering an order in the application of the operations. Formally, a tree mapping is 3-ple (M, T, T'); where T is the initial tree, T' the final tree, and M a set of pairs of integers (i, j) that satisfy:

1) $1 \le i \le |T|$, $1 \le j \le |T'|$;

2) For each (i_1,j_1) and (i_2,j_2) in M:

a) $i_1 = i_2$ if and only if $j_1 = j_2$

b) $i_1 < i_2$ if and only if $j_1 < j_2$

c) $T[i_1]$ is an ancestor of $T[i_2]$ if and only if $T'[j_1]$ is an ancestor of $T'[j_2]$.

Element mapping: Each pair $(i,j) \in M$ of one tree mapping is called an element mapping, or simply, a mapping.

We can now define the main objectives of our work:

1. To obtain a conceptual schema comparison algorithm that correctly identifies the evolution affecting each one of the elements of the initial conceptual schema.
2. The algorithm must establish a mapping for all the elements of the conceptual schemas.
3. The algorithm should not be limited to tree comparison. It has to be a specific CS comparison algorithm, and must use semantic information for the comparison.
4. The CSs that are compared could be dependent or independent.

Following, we will present an algorithmic analysis that solves the problem taking into account each of the established objectives. First, the minimum path between trees algorithm is presented, and then, the specific CS element identification problem is presented.

3. State of the art

3.1. The 'Tree to Tree' problem

The problem of obtaining the differences between two schemas is currently solved by different comercial tools using the trace of the evolution operations; i.e. VDIFF of Rational Rose [3]. The main problem with these tools is that the CSs have to be dependent. Moreover, the need for an evolution trace requires the schemas to be in the same format, or to pertain to the same application. These restrictions do not exist when using a comparison process.

Many structures and processes can be modeled as labeled trees, and sometimes it is very useful to know how to transform one tree into another tree using the minimum number of changes. A particular case of this problem occurs when the trees are of depth two. This is the case when we compare two strings, where each character is represented by one node with a fixed position in the leaf nodes. This problem was solved by several authors. Sankoff [4], and Wagner and Fisher [5] have presented an algorithm that is capable of computing a sequence of edition operations with minimum cost. This algorithm has an associated computational cost O(n*m), where m and n are the number of leaves of the trees. Later on, Wong and Chandra [6]; and Aho, Hirschberg and Ullman [7] demonstrated that Sankoff's algorithm is optimal for many computation models. Another algorithm that solves the problem with optimum cost was presented in 1977 by Selkow [8].

A generalization of the problem is to consider trees of any depth. In this case, the complexity of the problem increases by a quadratic factor. The following algorithm wich

was developed by Kuo-Chung Tai [9] solves the problem.

Given two trees T and T', the algorithm by Kuo-Chung Tai, computes their minimal distance with $O(V * V' * L^2 * L'^2)$ cost, where V and V' are the # nodes of T and T', respectively, and L, L' are the maximal depth levels.

3.2. Compound type changes

A model is proposed in [10] for the identification of type changes encountered in schema evolution. In this work, a set of algorithms for the schema comparison and for the compound type comparison are introduced. The comparison algorithm proceeds through three stages. First, in the *name comparison* stage, old and new types that have the same names in both versions are compared. In the second stage, called *use site comparison*, types using types that have been successfully compared are compared. In the final stage, called *exhaustive comparison*, each old type that does not already have a pair is compared to each new type. This algorithm has an associated cost of O(n*m).

The first inconvenience of this work, is that the structure of the algorithm is very rigid; the types are always compared using the same criterium and always in the same order. An error in the first stage of the algorithm, cannot be restored in the following stages, and can lead to errors in the following stages as well.

In [10], a set of compound type changes are presented to be solved by any comparison algorithm. These changes are the following:

-Inline: Replace a type reference by its type definition.

-Encapsulate: Create a new type by encapsulating parts of one or more types.

-Merge: Replace two or more type definitions with a new type that merges the old type definitions.

-Move: Move part of a type definition from one type to another existing type.

-Reverse Link: Reverse the connection between two types.

-Link Addition: Add a link between two existing types.

The following section presents a study to obtain a CS comparison algorithm wich is able to solve the problem by finding all the related compound changes and improving the [10] algorithm.

4. Algorithmic analysis based on semantic information

Selkow's algorithm computes the minimum path between two trees of depth two; and Kuo-Chung Tai's algorithm computes it for trees of any depth. Thus, we have an inferior limit of the cost of a CSs comparison algorithm. Kuo-Chung Tai's algorithm can be used for comparing CS; however, this algorithm wasn't defined to compare CS but rather to compare simple trees, and it wastes a lot of information of the schemas that can be used to improve the comparison result.

A lot of semantic information can in the context of CSs comparison for data migration be used by the comparison algorithms. Similarly, the information that is generated during the comparison of two elements can be reused in future comparisons of the other elements of the conceptual schemas. In this section, we are going to analyze exactly what information should be used and how it can be utilized to design an algorithm wich is specific to the CS comparison.

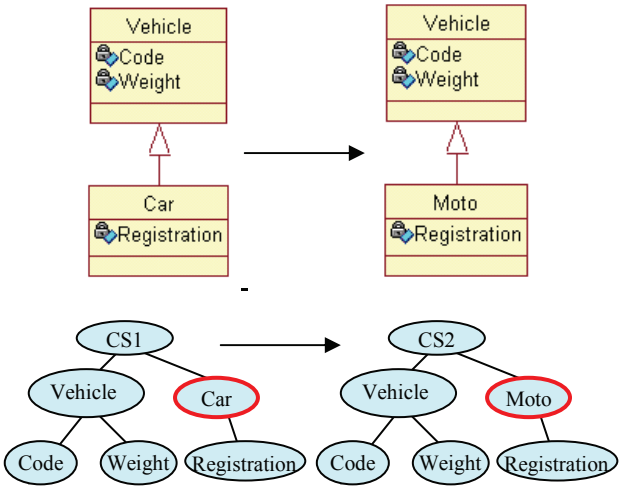


Figure 2: 'Tree to Tree' conceptual schema comparison.

The distance concept between two trees can be compared to the measure of

similarity between them. In an evolution context, this distance can measure the evolution level that exists between two schemas that are represented as tree structures. Nevertheless, the computation of the minimum path between the trees is not the best solution for identifying the evolution of two conceptual schemas; Figure 2 shows an example of this problem.

If the algorithm searches the minimum path between the trees, it will compute the substitution of 'Car' by 'Moto'. However, the class 'Car' has been deleted and the class 'Moto' has in reality been created so, in this case, a mapping between 'Car' and 'Moto' wouldn't exist. The search of a minimum way is not a good solution for the CSs comparison, because the CSs evolution follows random evolution patterns, that do not have to coincide with minimum paths. Optimization algorithms find the fastest form of converting one tree into another tree. In the following sections, the algorithms attempt to find alternative methods for comparing conceptual schemas; they attempt to identify the real evolution of the class instances. Also, the algorithms presented above do not take advantage of the semantic information of the models.

The 'Tree to Tree' approximation can be used when the conceptual schemas do not have instances to be migrated; but, in a data migration and evolution context, the conceptual schemas have a set of instances associated to them that have to be transferred to another database. In this context, 'Tree to Tree' approximation is not appropriate because it does not take into account the data evolution features.

To solve the problem, it is necessary to find alternative comparison methods; these methods have to take into account the data migration and have to establish the goal of identifying the analyst's proposal with each of the elements of the schemas changes when the evolution is performed. Accordingly, we need to find an algorithm that compares conceptual schemas taking advantage of their tree structure and their CS exclusive properties as well.

One of the initial premises was that the two trees must be labeled trees. The question is what information the labels of the nodes must contain? In a CS there is a lot of information available to compare CSs (identifiers, names, relationships, number of atributes of the classes, etc.) and this information can strongly determine the success of the comparison. Several criteria for CS comparison are analized in detail in [11]. We call the information used to compare a set of elements **comparison criteria**.

Limiting the comparison of two CS to only trees does not take advantage of a lot of semantic information. A CS comparison algorithm must make use of the tree structure, but also has to use the CS structure properties. This idea can provide a lot of rich information during the comparison. The algorithms of the previous section waste this semantic information, and consequently, process all nodes in the same way. Comparing all elements with all? In a CS-Tree, there exist four kinds of branches:

- Classes
- Attributes (subbranch of a class)
- Aggregation relationships
- Specialization relationships

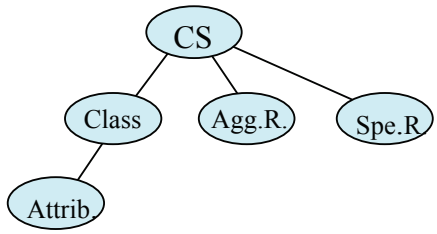
With these four kinds of branches of a CS, the comparisons done by an algorithm are limited to the comparisons between elements of the same type. It does not make sense to compare an attribute to a class. Therefore, the logical comparisons are classes with classes, attributes with attributes, aggregation relationships with aggregation relationships and, finally, specialization relationships with specialization relationships. Because of this, the general tree comparison algorithms can be strongly improved. To do this, we are going to utilize a fragmentation technique with the goal of dividing the trees into comparable subtrees.

4.1 Improving the 'Tree to Tree' algorithms using fragmentation techniques.

Given any two labeled trees, where V and V' are its numbers of nodes, and where L and L' are its respective depths; the Kuo-Chung Tai's algorithm computes the minimum way in terms of Additions, Updates and Deletions with a cost:

$$O(V \cdot V' \cdot L^2 \cdot L'^2)$$

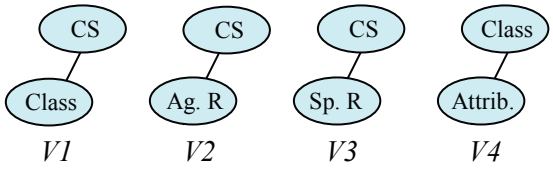
If we represent a conceptual schema as a tree:



The cost of the application of Kuo's algorithm is:

Cost1 = V·V'·4·4 = 16·V·V' ∈ O(V·V')

This cost can be improved if we reduce the number of comparisons by dividing the tree into comparable subtrees. Thereby, if we fragment the tree, we can reduce the cost of both variables L and L' to one:



The cost would be:

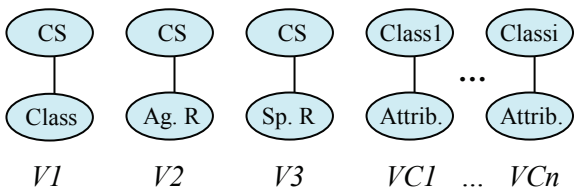
Cost2 = (V1·V'1) + (V2·V'2) + (V3·V'3) + (V4·V'4) ∈ O(V·V')

As $V = V1 + V2 + V3 + V4$ and $V' = V'1 + V'2 + V'3 + V'4$ then:

Cost1 = V1·V'1 + V1·V'2 + V1·V'3 + V1·V'4 + V2·V'1 + V2·V'2 + V2·V'3 + V2·V'4 + V3·V'1 + V3·V'2 + V3·V'3 + V3·V'4 + V4·V'1 + V4·V'2 + V4·V'3 + V4·V'4 ≥ Cost2

As any CS has V1>0 and V4>0, then cost1 is always greater than cost2: cost1 > cost2.

You can decrease this cost if you continue fragmentizing the subtrees (as we will explain later, this is useful only in case the comparison follows a logical order):



The cost would be:

Cost3 = (V1·V'1) + (V2·V'2) + (VC1·VC'1) + … + (VCn·VC'n) ∈ O(V·V')

As $V4 = VC1 + ... + VCn$

then

Cost2 = (V1·V'1) + (V2·V'2) + (V3·V'3) + (VC1·VC'1) +...+ (VC1·VC'n) +...+ (VCn·VC'1) +...+ (VCn·VC'n) ∈ O(V·V') > Coste3

The following always holds: cost1 > cost2 > cost3.

Another kind of semantic information that must be taken into consideration is the comparison order. It is not logical to compare attributes if its classes[1] have not already been compared. The CS comparison must follow a logical order, not a preestablished one but one that depends on the comparison criteria. For example, comparing two schemas using a criterium based on the OID, the logical order would be:

- Analysis of classes (part 1).
- Analysis of attributes.
- Analysis of classes (part 2).
- Analysis of relationships.

The first step (the analysis of classes) is to identify what classes of the initial conceptual schema have been deleted and what classes of the final conceptual schema have been added; the rest of the classes are considered to be invariant invariant[2] classes. The second step is the analysis of attributes wich is done for each of the invariant classes (all attributes of added classes are new; and all attributes of deleted classes have also been deleted). When the analysis of attributes ends, it is possible to determine which classes have been updated, and which are really invariant. Finally, when all classes are identified, the analysis of the relationships between them is done.

[1] Parent nodes in the tree.
[2] They might have been modified, but are mapping classes.

When semantic information is applied, the algorithm does not have to compare trees of depth two; it has to compare many trees of depth one. In this case, Selkow's algorithm might solve the problem perfectly by apply it to each pair of trees; but this is false. Selkow's algorithm considers the order of the nodes in the tree when it makes the comparison, and this is irrelevant for the CS comparison. Thus, we can reduce still more the restrictions of the problem even.

4.2 The compound class changes.

According to the compound type changes related in [9], the algorithm must incorporate mechanisms for the treatment of compound type changes. The analysis of each compound type change applied to the object oriented CSs follows:

-Inline: Replace a type reference by its type definition.

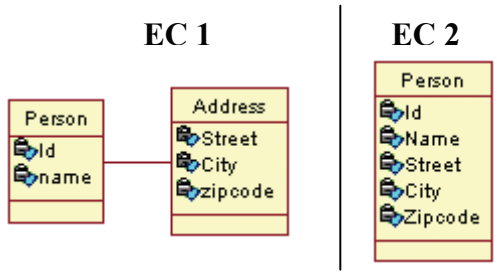


Figure 3: Inline example.

Inline occurs when all the attributes of two associated classes are included in one class or in both classes. An algorithm that solves inline has to establish mappings between attributes of two diferent but related classes. Then, it cannot limit the search of attributes of a class to attributes of its mapping class. In other words, with inline, mappings between attributes of unmapped classes, but related by association, aggregation or specialization in the original CS, can exist.

-Encapsulate: Create a new type by encapsulating parts of one or more types.

In the object-oriented approach, the encapsulate type change is a generalization of inline. Encapsulate allows the classes to be formed by attributes of one or more classes, and these classes can be related to it or not. If one algorithm solves the Encapsulate type changes, then it also solves Inline type changes. When an algorithm solves Encapsulate, the fragmentation presented in section 4.1 cannot be applied. In this case, attributes do not establish a new subbranch of the CS tree, because the algorithm must search a mapping of one attribute in all attributes of the conceptual schema.

-Merge: Replace two or more type definitions with a new type that merges the old type definitions. The main difference between Inline and Merge is that in Merge at least one of the attributes must have the same value in both classes of the CS. When Merge occurs, in reality, two objects exist in the system that are representing two different features of one object of the real model that is being represented. It implies that to determine when two objects of the initial CS have to be transferred to one of the final CS, it is necesary to verify values of the instances's attributes. To do this, the algorithm should use population comparison criteria; that is, criteria that use information from the instances stored in the databases [11].

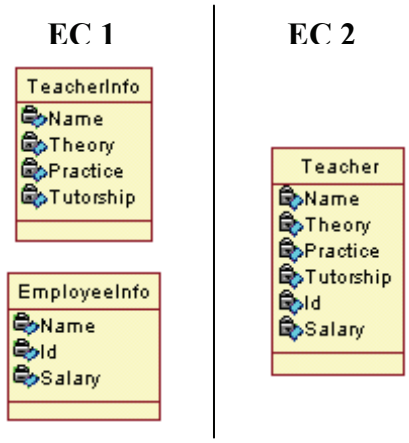


Figure 4: Merge example.

-Reverse Link: Reverse the connection between two types.

In object-oriented conceptual schemas, this kind of evolution is irrelevant, because the links of the types in the databases's schema are automatically generated from the CS.

-Link Addition: Add a link between two existing types.

This compound type change, like Reverse Link, does not influence the algorithm's design.

-Move: Move part of a type definition from one type to another existing type. This compound change is a subcase of Inline where only a few attributes change class. If Inline is solved, then Move is also solved.

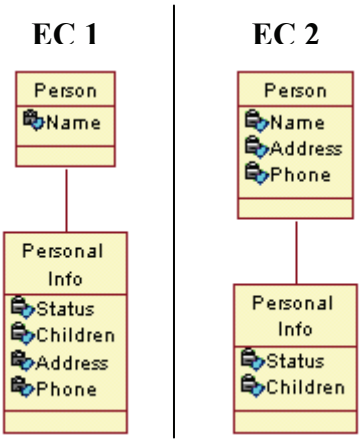


Figure 5: Move example.

-Duplicate: Duplicate part of a type definition in another type definition.

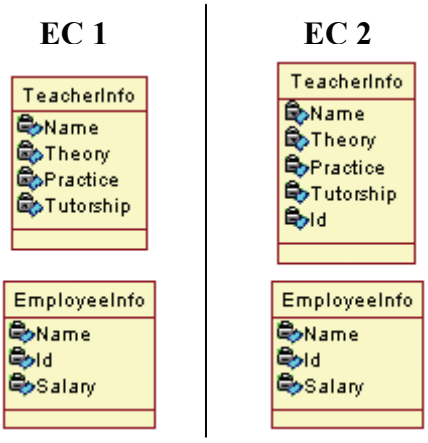


Figure 6: Duplicate example.

Like Merge, Duplicate requires population comparison criteria to be solved. In addition, this compound type change implies that several mappings can exist for just one attribute of the original CS. That is, the information of one object in the original CS, must be transferred to several objects of the final CS.

Now, we can present the algorithm that solves the comparison of conceptual schemas:

We start with the following information:

- **A, B:** Are lists with the elements of the tree T and T', respectively.
- **Criterium**(x, y)**:** Returns TRUE if x and y are the same in accordance with any preestablished comparison criterium.
- **Pair**(x, y)**:** Introduce x and y in the pair list, and take them out of the respective node lists.
- **Deletion**(x): Insert the element x in the deleted elements list.
- **Addition**(x): Insert the element x in the added elements list.
- **Sort**(Ai): As Ai is always a two depth subtree, all elements are at the same depth level. This function sorts the A list with a metric according with the comparison criterium used.

ALGORITHM:

Procedure *CompareTrees(A, B);*

(We assume that A and B have subtrees $A_1,...,A_n$; $B_1,...,B_m$ where the first represents the classes subtree, the second represents the aggregations subtree, the third represents the specializations subtree and the following represents all the attributes subtrees)

Procedure *CompareSubtrees(i, j)*

Begin

Sort(A_i); P_a = ***first***(A_i);

Sort(B_j); P_b =***first***(B_j);

/ P_a and P_b are pointers to the first element of A_i and B_j respectively */*

While *((a<Length*(A_i)*)* ***and*** *(b<Length*(B_j)*)))*

If Criterium(P_a, P_b) *= TRUE* ***then***

Pair(P_a, P_b); P_a = ***next***(A_i); P_b = ***next***(B_j);

Else If $P_a < P_b$ ***then***

Deletion(P_a); P_a = ***next***(A_i);

Else If $P_a > P_b$ ***then***

Addition(P_b); P_b = ***next***(B_j);

End if;

End While; */* Complete the next part of the list with insertions or deletions */*

End */* CompareSubtrees */*

Begin

For *i = 1* ***to*** *3* ***do***

CompareSubtrees(i, i);

End For*;*

For *each mapping of classes established in the first iteration where c_i, c_j are the parent nodes of the subtrees A_i and B_j respectively* ***do***

CompareSubtrees(i, j);

End For*;*

/ Look for compound type changes */*

For *each attribute b of B in a mapping class c_b* ***do***

/ Look for Inline and move */*

For *each attribute a of A in a class R related with the c_b mapping class where maximum cardinality of the R relationship is 1* ***do***

If Criterium*(a, b) = TRUE* ***then***

Pair*(a, b);*

End if;

End For*;*

/ Look for encapsulate */*

For *each attribute a of A in a class non-related with the c_b mapping class* ***do***

If Criterium*(a, b) = TRUE* ***then***

Pair*(a, b);*

End if;

End For*;*

End For*;*

End */* CompareTrees */.*

When the algorithm ends, we have three lists. The first one contains all elements that have been deleted from the initial CS; the second list contains all elements that have been added to the final CS; and the third list contains all elements that have been changed during the evolution process. This algorithm is totally dependent on the selected comparison criterium; this is right because the selected criterium determines when two elements are considered the same; this fact is very important because, all criteria are not equivalent, and no criterium exist that is the best in all cases.

Assuming a constant cost for the function 'criterium', this algorithm has an associated cost O(n*log(n))[3] for the first part, because it sorts the lists before its computation. For the search of the compound type changes, the cost is O(s*s), where s is the number of attributes in the CS. The comparison criterium can present a high complexity if it uses the information that the algorithm obtains during the comparison process.

The proposed algorithm is the core of a tool developed at DSIC in the Valencia University of Technology. This tool provides a CASE environment supporting evolution of schemas and the posterior transfer of data between their associated databases. This algorithm automatically determines the individual evolution of each element of the schemas and proposes a default migration plan for its instances.

5. Summary and concluding remarks

The CS comparison problem is a richer process than the simple tree-comparison problem. The optimized algorithms for tree-comparison can be used for comparing conceptual schemas, but they can also be improved using semantic information:

- In a conceptual schema there exist four kinds of comparable elements. Only elements of the same group must be compared.
- Each element of a CS is unique, and is perfectly identified: The first successful comparison of the algorithm can end the search for this element.
- Unlike the simple tree-comparison algorithm, CS comparison algorithms can make use of semantic information that is generated during the comparison process.
- The comparison algorithm must be driven using a logical order of comparison.
- The comparison criteria will strongly determine the final result of the comparison.

The main objective of a comparison is not the search of minimum paths; the main objective is the identification of the individual evolution of each element. Finally, the CS comparison problem is not an optimization problem.

[3] Cost would be n*log(n) + m*log(m) + n + m : the first two components are derived from the sort of the lists (using for example 'Mergesort' or 'Quiksort'); the third and fourth components of the cost represent the treatment of the lists.

References

[1] **Carsí J.A.**, **Ramos I.**, and **Molina J.C.**, "*Automatic generation of data migration plans from OO conceptual schemas"*, IDEAS2001,IV Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software, San José - Costa Rica, ISBN: 9968-32-000, April 2001. (in Spanish)

[2] **Carsí J.A.,** *"OASIS as conceptual framework for software evolution"*, PhD Thesis, Faculty of Computer Science, Valencia University of Technology, October 1999. (in Spanish)

[3] **Rational**, Rational Rose Software Development Company Home Page: http://www.rational.com/.

[4] **Sankoff D.**, "*Matching sequences under deletion/insertion constraints"*, Proc. Nat. Acad. Sci., USA 69, pp. 4—6, 1972.

[5] **Wagner R.A.** and **Fisher M.J.**, *"The string to string correction problem"*, *Journal of the ACM*, 21(1):168-173, January 1974.

[6] **Wong C.K.** and **Chandra A.K.**, *"Bounds for the string editing problem"*, Journal of the ACM, 23(1):13-16, January 1976.

[7] **Aho A.V., Hirschberg D.S.** and **Ullman J.D.**, *"Bounds on the complexity of the longest common subsequence problem"*, Journal of the ACM, 23(1):1-12, January 1976.

[8] **Selkow S.,** *"The tree-to-tree editing problem"*, Department of Computer Science, University of Tennessee, Knoxville, *Information Processing Letters*, 6(6):184-186, December 1977.

[9] **Kuo-Chung Tai,** *"The tree-to-tree correction problem"*, Department of Computer Science, North Carolina State University, Raleigh, Journal of the ACM, vol. 26 no 3, pp. 422-433, 1979.

[10] **Staudt B.,** *"A model for compound type changes encountered in schema evolution"*, University of Massachusetts, Amherst, *ACM Transactions on Database Systems, Vol. 25, No. 1, March 2000, Pages 83–127.*

[11] **Silva J.**, **Carsí J.A.** and **Ramos I.**, *"Classification of comparison criteria for object-oriented conceptual schemas"*, Technical Reports DSIC-II/04/02, Valencia University of Technology, February 2002. (in Spanish)

[12] **Assenova P., Johannesson P.**, *"Improving Quality in Conceptual Modeling by the Use of Schema Transformations"*, in: B. Thalheim (Ed.): Conceptual Modeling: proceedings ER '96, Springer, Berlin et al., S. 277-291, 1996.

[13] **Grau A.,** *"Computer-Aided Validation of formal conceptual models"*, PhD. Institute For Software, Information Systems Group, Technical University of Braunschweig, 1998.

[14] **Herden O.**, *"Measuring Quality of Database Schemas by Reviewing - Concept, Criteria and Tool"*, Oldenburg Research and Development Institute for Computer Science Tools and Systems, Escherweg 2, 26121 Oldenburg, Germany.

[15] **Letelier P., Ramos I., Sánchez P.** and **Pastor O.,** *"OASIS 3.0: A formal approach for object-oriented conceptual modelling"*,SPUPV-98.4011, Department of Information Systems and Computation, Valencia University of Technology, 1998.

[16] **Pérez J., Anaya V., Silva J., Carsí J.A** and **Ramos I.,** *"Automatic generation of data migration plans between object-oriented databases",* I Workshop on Distributed Objects Languages Methods & Environments, DOLMEN2001, Sevilla. (in Spanish)

[17] **Pastor O., Insfrán E., Pelechano V., Romero J.,** and **Merseguer J.**, "*OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods",* Conference on Advanced Information Systems Engineering (CAiSE'97). LNCS (1250), pag. 145-159. Springer-Verlag 1997. ISBN: 3-540-63107-0. Barcelona, Juny 1997.

On the implication of application's requirements changes in the persistence layer: an automatic approach

Jennifer Pérez, José Ángel Carsí, Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n
E-46071 Valencia – Spain
{jeperez | pcarsi | iramos}@dsic.upv.es

Abstract. This paper presents a solution to the data evolution problem of information systems. This solution follows an automatic approach that reduces the number of people and the time invested in the software maintenance process. The information captured by the UML-like OASIS OO conceptual schemas, representing system evolution, is used in order to generate automatically a data migration plan. This paper is focused in the process of automatic generation of the structure and the content of this migration plan using patterns. A real migration example is used to explain the designed patterns. The data migration plan execution evolves and migrates the data from the initial conceptual schema database to the evolved conceptual schema database. The solution has been implemented in the migration tool ADAM (Automatic DAta Migration)[1].

Key words: Data migration plan, migration order, correspondences, migration patterns and migration expressions patterns

1. Introduction

Information systems have a dynamic nature and they frequently undergo changes. These changes are due to volatile business rules or to external factors. For this reason, system management tools must be able to deal with them in an efficient and complete way.

Existing CASE tools are able to generate applications following the paradigm of automatic prototyping. These CASE tools are called model compilers. They generate automatically the application code and the database schema from the conceptual schema of an information system. The automatic generation can be complete as Oblog Case ([Ser94]), Sosy Modeler® (OO-Method/CASE ([Pas97]), or partial, as Rational Rose ([Rat02]), System Architect ([Sys02]), Together ([Tog02]) and others. Moreover, they can evolve the application modifying the conceptual schema and regenerating the code and the database schema from the modified conceptual schema (see figure 1).

These model compilers do not consider the data stored in the database during the evolution process. When an information system undergoes an evolution, its conceptual schema is updated and a new schema results. From the new schema, a model compiler generates a new code and a new empty database. The structure and the properties of the new database could be different from the old

[1] ADAM is a CONSOFT technology product.

Pérez J., Carsí J.A, Ramos I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR'2002) at the ICSM'02. 2002.

database. Therefore, the data remains compliant to the old database schema and the designer of the application must preserve the data of his company in spite of the evolution of the information system conceptual model. For this reason, the data remaining in the old database must be correctly migrated to the new one in order to satisfy the properties of the new database.

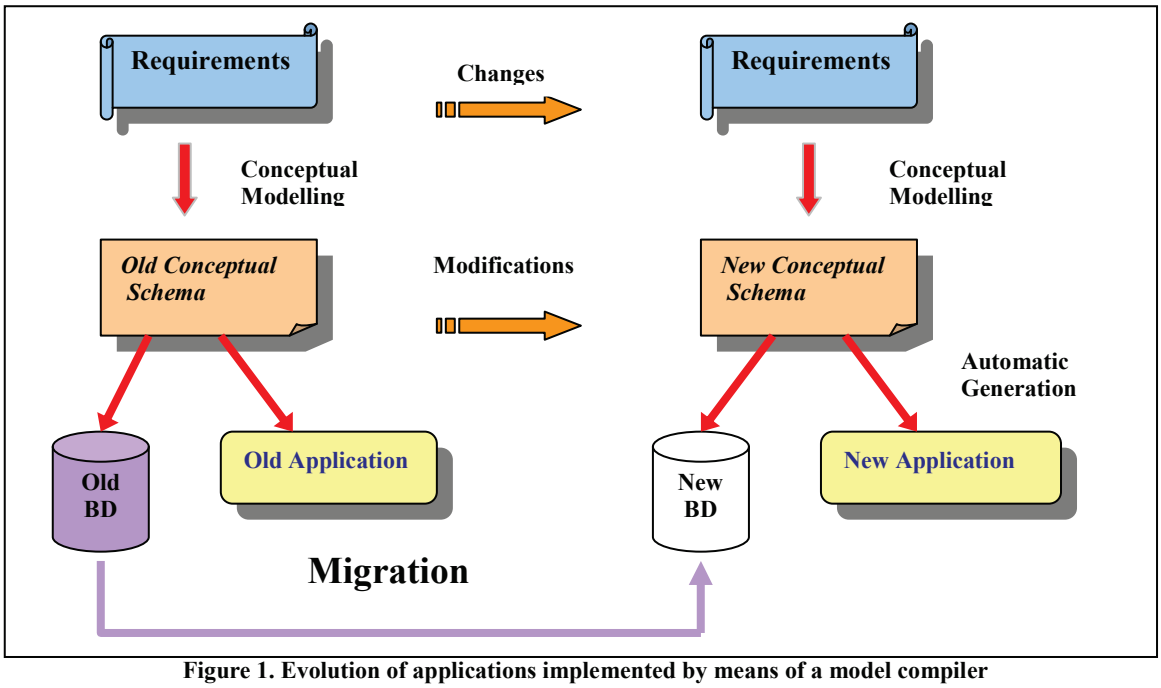


Figure 1. Evolution of applications implemented by means of a model compiler

The migration task is necessary and normally is handmade. This task increases considerably the maintenance cost of a software product. For this reason, our work is focused on data evolution in order to improve this database maintenance process. In this paper, we present a solution for the data evolution problem following an automatic approach and a migration tool (ADAM) has been implemented. Nowadays, ADAM is used together with Sosy Modeler®, a model compiler generating automatically three layer architecture (presentation, business logic and persistence) applications. A formal language OASIS ([Pas95]) based in dynamic logic is used as the specification language of UML-like models allowing its automatic compilation.

ADAM is a migration tool developed in the Department of Information Systems and Computation of the Valencia University of Technology in collaboration with the industrial partner CONSOFT. The data migration process followed by ADAM transfers and updates information system data from the old database to the new one. It reduces the time invested and the number of people involved in data evolution process up to an 80%. This maintenance improvement is due to the automatic tasks that constitute the three steps of the migration process. Despite the fact that these steps are performed automatically, its results can be modified by the analyst in a free way. In this last case, the process will be semi-automatic. The three steps involved are the following:

1.- Old and new conceptual schemas are compared using an algorithm and several comparison criteria ([Sil02]) to automatically obtain the matching (correspondences) between them.

2.- Changes on the data are obtained from these matching and are automatically translated to migration expressions using an object oriented migration language. These expressions constitute a OO data migration plan ([Per01-b]). The migration plan is generated as XML document, used by the third step to extract easily the migration information.

3.- The object oriented data migration plan is translated to a relational one. This translation process generates *Data Transformation Service* packets ([Cha00]) whose execution migrates automatically data from the old database to the new database ([Ana01]). These packets are executed by *SQL Server* ([SQL]).

The structure of the paper is the following: first, we show a real migration example that we use to explain the automatic generation of a data migration plan. Section 3 presents the generation process, the information that it needs, the patterns used to generate a data migration plan and the applied methodology. In section 4, a brief summary of related work is given. Finally, the future work and conclusions are presented in section 5.

2. A Real Example of Migration

In the paper, the well known database example of bills-customers-products presented in Figures 2 and 3 is used. We take a part of the object model of the complete example to understand the reasons why the *Product* class of the example has been evolved. The migration plan generation is exemplified in the paper using the *Product* class evolution. The OASIS schemas capture the structure of the billing management of an information system. The new conceptual schema (figure 3) improves the previous one (figure 2). This improvement consists in:

- Partition of the initial customers. As a result, the *Customer*, the *Person* and the *Company* classes of the new conceptual schema will extract data from the *Customer* class of the old conceptual schema.
- A finer specification of the *Bill* class. As a result, the *Bill* and the *Bill_row* classes of the new conceptual schema will extract their data from the *Bill* class of the old conceptual schema.
- Currency change. As a result, all the prices of the products must be converted into euros.
- Some model errors corrections.

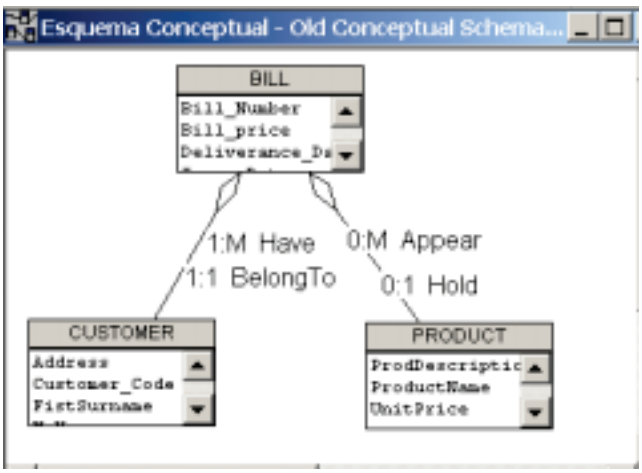


Figure 2. Old conceptual Schema

As a technical detail, it is worth noting that these schemas are modeled using OASIS. However, if we model them using UML, their appearance would be different. This is due to the fact that in OASIS the association is considered as a kind of aggregation and thus, the referential aggregation and the inclusive aggregation are represented by a lozenge. This is the reason why UML associations are modeled like OASIS referential aggregations. For example: The relationship between the classes *Bill_Row* and *Product* is an association represented by an OASIS referential aggregation.

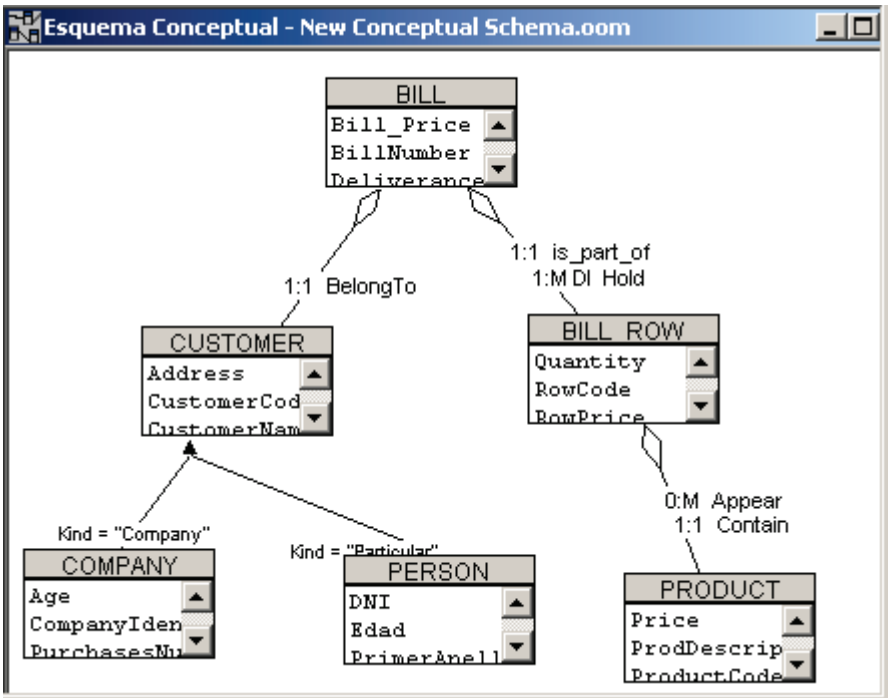


Figure 3. Old conceptual Schema

3. Automatic Generation of a Data Migration Plan

A data migration plan is a set of data changes that are specified using a migration language (ADML). Its execution allows us to transfer data from an old database to a new database. A data migration plan must include all the necessary changes in order to perform a correct migration and in the right order. It is structured as the composition of the following different elements: migration expressions, changes and modules. The highest abstraction level of this composition is constituted by migration modules whereas migration expressions are the lowest level. The definition of these elements is as follows:

- *Migration expressions:* Migration expressions are those expressions that can be specified in a data migration plan. Each type of migration expression has different semantic and follows different syntactic restrictions. Examples of migration expressions are: Data Source, Transformation Function, Filters, etc

- *Changes:* A change is the set of migration expressions that specify the updates undergone and the filters applied on the old database's instances.

- *Migration modules:* A migration module has a transactional behavior when it is executed by the ADAM tool. The composition of modules forms a data migration plan. Finally, it is important to note that there is one migration module for each class, aggregation and specialization of the new conceptual schema.

Each one of the changes that can be undergone in the old database are specified using a declarative language (a language which express properties of WHAT to do and not HOW to do it). This migration language (ADML) follows the object-oriented metaphor. The main advantage of ADML is the independence from any DBMS due to its high abstraction level.

This language makes use of path expressions notation to specify changes that have been undergone in a conceptual schema element[2]. Conceptual schema elements are a set of origins of the data, transformations of the data and filters that will be applied on the data at the execution moment.

3.1. Inputs

The generation process of a migration plan needs several data sources to generate automatically the structure and the contents of the plan. The inputs of this process are the following:

- *Matchings between conceptual schemas:* A matching defines the old element from which must be obtained the data to provide to a determinate new element. This information is available because the previous step of ADAM finds the matches between elements of both conceptual schemas. This step executes an algorithm that uses different comparison criteria in order to find the most suitable matches ([Sil01]). As a result, this process produces the correspondences between the elements of the conceptual schemas that have been compared. For example: The *Product* class of the new conceptual schema obtains its data from the *Product* class of the old conceptual schema (matches at class level) and the matches between their attributes are the following:

OLD CONCEPTUAL SCHEMA	NEW CONCEPTUAL SCHEMA
Cod_Product	ProductCode
ProductName	ProductName
ProdDescription	ProdDescription
UnitPrice	Price

Table.1. Matches between the attributes of the old Product class and the new Product class

- *Properties of the conceptual schema elements:* The properties of the conceptual schema elements are necessary in order to know the changes between the elements of a match instance and to generate the implied transformations. These transformations must be applied on the data of the old element in order to be compliant with the new element. This information is afforded by the conceptual schemas where only the properties of each element are defined. In our example we are going to focus on the following correspondence:

[2] The conceptual schema elements that have been considered for the generation of a data migration plan are: classes, attributes, aggregations, associations and specializations.

- The *Price* attribute of the *Product* class of the new conceptual schema obtains its data from the *UnitPrice* attribute of the *Product* class of the old conceptual schema. The properties of both attributes are included in the specification of the classes (see figures 4 and 5).

Atributos

Nombre	Tipo atributo	Tipo dato	Id	Tamaño	Valor defecto	Pedir al crear	Nulos
Cod_Product	Constante	Autonumerico	0			No	No
ProductName	Variable	String		15		Sí	Sí
ProdDescription	Variable	String		100		Sí	Sí
UnitPrice	Variable	Nat				Sí	Sí

Figure 4. Attributes of the Product class of the old conceptual schema

Atributos

Nombre	Tipo atributo	Tipo dato	Id	Tamaño	Valor defecto	Pedir al crear	Nulos
ProductName	Variable	String		15		Sí	Sí
ProdDescription	Variable	String		80		Sí	Sí
Price	Variable	Real				Sí	No
ProductCode	Constante	Autonumerico	0			No	No

Figure 5. Attributes of the Product class of the new conceptual schema

We can see at the figures 4 and 5 that the prices of the products were in pesetas, and now they must be in Euros. As a result, the data type of the *Price* attribute was *natural* in the old schema and is *real* in the new schema and also the *Price* value must be converted to the equivalent in euros. In addition, the attribute *UnitPrice* allowed null values and now the attribute *Price* does not allow null values.

- *Migration order:* Migration order is the order in which the data should be migrated. This preserves the database not to be in inconsistent states during the migration process. Moreover, this order affords the migration modules composition order and also the migration order between non-related modules. The computation of this order is made by means of an algorithm. The migration order algorithm analyses the structure of the new conceptual schema in order to obtain the relationships between elements. These relationships imply dependences determining the order to be followed in the data migration process ([Aba01]). The migration order obtained by the algorithm for the new conceptual schema of our example is the following: Customer, Bill, Product, Company, Person, Bill_Row.

3.2. Patterns

The automatic generation of a data migration plan implies generating its structure and its contents. First of all, ADAM generates the structure creating an empty migration module for each element of the new conceptual schema and includes the modules in the data migration plan following the computed migration order. Next, the module's content is automatically generated giving the migration expressions of each migration module and including them into their modules. Finally, a complete migration plan results. This automatic and complete generation of the migration plan is performed using two types of patterns: migration patterns and migration expressions patterns. We have specified them using the patterns design criteria proposed by [Ale79] and [Gam94]. They are available

in a patterns catalog ([Per01-a]), where they can be identified by a pattern number (**P-number**) and a title. Each pattern is composed of several sections that explain different qualities of the pattern. In the example patterns (see tables 2 and 3), the solution and the example sections are presented.

- Migration expressions patterns

There is a pattern for each one of the element's properties that can be changed by the schema evolution process and for any of the possible combinations of them. Each pattern produces a migration expression or a set of migration expressions specifying the correct transformation of data.

The generation of the migration expressions for a new element consists of determining which old element is related to it in the mapping and consulting their different properties. Next, apply the specific element pattern that specifies the migration expression code for the updated properties is instantiated and the resulting migration expressions are generated. Finally, these expressions are included in the new element module.

When the data migration plan is executed, the generated migration expressions of an element will be evaluated and the instances migrated to the new database. An example of a migration expression pattern is the one for an *attribute* when the "*name*", the "*data type*" and the "*not null value*" properties change (**P-12).**

P-12: Pattern for an attribute when the "name", the "data type" and the "not null value" properties change.
Solution
The solution presents the generic migration expressions that specify the attribute changes of "name", "data type" and "not null value" properties. In this case, as in the **P-04**[3] and **P-08**[4] patterns, it is necessary to perform a type conversion in the transformation function as follows: old_data_typeTOnew_data_type (old_attribute) This pattern is a composition of the "name", the "data type" and the"not null value" property patterns (**P-03**[5]**, P-04** and **P-06**[6] or **P-08** and **P-10**[7] patterns). The migrations expressions that express these changes are the following: Filter: IDENT_clase '.' IDENT_attr <> NULL Transformation_Function: generic_func '(' IDENT_clase '.' <IDENT_attr> ')'
Example
The prices of the products were in pesetas, and now they must be in euros. As a result, the data type of the *UnitPrice* attribute was natural in the old schema and is real in the new schema and the *Price* value must be converted to €. In addition, *Number* allowed null values and now does not allow null values. 

[3] **P-04:** Pattern for an attribute when the "data type" property change.
[4] **P-08:** Pattern for an attribute when the "name" and the "data type" properties change.
[5] **P-03:** Pattern for an attribute when the "name" property change.
[6] **P-06:** Pattern for an attribute when the "not null value" property change.
[7] **P-10:** Pattern for an attribute when the "name" and the "not null value" properties change.

```
Text Format
   Filter: OldCS.Product.UnitPrice <> NULL
      Transformation_Function: NatToReal(OldCS.Product.UnitPrice)
XML Format
   <Filter> OldCS.Product.UnitPrice <> NULL </Filter>
     <Transformation_Function> NatToReal(OldCS.Product.UnitPrice)
      </Transformation_Function>
```

Table.2 Solution and Example sections of the pattern P-12: Transformation function for the change of name, data type and not null value properties

In our example, ADAM uses the necessary patterns for each one of the established correspondences between the attributes of the new *Product* class and the old *Product* class. Moreover, we need take into account that the transformation function generated by the pattern (NatToReal(OldCS.Product.UnitPrice)) must be modified by the user in order to add the currency conversion function[8]. As a result, the transformation function that will be included in the data migration plan is PtsToEuro(NatToReal(OldCS.Product.UnitPrice)).

- Migration patterns

Migration patterns establish the way of migrating data for each type of conceptual schema element together with the necessary actions to migrate each type of conceptual schema element and the allowed migration expressions for each one. We must take into account, at the design process of a migration pattern, the type of conceptual schema element that will use the pattern, because the transformations that can undergo each conceptual schema element are different.

Moreover, migration patterns specify the expression place inside the module for each type of conceptual schema element, in the internal structure of a module.

A type of a conceptual schema element can have different associate patterns because there are different properties that influence the migration process. For example, the migration of a specialization relationship is different if its condition is more restrictive or less restrictive than the previous one or different because we must apply different types of filters on the data and different migration expression in a different place.

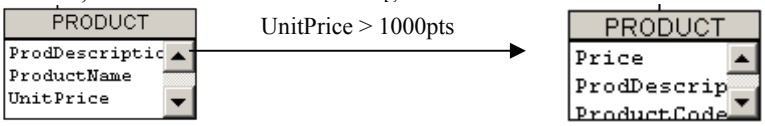
An example of a migration pattern is the pattern of the elemental class (**P-01)**:

P-01.Pattern: Elemental class
Solution
Let S be a set of schemas, C be an alphabet of classes, A be a set of attributes, G be a set of filters that are applied on old class population, GC be a set of conditions that are applied over old attributes, F be a set of transformation functions, SM be a set of matches between conceptual schemas, CM be a set of matches between classes of new and old conceptual schemas, and AM be a set of matches between attributes of conceptual schemas. $S1, S2 \in S \wedge \ S1.C1, S2.C2 \in C \ \wedge \ S1.C1.a1, S2.C2.a2 \in A \ \wedge \ f_1, .., f_n \in F \wedge \ g_1,...,g_n \in G \ \wedge$ $gc_1,...,gc_n \in GC \wedge \ SM1 \in SM \ \wedge \ CM1 \in CM \ \wedge \ AM1 \in CA \ \wedge \ SM1.old=S1 \ \wedge \ SM1.new=S2 \ \wedge$ $CM1.old=S1.C1 \wedge CM1.new=S2.C2 \wedge AM1.old=S1.C1.a1 \wedge AM1.new=S2.C2.a2 \rightarrow$ data (S2.C2) = $\{y \mid \exists x \in data(S1.C1) \ \wedge \ \forall i \vDash_x g_i \ i=1,..,n \wedge ((y.a2 = f_n of_{n-1}...of_1(x.a1) \vee y.a2 = cte) \wedge \forall i \vDash_{x\downarrow a1} gc_i)\}$

[8] The PtsToEuros conversion function is included in the ADAM's set of built-in transformation functions.

Example

The *Product* class of the new conceptual schema obtains its data from the *Product* class of the old conceptual schema. However, the analyst of this system is only interested in the products that have a price higher than 1000. Moreover, all its attributes must be migrated with its transformations and conditions.



Text Format:

S1, S2 ∈ S ∧ S1.Product, S2.Product ∈ C ∧ S1.Product.Cod_Product, S1.Product.ProductName, S1.Product.ProdDescription, S1.Product.UnitPrice, S2.Product.ProductCode, S2.Product.Price, S2.Product.ProductName, S2.Product.ProdDescription ∈ A ∧ NatTOReal, RightTrunc, PtsToEuro ∈ F ∧ {S1.Product.UnitPrice > 1000pts}, {S1.Product.UnitPrice <> NULL} ∈ G ∧ SM1 ∈ SM ∧ CM1 ∈ CM ∧ AM1, AM2, AM3.AM4 ∈ CA ∧ SM1.old=S1 ∧ SM1.new=S2 ∧ CM1.old=S1.Product ∧ CM1.new=S2.Product ∧ AM1.old=S1.Product.Cod_Product ∧ AM1.new=S2.Product.ProductCode ∧ AM2.old=S1.Product.ProductName ∧ AM2.new=S2.Product.ProductName ∧ AM3.old=S1.Product.ProdDescription ∧ AM3.new=S2.Product.ProdDescription ∧ AM4.old=S1.Product.UnitPrice ∧ AM1.new=S2.Product.Price → data (S2.Product) = { y | ∃x ∈ data(S1.Product) $\models_x$ (S1.Product.UnitPrice > 1000 AND S1.Product.ProductName <> NULL AND S1.Product.ProdDescription <> NULL AND S1.Product.UnitPrice <> NULL ∧ (y.ProductCode = x.Cod_Product) ∧ (y.ProductName = x.ProductName) ∧ (y.ProdDescription = RightTrunc(x.ProdDescription) ∧ y.Price = PtsToEuro(NatToReal(x.UnitPrice)) }

XML Format:

```
<New_Conceptual_Schema>
 <Class>
  <Name> Product </Name>
  <Origin>
   <Name> Product </Name>
   <Filtered>
    <Filter> OldCS.CS1.Product.UnitPrice > 1000 AND
          OldCS.Product.ProductName <> NULL AND
          OldCS.Product.ProdDescription <> NULL AND
          OldCS.Product.UnitPrice <> NULL
    </Filter
    <Attribute>
     <Name> ProductCode </Name>
     <OriginAttribute> OldCS.Product.Cod_Product </OriginAttribute>
     <Transformation_Function> OldCS.Product.Cod_Product
     </Transformation_Function>
    </Attribute>
    <Attribute>
     <Name> ProductName </Name>
     <OriginAttribute> OldCS.Product.ProductName </OriginAttribute>
     <Transformation_Function> OldCS.Product.ProductName
     </Transformation_Function>
    </Attribute>
    <Attribute>
     <Name> ProdDescription </Name>
     <OriginAttribute> OldCS.Product.ProdDescription </OriginAttribute>
     <Transformation_Function> RightTrunc(OldCS.Product.ProdDescription)
     </Transformation_Function>
    </Attribute>
```

```
      <Attribute>
        <Name> Price </Name>
        <OriginAttribute> OldCS.Product.UnitPrice</OriginAttribute>
        <Transformation_Function> PtsToEuro(NatToReal(OldCS.Product.UnitPrice))
        </Transformation_Function>
      </Attribute>
    </Filtered>
  </Origin>
</Class>
</New_Conceptual_Schema>
```

Table.3 Solution and Example sections of the pattern P-01: Elemental Class

4. Related work

Nowadays, data evolution support is given by DBMS: Relational DBMS such as ORACLE ([Ora02]) or SQL Server ([Sql02]) and object-oriented DBMS like O2 ([Ard02]), Poet ([Poe02]) or Versant ([Ver02]).

Existing works in the schema evolution field focuses in the consistence problems between modified schemas and the corresponding databases. A set of invariants were defined by [Kim89] and several DBMS uses the same or similar invariant set. Existing solutions vary from those that use schema evolution (OTGen [Ler90] or Cocoon [Tre92]) to those that use schema versioning or views (Avance [Bjø88], CLOSQL [Mon93] or Multiperspectives [Odb95]). Our work follows the schema evolution solution due to our interest in application dynamics in order to be compliant with new requirements.

Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done by means of SQL statements or user defined scripts which can be executed on the database. However, these tools do not provide automatic support for the generation of these statements and scripts as theADAM tool does. For this reason, DB administrators must write migration code by hand. ETL tools offer a friendly interface to migrate data from an old database to a new database. However, they do not solve two important problems at the software maintenance stage: the high number of people required for the migration process and the high temporal cost.

The closest point of view to our approach is the TESS tool presented by Barbara Staund Lerner in her paper [Sta00]. Both approaches have two things in common: the processes are automatic and the tools are based on schema evolution. TESS uses an intermediate language generated from the relational schema code. This is an important difference with our approach, because we deal with directly the OO conceptual schemas and we do not have to translate them to an intermediate language. The OO conceptual schemas give us a higher abstraction level and save us from the translation process. It also implies that we do not have to take implementation details into account and so our process is more agile and has a higher abstraction level than the TESS process.

5. Further work and conclusions

The paper presents a solution to the data evolution problem of information systems. It explains the importance of this evolution problem and the need to support data changes in information systems.

The presented solution uses conceptual schemas, which represent the system evolution, to generate a data migration plan. This data migration plan is able to evolve and migrate data between the

respective databases of the conceptual schemas. The automatic generation process gives us a first version of a data migration plan that can be modified by the analyst. The contents and the structure of this first version of a data migration plan are generated by means of a set of patterns.

This solution has been implemented in a migration tool whose name is ADAM. Nowadays, ADAM is being used together with Sosy Modeler®, a model compiler. ADAM has achieved to reduce the time invested in the maintenance stage and the number of employees involve in the data evolution task up to 80 % in real experiences.

The high abstraction level of the migration language ADML allows us to be independent from the underlying DBMS. In order to use a given DBMS, it is only necessary to construct its specific "driver". ADAM specifically is a compiler which generates DTS packets of SQL Server from migration plan .Using ADAM we do not care of implementation details at the comparison and plan generation processes. The final "driver" will do automatically the work.

Finally, it is important to note that it is possible to improve the automatic migration process. Therefore, there are some aspects that we must solve in the future: the migration language needs to be able to specify migration expressions that contain elements of the new conceptual schema and the migration process should take into account also the dynamics of the system (States Transition Diagrams (STD)), etc.

6. Bibliography

[Aba01] Abad, S, Carsí, J.A, Ramos, I, *How to obtain a data migration order for the elements of an OO conceptual schema*, Workshop on Evolution in the VI conference on Software Engineering and Databases , Almagro, Ciudad Real, Spain, November 2001 (in Spanish)

[Ana01] Anaya, V. *Generation of transformation modules to migrate data between databases using a data migration plan*, Msc Project, Computer Science Faculty, Valencia University of Technology, September 2001 (in Spanish)

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press. 1979.

[Ard] Ardent Software, *O2*, http://www.ardent.com/

[Bjø88] Bjørnerstedt A., Britts S., *Avance - An object management system*, Proc. of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA), San Diego, California, USA, pag. 206-221, September 1988.

[Cha00] Chaffin M., Knight B., Robinson T., Professional SQL Server 2000 DTS (Data Transformation Services), Wrox, 2000

[Gam94] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addyson-Wesley.1994.

[Kim89] Kim W. and Lochovsky, F.H. (eds.), *Object-Oriented Concepts, Databases and Applications*, ACM Press, Addison-Wesley, 1989.

[Ler90] Lerner B.S., Habermenn A.N., *Beyond Schema Evolution to DatabaseReorganization*, In Proc. Of the Joint Conference on Object-OrientedSystems, Languages and Applications (OOPSLA) and ECOOP, Ottawa, Canada,pag.67-76, Octubre 1990.

[Mon93] Monk, S., *A Model for Schema Evolution in Object-Oriented Database Systems*, Ph. D. Thesis, Computing Department, Lancaster University, Febrero 1993.

[Odb95] Odberg, E., *Multiperspectives: Object Evolution and Schema Modificacion Management for Object-Oriented Databases*, Ph.D. Thesis, Department of Computer Systems and Telematics, Norwegian Institute of Technology, Febrero 1995.

[Ora] Oracle Corporation, *Oracle*, http://www.oracle.com

[Pas95] Pastor O. Et al., *OASIS versión 2 (2.2): A Class-Definition Language to Model Information Systems using an Object-Oriented Approach*, SPUPV-95.788, Universidad Politécnica de Valencia, España, 1995.

[Pas97] Pastor O. Et al, *OO-METHOD: A Software Production Environment Combining Conventional and Formal Methods*, Procc. of 9th International Conference, CaiSE97, Barcelona, 1997.

[Per01-a] Pérez, J. *Generation of a data migration plan for OASIS conceptual schemas*, Msc Project, Computer Science Faculty, Valencia University of Technology, September 2001 (in Spanish)

[Per01-b] Pérez J., Carsí J.A, Ramos I., Anaya V., Silva J.F., D*ata migration of conceptual schemas*, Workshop on Evolution in the VI conference on Software Engineering and Databases , Almagro, Ciudad Real, Spain, November 2001 (in Spanish)

[Poe] POET Software, *POET*, http://www.poet.com

[Rat] Rational Software, *Rational Rose*, http://www.rational.com/products/rose/

[Ser94] Sernadas A., Costa J.F., Sernadas C., "Object Specifications Through Diagrams: OBLOG Approach" INESC Lisbon 1994

[Sil02] Silva, J.F.., Carsí, J.A., Ramos, I., *Theoric analyze of the criteria of OO conceptual schemas comparison,* Ingeniería Informática Magazine, ISSN:0717-4197, Enero, http://www.inf.udec.cl/revista/edicion7/jsilva.htm

[SQL] Microsoft, *SQL Server*, http://www.microsoft.com/sql

[Sys] System Architec, http://www.popkin.com/products/sa2001/systemarchitect.htm

[Sta00] Staund Lerner, B., *A Model for Compound Type Changes Encountered in Schema Evolution*, ACM Transactions on Database Systems (TODS) Marzo 2000, Volumen 25 número 1

[Tre92] Tresch M., Scholl M.H., *Meta object Management and its applications to database evolution*, In Proc. Of the 11th International Conference on the Entity-Relationship Approach, Karlsruhe, Germany, pag.299-321, Octubre 1992

[Tog] TogetherSoft Corporation http://www.togethersoft.com/

[Ver] Versant Object Technology, *Versant*, http://www.versant.com/

ADML: a Language for Automatic Generation of Migration Plans

Jennifer Pérez, José A. Carsí, Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n
E-46071 Valencia – Spain
{jeperez|pcarsi|iramos}@dsic.upv.es

Abstract. This paper presents a solution to the data evolution problem of information systems. This solution follows an automatic approach that reduces the number of people and the time invested in the software maintenance process. It uses the information of UML-like OASIS OO conceptual schemas, representing system evolution, in order to automatically generate a data migration plan. This work defines a specification language (ADML) for migration plans which are automatically generated using patterns. The plan execution migrates data from the database of the initial conceptual schema to the database of the evolved conceptual schema. This solution has been implemented in a migration tool (ADAM).

Key Words: Data migration plan, migration language, patterns, migration expressions, automatic generation.

1. Introduction

Information systems have a dynamic nature and they frequently undergo changes. These changes are due to volatile business rules or to external factors. For this reason, system management tools must be able to deal with them in an efficient and complete way.

Existing model compilers generate the database schemas and the application code from its conceptual schemas. The application generation can be complete, like Oblog Case ([Ser94]), OO-Method/CASE ([Pas97]), or partial, like Rational Rose ([Rat02]), System Architect ([Sys02]), Together ([Tog02]) and others. However, these modeling tools do not deal with the data evolution problem. In other words, they do not take the evolution of the data stored in databases into account. In this paper a solution to the data evolution problem is presented.

Nowadays, our solution is used together with OO-Method/CASE, a model compiler generating automatically applications with three layer architecture (presentation, business logic and persistence). A formal language OASIS

Pérez, J., Carsí, J. A., Ramos, I. ADML: A Language for Automatic Generation of Migration Plans. Proc. of the First Eurasian Conference on Advances in Information and Communication Technology. 2002.

([Pas95]) based in dynamic logic is used as design language allowing the automatic compilation of UML-like models.

Our starting point to implement data evolution is the OASIS ([Pas95]) conceptual schemas that represent system evolution (the old and the new schemas). From the new schema, the OO-Method/CASE compiler generates a new code and a new empty database. Therefore, the information system data remaining in the old database must be correctly transferred to the new one. We propose a solution for the migration of data, consisting of three steps:

1.- Old and new conceptual schemas are compared using different comparison criteria ([Sil02]) to automatically obtain the differences between them.

2.- Changes on the data are obtained from these differences and are specified as migration expressions using a migration language (ADML). These expressions constitute a data migration plan ([Per01-b]).

3.- The execution of the plan migrates data from an old database to a new database ([Ana01]).

This approach has been implemented in a migration tool (ADAM) developed by the Department of Information Systems and Computation of the Valencia University of Technology in collaboration with the industrial partner CONSOFT. We have obtained very good results applying this solution. The time invested and the number of people involved in data evolution process have decreased up to 80%.

ADML captures all changes undergone in the schema and its related updates on the persistence layer. It is accurate and complete. This language is not ambiguous and has a high abstraction level. ADML and different migration expression patterns ([Per01-a]) are used to generate a data migration plan based on the OASIS object-oriented model.

The structure of the paper is the following: first, the process of generating expressions is explained and then, the ADML language and patterns used to generate a data migration plan are introduced. In section 3, the article presents a brief summary of related work. Finally, future work and conclusions are presented in section 4.

2. A Data Migration Language (ADML) and the Patterns Used to Capture Migration Features

A data migration plan is a set of data changes that are specified using a migration language. Its execution allows us to transfer data from an old database to a new database. A data migration plan must include all the necessary changes in order to perform a correct migration and in the right order. It is structured as the composition of different elements: modules, changes and migration expressions. The highest abstraction level of this composition is constituted by migration modules whereas migration expressions are at the lowest level. The definition of these elements is as follows:

– *Migration expressions:* Migration expressions are those expressions that can be specified in a data migration plan. Each type of migration expression has a different semantic and follows different syntactic restrictions.
– *Changes:* A change is the set of migration expressions that specify the updates undergone and the filters applied on the old element's[1] instances.
– *Migration modules :* A migration module has a transactional behavior when it is executed by ADAM tool. The composition of modules using aggregations forms a data migration plan. Finally, it is important to note that there is one migration module for each class, aggregation and specialization of the new conceptual schema.

ADML is declarative and is used to specify all expressions that make up a data migration plan. Object-oriented conceptual schema elements are the data that are managed by the migration language. This language follows the object-oriented model. ADML makes use of path expressions to specify:

- The changes undergone in a conceptual schema element.
- The data belonging to the old conceptual schema.
- The filters that will be applied on the data, if necessary.

In the paper, the well known example order-customer-products in Figures 1 and 2 is used. We take a part of the complete example and we focus on the language and the generation of the migration expressions of this plan. The following OASIS schemas capture the structure of the order management of an information system. The new conceptual schema (figure 2) improves the previous one (figure 1). This improvement consists in a classification of the customers, a better management of the orders and some corrections of model errors. The example is the following:

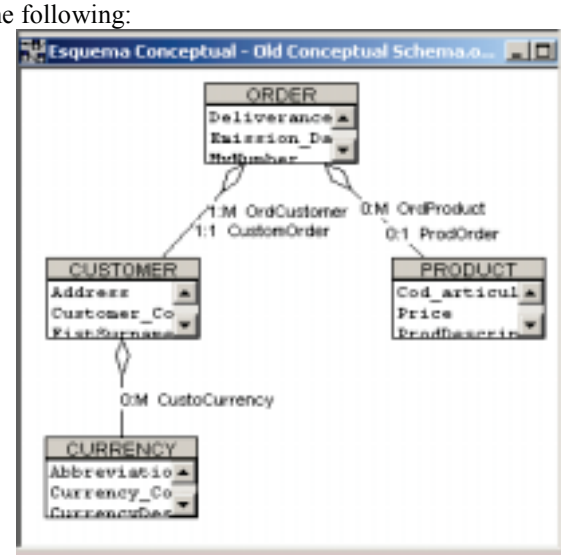


Figure 1. OASIS Old Conceptual Schema of the IS Orders Management

[1] The conceptual schema elements that has been considered for the generation of a data migration plan are: classes, attributes, aggregations, associations and inheritances.

It must be taken into account that these schemas are modeled using OASIS. However, if we model them using UML, their appearance would be different. This is due to the fact that in OASIS the association is considered as a kind of aggregation and thus, the referential aggregation and the inclusive aggregation are represented by a lozenge. This is the reason why UML associations are modeled like OASIS referential aggregations. For example: The relationship between the classes *Customer* and *Currency* is an association represented with a OASIS referential aggregation.

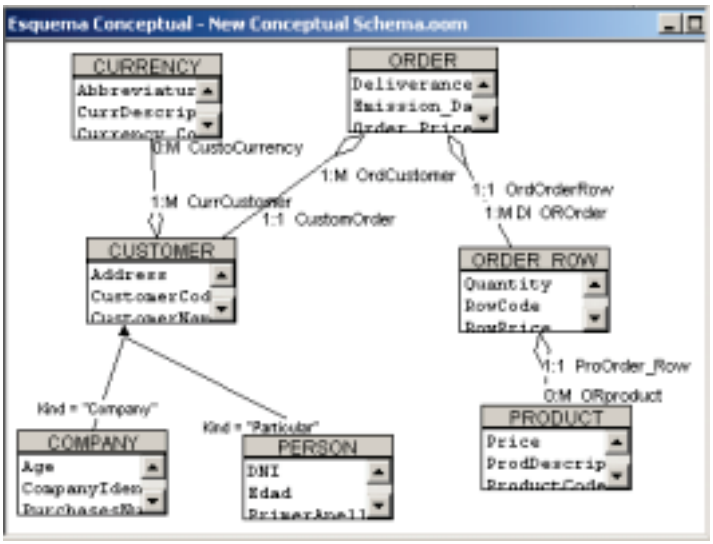


Figure 2. OASIS New Conceptual Schema of the IS Orders Management

2.1. A Data Migration Language (ADML)

The ADML specifies of the data evolution by means of migration expressions. Its main advantage is the independence from any DBMS due to its high abstraction level. The high abstraction level allows us to express a migration plan in an easy and user friendly way and it does not need to take into account implementation details. In addition, the migration language can specify five kinds of expressions allowing us to define the migration module of each kind of conceptual schema element. These migration expressions are the following:

- *Origin:* The origin specifies the data source for an element of the new conceptual schema. The data source is the old element's contents that the analyst wants to migrate to the new one. The origin must be specified for all the element migrations of the plan. For example, the origin of the *Quantity* attribute of *Order_Row* class is the *Number* attribute of *Order* class. The origin of *Quantity* is specified by the migration tool as follows:

```
Origin: OldCS.Order.Number
```

- *Filter*: The filter selects the population that will be migrated from an old element to another new element. The filter can restrict the population of a class or an aggregation. The filter is a boolean condition that will be evaluated at execution.

For example, if the analyst of this system is only interested in the orders made after 1999, then this restriction can be specified by means of a filter in the *Order* class and in the *Order_ Row* class. The *Order_ Row* filter is specified as follows:

```
Filter: OldCS.Order.Issue_date > 31/12/1999
  Origin: OldCS.Order
```

– *Sorting*: The migration order of the instances is represented in the sorting section. This expression specifies the order using the class attributes and gives its direction (ascendant or descendant). For example, if the analyst wants to migrate the orders sorted by *emission_date* and *customer* in ascending order, the *Order_ Row* order is specified in the migration tool as follows:

```
Filter: OldCS.Order.Issue _date > 31/12/1999
 Origin: OldCS.Order
 Sorting: OldCS.Order.Issue _date ASC,
          OldCS.Order.CustomOrder.Cod_customer ASC
```

– *Transformation function*: The transformation function is the migration expression that specifies the changes that the data will undergo during the process of migration. When the data need more than one change, the transformation function is the composition of changes. This expression transforms the old data to be compliant with the new element definition. That is, the transformation function execution makes the data evolution of one element. This migration expression can only be specified when migrating attributes, specializations and aggregations. The transformation function always is executed. However, if it depends on a condition, it will only be executed when the condition is true. For example, the migration tool applies a type conversion on the *Number* attribute to migrate its data to the *Quantity* attribute. The transformation function of the *Quantity* attribute is specified by the migration tool as follows:

```
Origin: OldCS.Order.Number
  Transformation function: RealToNatural(OldCS.Order.Number)
```

– *Condition*: The condition is the migration expression that determines which transformation function will be executed. This migration expression can only be specified for attributes. Depending on the case, one attribute can have more than one possible transformation function associated to it. Each one is represented by means of a pair (condition, transformation function). The value of the condition determines which transformation function will be executed. For example, the migration tool applies a type conversion on the *Number* attribute to migrate its data to the *Quantity* attribute. This conversion will only be executed if the *Number* has decimals. The transformation function of the *Quantity* attribute is specified in the migration tool as follows:

```
Origin: Order. Number
     Condition: HasDecimals(OldCS.Order. Number) = true
     Transformation function: RealToNatural(OldCS.Order.Number)
     Condition: HasDecimals (OldCS.Order. Number) = false
     Transformation function: OldCS.Order.Number
```

ADML specifies conceptual schema elements and relationships between them. The navigation through elements gives flexibility and expressivity to construct migration expressions. For example, one can express a filter of the *Customer* class by means of

the *Emission_Date* attribute which belongs to the *Order*class that is related to the *Customer* class (`EXIST(CUSTOMER.OrdCustomer.Emission_Date > 31/12/1999`)).

For the moment, all elements that can be specified in a migration plan must belong to the old conceptual schema. This is a limitation of the first version of ADML language that we are currently solving. Migration languages must allow us to specify new conceptual schema elements, despite the fact that this flexibility implies a lot of complexity in the migration tasks. A good example of this increase in complexity is the setting of the *migration order* ([Aba01]). The inclusion of new elements in a data migration plan is more complex than specifying only old elements, because a new element must be migrated before using it in a migration expression.

ADML uses simple and iterative control structures. These simple structures analyze different migration cases of an element. The iterative structures can treat each one of the instances of an element. The migration language is more flexible thanks to these structures and is able to specify the same expression in different ways. For example, one may specify a condition and a transformation function using different expressions or specify a transformation function including an "if statement" with the condition and the transformation function :

```
Transformation function: if HasDecimals(OldCS.Order. Number) = true then
                              RealToNatural(OldCS.rder.Number)
                         Else
                              OldCS.Order.Number
                         End if
```

The last feature of the grammar are operators. It has different operators (arithmetic, boolean and aggregate functions) to express actions that the analyst might need. For this reason, it was necessary to build new operators to perform specific actions that were not possible with the standard operators. For example: FIRST(data source, condition, number of elements). This operator is useful when the cardinality of a relationship has been changed, because it allows to migrate only the highest cardinality of elements that satisfy the condition.

2.2. Generation of Migration Expressions

ADAM automatically generates the first version of a data migration plan. This version could be the final one; however, it can be updated by the analysts to suit their purpose.

The automatic generation of a data migration plan implies generating its structure and its contents. The structure generation creates a migration module for each new element and includes them in the data migration plan following the migration order. The content generation consists in automatically constructing all the migration expressions of a complete migration plan. This paper is focused on the generation of the contents using ADML and a set of patterns ([Alexander79] y [Gamma94]). To do this, two data sources are necessary:

- *Mappings*: Mappings specify the data sources of the new conceptual schema elements ([Sil02]). This information is generated previously by ADAM tool. The mappings are produced by comparing the old element and the new one in order to identify the differences between them.

– *The conceptual schemas:* Conceptual schemas contain all the properties and the types of their elements. The migration tool needs this information in order to determine which properties have been changed between an old element and a new element in the previous mapping. The properties of the elements must also be known in order to determine the pattern that the migration tool must apply.

There is a pattern for each one of the element properties that can be changed by schema evolution and for all the combinations of the patterns. This set of patterns is included in a pattern catalog [Per-a] (P-01, P-02). Each pattern has a set of migration expressions which allows us to specify the correct transformation of data.

The generation of the migration expressions for a new element consists of determining which old element is related to it in the mapping and consulting their different properties. Later, ADAM applies the specific element pattern that specifies the generation for the updated properties and generates the migration expressions. Finally, the tool associates these expressions to the new element.

When the data migration plan is executed, the migration expressions of an element which are generated will be evaluated in its instances and these will be migrated to the new database. These instances will satisfy the properties of the new database.

Each pattern is composed by several sections that explain different qualities of the pattern. An example of migration expression pattern arises when the "name", the "data type" and the "not null value" properties of an attribute change. The solution and the example sections of this pattern (**P-12**) are as follows:

Solution

The solution presents the generic migration expression that specifies the attribute changes of "name", "data type" and "not null value" properties. In this case, as in the **P-04** and **P-08** patterns, it is necessary to do a type conversion in the transformation function as follows:

```
old_data_typeTOnew_data_type (old_attribute)
```

This pattern is a composition of the name, data type and "not null value" property patterns (**P-03, P-04** and **P-05** or **P-08** and **P-10** patterns). The migrations expressions that express these changes are the following:

`Filter:` <IDENT_clase> . <IDENT_attr> <> NULL

`Transformation Function:` <generic_func> '(' <IDENT_clase> . <IDENT_attr> ')'

Example

The origin of *Quantity* attribute of *Order_ Row* class is the *Number* attribute of *Order* class. The *Number* attribute of the old conceptual schema is called *Quantity* in the conceptual schema. In addition, *Number* allowed null values and had float type, and now *Quantity* does not allow null values and has natural type in order to correct a model error of the old schema.

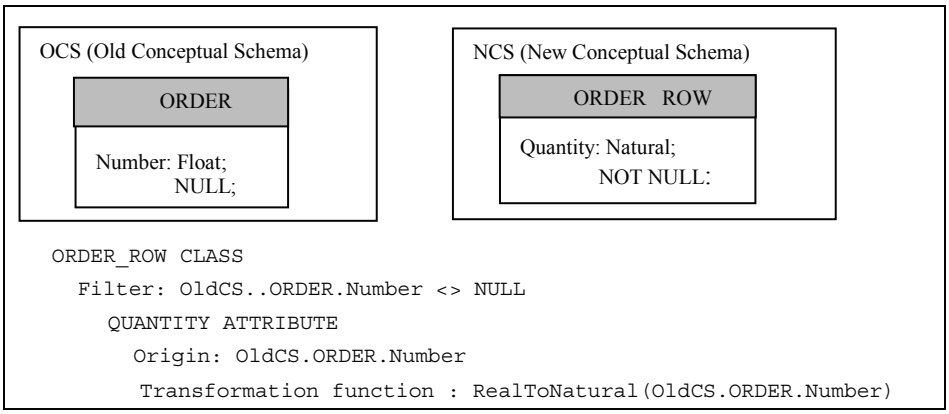


Figura.2 Solution and Example sections of the pattern P-12: Transformation function for the change of name, data type and not null value properties

After applying the necessary patterns to automatically generate the migration module of the *Row_Order* class and specifying the sentences that the analyst desires, the migration tool obtains the following code:

```
ORDER_ROW CLASS
  Origin: OldCS.Order
Sorting: OldCS.Order.Emission_date ASC,
         OldCS.Order.CustomOrder.Cod_customer ASC
   Filter:( OldCS.Order.Emission_date > 31/12/1999) AND
           (OldCS.Order.Number <> NULL)
      ROW_CODE ATTRIBUTE
        Origin:λ
        Transformation function: 1
      END ROW_CODE ATTRIBUTE
      QUANTITY ATTRIBUTE
        Origin: OldCS.Order.Number
        Condition: Decimal(OldCS.Order. Number) <> 0
        Transformation function: RealToNatural(OldCS.Order.Number)
        Condition: Decimal(OldCS.Order. Number) = 0
        Transformation function: OldCS.Order.Number
      END QUANTITY ATTRIBUTE
      ROW_PRICE ATTRIBUTE
        Origin: OldCS.Order.Order_price
        Transformation function: OldCS.Order.Order_price
        END ROW_PRICE ATTRIBUTE
END ORDER_ROW CLASS
```

3. Related Work

Nowadays, data evolution support is given by DBMS. Relational DBMS such as ORACLE ([Ora02]) or SQL Server ([Sql02]) and object-oriented DBMS like O2 ([Ard02]), Poet ([Poe02]) or Versant ([Ver02]).

Several DBMS allow for data migration using their ETL (Extract, Transforma & Load) tools. This migration can be done by means of SQL statements or user defined scripts which can be executed on the database. However, these tools do not provide automatic support to generate these statements and scripts as ADAM tool does. For this reason, DB administrators must write migration code by hand. ETL tools offer a friendly interface to migrate data from an old database to a new database. However, they do not solve two important problems at software maintenance stage: the high number of people required for the migration process and the high temporal cost.

The closest point of view to our approach is the TESS tool presented by Barbara Staund Lerner in her paper [Sta00]. These approaches have two things in common, both processes are automatic and both tools are based on schema evolution. TESS processes an intermediate language generated from a relational schema code. This is an important difference from our approach, because we analyze the OO conceptual schemas and we do not have to translate to an intermediate language. The OO conceptual schemas give us a higher abstraction level and save us from the translation processes. It also implies that we do not have to take implementation details into account and so our process is more agile and has a higher abstraction level than the TESS process.

4. Conclusion and Future Work

The paper presents a solution to the data evolution problem of information systems. It explains the importance of this evolution feature and the need to support data changes in information systems.

The presented solution uses conceptual schemas, which represent system evolution, to generate a data migration plan and evolve data. The automatic generation gives us a first version of a data migration plan. The contents of a data migration plan are defined by a set of patterns and they are composed of migration expressions that are specified using our migration language (ADML).

The high abstraction level of ADML allows us to be independent from the underlying DBMS. In order to use a given DBMS, it is only necessary to construct its specific “driver”. We do not take implementation details into account at the comparison and plan generation processes. The high automation of this solution also reduces the time invested in the maintenance stage and the number of employees involve in the data evolution task up to 80 % in our own experiences.

Finally, it is important to note that it is possible to improve the automatic migration process. Therefore, there are some aspects that we must solve in the future: the migration language needs to be able to specify migration expressions that contain elements of the new conceptual schema and the migration process should take into account the dynamics of the system (States Transition Diagrams (STD)), etc.

Bibliography

[Aba01] Abad, S, Carsí, J.A, Ramos, I, *How to obtain a data migration order for the elements of an OO conceptual schema*, Workshop on Evolution in the VI conference on Software Engineering and Databases , Almagro, Ciudad Real, Spain, November 2001 (in Spanish)

[Ana01] Anaya, V. *Generation of transformation modules to migrate data between databases using a data migration plan*, Msc Project, Computer Science Faculty, Valencia University of Technology, September 2001 (in Spanish)

[Alexander79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press. 1979.

[Ard] Ardent Software, *O2*, http://www.ardent.com/

[Gamma94] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addyson-Wesley.1994.

[Ora] Oracle Corporation, *Oracle*, http://www.oracle.com

[Pas95] Pastor O. Et al., *OASIS versión 2 (2.2): A Class-Definition Language to Model Information Systems using an Object-Oriented Approach*, SPUPV-95.788, Universidad Politécnica de Valencia, España, 1995.

[Pas97] Pastor O. Et al, *OO-METHOD: A Software Production Environment Combining Conventional and Formal Methods*, Procc. of 9th International Conference, CaiSE97, Barcelona, 1997.

[Per01-a] Pérez, J. *Generation of a data migration plan for OASIS conceptual schemas*, Msc Project, Computer Science Faculty, Valencia University of Technology, September 2001 (in Spanish)

[Per01-b] Pérez J., Carsí J.A, Ramos I., Anaya V., Silva J.F., *Data migration of conceptual schemas*, Workshop on Evolution in the VI conference on Software Engineering and Databases , Almagro, Ciudad Real, Spain, November 2001 (in Spanish)

[Poe] POET Software, *POET*, http://www.poet.com

[Rat] Rational Software, *Rational Rose*, http://www.rational.com/products/rose/

[Ser94] Sernadas A., Costa J.F., Sernadas C., "Object Specifications Through Diagrams: OBLOG Approach" INESC Lisbon 1994

[Sil02] Silva, J.F.., Carsí, J.A., Ramos, I., *Theoric analyze of the criteria of OO conceptual schemas comparison,* Ingeniería Informática Magazine, ISSN:0717-4197, Enero, http://www.inf.udec.cl/revista/edicion7/jsilva.htm

[Sys] System Architec, http://www.popkin.com/products/sa2001/systemarchitect.htm

[Sql] Microsoft, *SQL Server*, http://www.microsoft.com/sql

[Sta00] Staund Lerner, B., *A Model for Compound Type Changes Encountered in Schema Evolution*, ACM Transactions on Database Systems (TODS) Marzo 2000, Volumen 25 número 1

[Tog] TogetherSoft Corporation http://www.togethersoft.com/

[Ver] Versant Object Technology, *Versant*, http://www.versant.com/

Una plataforma semántica para la gestión de modelos

Artur Boronat, José Á. Carsí, Isidro Ramos

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
C/Camino de Vera s/n
E-46071 Valencia- España
{aboronat | pcarsi | iramos}@dsic.upv.es

Resumen. La gestión de modelos engloba todo tipo de tareas que permitan su representación y/o manipulación, entre ellas, la evolución de sistemas software, la migración de datos, la interoperabilidad, etc. En el trabajo se presenta una plataforma resultado de nuestra experiencia previa en dos proyectos industriales de migración de datos y de recuperación automática de aplicaciones legadas. La plataforma ha sido concebida y realizada en un contexto formal lo que permite deducción y verificación automática de propiedades de los modelos considerados. En nuestra aproximación, los modelos están representados por términos de un álgebra construidos según un determinado orden canónico. Estos términos están basados en la estructura básica de tripleta de RDF. La manipulación de modelos se centra en el establecimiento de correspondencias entre los términos que representan dos o más modelos. Estas correspondencias son expresadas mediante morfismos que permiten generar nuevos modelos y establecer relaciones entre modelos existentes. El uso de herramientas declarativas para la creación de los modelos y de los morfismos proporciona una gran modularidad que favorece la utilización de técnicas de patronaje para realizar estas correspondencias, la generación automática de modelos mediante reescritura de términos. Además, la plataforma puede ser vista como una gran red semántica formada por diferentes niveles de abstracción que permite la validación automática de la corrección de modelos.

Palabras clave: gestión de modelos, RDF, metainformación, mantenimiento de software.

1. Introducción

Según [2], "modelo" es una estructura compleja que representa un artefacto de diseño, como un esquema relacional, una interfaz orientada a objeto, un modelo UML, un DTD de XML, el esquema de una página web, una red semántica, un documento complejo, o una configuración de software. La gestión de modelos engloba todo tipo de tareas que permiten su representación y/o manipulación. De esta forma, se encuentra presente siempre que se trate con algún tipo de información, bien sea para uso humano, como la información que se almacena en la base de datos de una aplicación, o para uso del propio computador, como la información almacenada en un modelo RDF. El uso de modelos conlleva la gestión de cambios en los modelos

Boronat, A., Carsí, J. A., Ramos, I. Una plataforma semántica para la gestión de modelos. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos. 2003.

y las transformaciones de información de un modelo a otro, que implican la representación explícita de correspondencias entre estos modelos.

Este trabajo se centra en la gestión de modelos fruto de la experiencia previa en dos proyectos industriales de migración de datos y de recuperación automática de aplicaciones legadas. El primero (ver [6]) presenta una solución para la migración de información entre esquemas conceptuales orientados a objeto cuya capa de persistencia está formada por esquemas relacionales. Se aplica un algoritmo de comparación de esquemas cuyo resultado es analizado mediante patrones que generan un plan de migración relacional [11]. Este plan de migración establece las correspondencias entre elementos de ambas bases de datos indicando cómo se realiza la copia de información. El segundo (ver [10]) presenta una solución y un método para la recuperación de bases de datos legadas procedentes de varios SGBD usando técnicas basadas en métodos formales y patrones. Los métodos formales (sistemas de reescritura de términos) se aplican en el proceso de ingeniería inversa de datos y las técnicas de patronaje se utilizan durante el proceso de recuperación de los datos de la base de datos legada.

Ambos proyectos coinciden en el tratamiento de modelos de información mediante técnicas formales como la reescritura de términos o la utilización de patrones para conseguir una mayor automatización del proceso y así reducir su coste. Mediante la plataforma de gestión de modelos se pretende definir una base formal que permita representar cualquier modelo y operacionalizar en general estas técnicas. El objetivo de la plataforma no se restringe al campo de aplicación presentado por ambos proyectos anteriores sino que será la base que permitirá realizarlos de una forma integrada en una misma herramienta. A la vez, se dará soporte a proyectos con otros objetivos como la interoperabilidad, la gestión automática de recursos web, etc.

En esta sección se ha introducido el marco en el que se encuadra el trabajo así como la motivación que nos ha llevado a él. En el apartado 2 se ofrece una visión global de la plataforma y en el apartado 3 se especifica su estructura, describiendo cada uno de los cinco niveles reflexivos que la forman y los mecanismos que permiten su formalización. En el apartado 4 se indican trabajos relacionados. Por último, en el apartado 5 se comentan las conclusiones y se citan los trabajos futuros.

2. Visión global

Se presenta una plataforma que permite la gestión y manipulación de modelos de datos así como de la información que permiten almacenar. La estructura básica utilizada para representar modelos de datos es la tripleta RDF sujeto-predicado-objeto, que permite describir las propiedades *objeto* que posee un elemento *sujeto* mediante relaciones *predicado* (ver figura 1). La versatilidad de esta estructura se basa en la posibilidad de utilizar un elemento *sujeto* como propiedad *objeto* de otro elemento *sujeto* distinto. De esta forma se pueden crear estructuras en forma de grafo que representen cualquier modelo de datos. Todo predicado tiene asociado un dominio y un rango y permite la bidireccionalidad de la relación (siempre existe la relación inversa). La representación de proposiciones en forma de tripletas permite la utilización de técnicas formales tanto para la representación de un modelo mediante

términos de un álgebra como para establecer relaciones entre ellos. Esta estructura básica proporciona una gran modularidad que facilita la aplicación de técnicas de patronaje para realizar correspondencias entre modelos y reglas de reescritura para la generación automática de modelos.

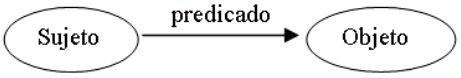


Fig. 1. Representación en grafo de la estructura básica utilizada para representar modelos

La plataforma está formada por una jerarquía de cinco niveles, donde cada nivel agrupa elementos con el mismo grado de abstracción. De esta manera, los elementos que constituyen un metamodelo son definidos en un nivel de abstracción. Los elementos que constituyen un esquema conceptual de ese metamodelo se sitúan en un nivel más concreto que el anterior. Por último, las instancias del esquema conceptual citado se ubican en un nivel aún más concreto. Los niveles que forman la plataforma son los siguientes, por orden de abstracción:

1. Nivel Base: especifica cómo crear los términos básicos que constituyen cualquier modelo RDF a partir del vocabulario RDF Schema.
2. Nivel Gestión de Modelos: forma el núcleo de la plataforma de gestión de modelos y proporciona un mecanismo común de definición de metamodelos o metaontologías para diferentes tipos de modelos de datos.
3. Nivel Metamodelo: alberga los metamodelos o metaontologías que han sido definidos a partir del Nivel de Gestión de Modelos y permite definir modelos de datos u ontologías en el nivel de Modelo.
4. Nivel Modelo: contiene los esquemas conceptuales que representan la información mediante estructuras de datos pertenecientes a un metamodelo determinado.
5. Nivel Información: en este nivel nos encontramos con la información que es tratada y almacenada en el modelo de datos.

Los elementos de un nivel intermedio pueden ser interpretados mediante el concepto de metaobjeto [5]. De esta manera pueden ser vistos como clases al describir elementos de niveles inferiores (más concretos) y como objetos al estar descritos por elementos de niveles superiores (más abstractos). Se utilizan dos mecanismos que permiten el cambio de abstracción de los elementos que pertenecen a un determinado nivel: la reflexión, que indica cómo un elemento específico del nivel N define un conjunto de elementos del nivel inmediatamente más concreto (N+1); y la reificación, mediante la cuál se indica, para un determinado elemento situado en un nivel N, el elemento del nivel inmediatamente más abstracto (N-1) que lo describe. En nuestra aproximación se colapsa reificación y tipado, puesto que para expresar las relaciones de reificación entre niveles utilizamos las propiedades de tipado que proporciona RDF, como se verá en el siguiente apartado.

3. Estructura de la plataforma semántica

A continuación se describen los diferentes niveles de abstracción que constituyen la plataforma, especificando los mecanismos de reflexión y reificación para cada uno de ellos.

3.1. Nivel Base

Es el nivel más abstracto de la plataforma de gestión de modelos y está constituido por una estructura de elementos básicos que permite la creación de modelos RDF en los niveles inferiores de la plataforma. Este nivel cierra el ciclo de reflexión que caracteriza la plataforma, basándose en la semántica definida para RDF [7]. Su función consiste en proporcionar un marco invariable sobre el que se especifican todos los elementos de la plataforma de gestión de modelos, tanto los propios modelos como aquellos elementos que permiten manipularlos. De esta forma, permitimos que la propia plataforma se pueda manipular con los mismos operadores que actúan sobre un modelo.

Con el objetivo de beneficiarse de las ventajas que proporciona la web semántica se utilizan elementos de RDFSchema en la creación del esquema conceptual de este nivel. Está formado por el mínimo conjunto completo de elementos de RDF Schema. Entendemos como conjunto mínimo completo a aquél que contiene el número mínimo de elementos necesario para crear cualquier modelo RDF completo (conjunto de constructores funcionalmente completo). En este nivel hemos considerado cinco elementos básicos del vocabulario RDFSchema: *rdfs:Resource*,*rdfs:Class*, *rdf:Property*, *rdf:type* y *rdfs:subClassOf*.

En [4], se propone como concepto central a *rdfs:Resource*, y como conceptos especializados de éste aparecen *rdfs:Class* y *rdf:Property* mediante la propiedad rdfs:subClassOf. Contradictoriamente, cuando se especifica la relación de instanciación entre estos elementos mediante el predicado *rdf:type*, *rdfs:Class* aparece como raíz de la jerarquía de instanciación. De esta forma *rdfs:Resource* y *rdf:Property* son instancias de *rdfs:Class*. Además, *rdfs:Class* cierra el mecanismo de instanciación indicando que es instancia de él mismo.

Como la semántica de instanciación que se da al predicado *rdf:type* en la especificación de RDF Schema conlleva un cambio de nivel de abstracción, se propone el grafo RDF de la figura 2 como nivel base de la plataforma. Éste contempla los elementos necesarios para especificar cualquier modelo RDF, es decir, permite definir nodos y relaciones entre nodos llamadas predicados.

El modelo está formado por los siguientes elementos:

- *rdf:type* : predicado que indica el tipo del nodo que se encuentra en el extremo origen de la relación *rdf:type*. Al mismo tiempo, este predicado es de tipo *rdf:Property*. La relación establecida por rdf:type es unidireccional puesto que no posee propiedades que la describen. Así pues, no se proporciona ningún mecanismo para deducir el nodo sujeto a partir del nodo objeto de la relación ni se establece ninguna restricción sobre su cardinalidad. En este nivel, el predicado rdf:type no implica ningún cambio de nivel de abstracción, puesto que se utiliza para definir la estructura básica de representación de modelos. En cambio, este predicado se utiliza como mecanismo de reificación entre diferentes niveles, indicando el tipo abstracto de un nivel superior usado para describir un nodo concreto en el nivel inferior.
- *rdfs:Class* : es el nodo raíz de la red semántica formada por la plataforma de gestión de modelos. Permite la creación de nodos en cualquier nivel de la plataforma mediante el predicado *rdf:type*. Un nodo de los niveles Base y de Gestión de Modelos debe tener como tipo a *rdfs:Class* directamente. Mientras que

en los demás niveles inferiores, un nodo debe tener como tipo a un subtipo de *rdfs:Class*. Este nodo se reifica en sí mismo, cerrando el salto a niveles abstractos superiores, puesto que no existen.

– *rdf:Property* : es un nodo de tipo *rdfs:Class* que permite crear predicados en cualquier nivel inferior de la plataforma. A pesar que utilice el predicado de reificación, se mantiene al mismo nivel de abstracción que *rdfs:Class* puesto que ambos nodos representan conceptos básicos que se necesitan para la definición de modelos RDF.

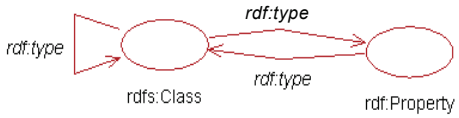


Fig. 2. Esquema conceptual del Nivel Base

En la plataforma cualquier esquema conceptual está formado por un conjunto de términos ordenados según un orden canónico. A continuación se establecen las reglas que dan lugar a estos términos y el orden entre ellas. Debemos establecer dos conceptos importantes para poder especificar correctamente los términos del álgebra:

– Nodo válido: un nodo es válido si se puede reificar en el nodo *rdfs:Class*. Es decir en la proposición *(S, p, O)*, el nodo *S* es válido si y sólo si existe una proposición de la forma *(S, rdf:type, rdfs:Class)*.
– Predicado válido: un predicado es válido si se puede reificar en el nodo rdf:Property. Es decir, en la proposición *(S, p, O)*, el predicado p es válido si y sólo si existe una proposición de la siguiente forma *(p, rdf:type, rdf:Property)*.

Un esquema conceptual está formado por términos de un álgebra. Se pueden crear dos tipos de términos:

– De creación de un nodo: *add(S, rdf:type, O)* donde *S* es el elemento que se crea y puede ser un nodo o un predicado, *rdf:type* es el predicado que define el tipo del nodo *S*, y *O* es *rdf:Class* si se está creando un nodo, o *rdf:Property* si se está creando un predicado.
– De creación de una proposición: *add(S, p, O)* donde S es el nodo válido al que se le añade una propiedad, *p* es el predicado válido que define la relación entre el nodo *S* y el nodo *O*, y *O* es el nodo válido que describe la propiedad que se añade a *S*.

A continuación se establece un orden canónico entre estos dos constructores de términos que permite validar la corrección de un esquema RDF durante su construcción: sólamente se puede añadir un constructor de la forma *add(S, p, O)* donde p es distinto de *rdf:type* si y sólo si *S* y *O* son nodos válidos y están en el mismo nivel, y *p* es un predicado válido definido en ese mismo nivel. Una manera sencilla para comprobar el orden canónico consiste en comprobar si existen los términos *add(S, rdf:type, ...), add(p, rdf:type, ...)* y *add(O, rdf:type, ...)* correspondientes.

Estos términos permiten describir el esquema conceptual de un determinado nivel en función de los elementos de un esquema conceptual situado en un nivel inmediatamente superior en la jerarquía de abstracción. De esta manera los términos que constituyen el esquema conceptual del nivel base se construyen sobre ellos

mismos y son los siguientes: *add(rdf:Class, rdf:type, rdfs:Class), add(rdf:Property, rdf:type, rdfs:Class), add(rdf:type, rdf:type, rdf:Property)*. Estos constructores constituyen el conjunto funcionalmente completo de la plataforma y permiten la creación de cualquier modelo RDF. El orden canónico no es aplicable a la secuencia de estos términos puesto que son necesarios los tres para poder crear cualquier elemento de un modelo RDF.

3.2. Nivel de Gestión de Modelos

Se encuentra entre el Nivel Base y el Nivel de Metamodelo en la jerarquía de abstracción de la plataforma. Este nivel se forma a partir de los términos definidos para el nivel Base y es el más abstracto de la plataforma que contiene mecanismos para la gestión de modelos. Entre ellos encontramos información de versionado de los elementos del esquema conceptual, información de seguridad, mecanismo de reflexión hacia el nivel Base y mecanismo de reificación hacia el nivel de Metamodelo. Su principal objetivo es actuar como soporte formal para crear y gestionar metamodelos y metaontologías.

Su esquema conceptual está formado por los elementos de la figura 3, donde el nodo *issi:MetaNode*, como cualquier otro nodo de este nivel, es de tipo *rdfs:Class* y constituye el nuevo tipo que describe a los nodos del nivel inferior. El nodo *issi:MetaProperty* forma un subtipo de *rdf:Property* mediante el predicado *rdfs:subPropertyOf* permitiendo describir y crear nuevos predicados en el nivel de Metamodelo. El nodo *issi:MetaProperty* tiene diferentes propiedades que especifican el tipo de los predicados que se crearán a partir de él. Así, se indica que un predicado puede tener como sujeto y como objeto a un nodo de tipo *issi:MetaNode* mediante *issi:domain* y *issi:range*, respectivamente. Este hecho caracteriza la bidireccionalidad de la relación establecida por un predicado de este tipo, puesto que un nodo objeto en una proposición puede deducir el nodo sujeto gracias a la propiedad issi*:domain* del predicado. Además un predicado de tipo *issi:MetaProperty* debe tener un sujeto y un objeto en la sentencia que aparezca, pero sólo uno, tal y como indica la cardinalidad mínima de la propiedad mediante el predicado *issi:minCardinality* y la máxima mediante el predicado *issi:maxCardinality*.

Los dos nodos principales poseen propiedades relevantes como información sobre seguridad (*issi:metaNodeSecurityInfo* e *issi:metaPropertySecurityInfo*), que permite administrar permisos a un usuario para gestionar nuevos metamodelos. Otra propiedad relevante es la expresada por los predicados *issi:metaNodeVersion* e *issi:metaPropertyVersion* que indican la versión del nodo correspondiente, proporcionando la capacidad de evolución al nivel de Gestión de Modelos.

Este nivel posee un mecanismo de reificación que permite agrupar todos los nodos del nivel de MetaModelo que utilizan *issi:MetaNode* o *issi:MetaProperty* como tipo. Así pues el nodo *issi:MetaNodePopulation* representa la población de nodos del Nivel de Metamodelo mediante el predicado *issi:metaNodeInstance*. Cuando se añade un nodo en el nivel inferior se crea un predicado *issi:metaNodeInstance* que parte de *issi:MetaNodePopulation* y que apunta al nodo recién creado en el nivel inferior (no aparece representado en la figura 3). La población de proposiciones construidas en el Nivel de Metamodelo es recogida por el nodo *issi:MetaNodePopulation*. Cuando se

crea una proposición mediante un predicado diferente a *rdf:type* o *rdf:subProperty* se inserta un nodo de tipo *rdfs:Statement* y se establece como propiedad de *issi:MetaPropertyPopulation* mediante el predicado *issi:metaPropertyInstance*. El nodo creado posee tres propiedades a través de los predicados *rdfs:subject, rdfs:predicate* y *rdfs:object* que hacen referencia al sujeto, predicado y objeto de la proposición recién creada en el nivel inferior.

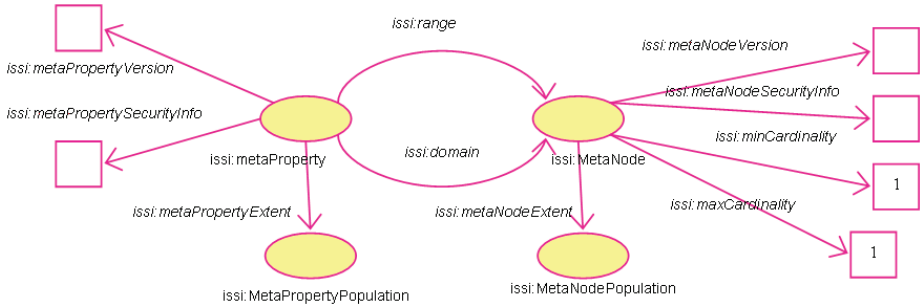


Fig. 3. Esquema conceptual del Nivel de Gestión de Modelos

El esquema conceptual de este nivel se crea mediante los contructores descritos en el Nivel Base. A continuación, se presenta una secuencia de términos que respeta el orden canónico descrito en el apartado anterior y que describe parte del esquema conceptual del Nivel de Gestión de Modelos: *add(issi:MetaNode, rdf:type, rdfs:Class); add(issi:MetaProperty, rdf:type, rdfs:Class); add(rdfs:domain, rdf:type, rdf:Property)*; *add(issi:MetaProperty, rdfs:domain, issi:MetaNode)*.

3.3. Nivel de Metamodelo

En este nivel se representan los metamodelos o metaontologías creados a partir del esquema conceptual definido para el Nivel de Gestión de Modelos. Los metamodelos permiten definir esquemas conceptuales en el Nivel de Modelo, describiendo una estructura que es capaz de almacenar información. De forma análoga, una metaontología permite definir ontologías en el Nivel de Modelo. Este hecho nos permite considerar las ontologías como modelos, extendiendo el campo de acción de la plataforma al mundo de las ontologías.

Como se ha indicado en apartados anteriores, la utilización de conceptos de RDF Schema en todos los niveles de la plataforma se debe a las ventajas que aporta, pero no se han utilizado todos los elementos que proporciona este vocabulario. En este nivel, es donde se puede definir todo el vocabulario propuesto por RDF Schema para utilizarlo en el Nivel de Modelo y así poder describir la información de recursos web en el Nivel de Información.

En [3] se extiende la definición de nodo y de predicado válidos para tener en cuenta los diferentes niveles de abstracción introducidos por debajo del Nivel Base en la definición de términos. En este documento también se especifica el mecanismo de creación de nodos y proposiciones en este nivel.

Las relaciones que se especifican entre dos nodos en el Nivel de Metamodelo se reflejan en el Nivel de Modelo mediante otra relación. Como una relación entre nodos

se expresa mediante predicados y un predicado no puede constituir el tipo de ningún otro predicado, en [3] se define un mecanismo alternativo para especificar propiedades a partir de predicados. De esta manera se establece la doble visión del concepto predicado según la aproximación de metaobjeto [5]: un predicado es una relación entre nodos que tiene como tipo a *rdf:Property* y a su vez constituye la descripción de una propiedad que da lugar a predicados concretos en el nivel inferior.

3.4. Nivel de Modelo

Este nivel se sitúa entre el de Metamodelo y el de Información, y en él se describen los modelos que permiten representar información en el nivel inferior. Los términos que constituyen un esquema conceptual en este nivel se construyen como instancias de los elementos del metamodelo definido en el nivel inmediatamente superior. Los mecanismos de reflexión y de reificación para este nivel son similares a los establecidos para el nivel de metamodelo.

3.5. Nivel de Información

Este nivel es el más concreto de todos los niveles y se sitúa por debajo del Nivel de Modelo. Los esquemas conceptuales que se representan en él se construyen a partir de un esquema conceptual definido en el Nivel de Modelo y expresan la información que se almacena en ellos. La información que contienen no aporta ninguna semántica para interpretar los datos, puesto que esta metainformación está representada en el Nivel de Modelo.

4. Trabajos relacionados

Existen herramientas que facilitan la gestión de modelos. Entre ellas destaca RONDO (ver [9]) que propone la representación de modelos mediante teoría de grafos y una serie de operadores de alto nivel que permiten manipular modelos y las correspondencias entre ellos. Los modelos se traducen a grafos mediante conversores específicos para cada tipo de modelo que deben tener en cuenta consideraciones sobre las operaciones de manipulación para que se puedan aplicar sobre los modelos resultantes. Para automatizar la creación de correspondencias entre modelos se propone en [8] una plataforma que permite almacenar la experiencia previa sobre comparación de esquemas. Esta propuesta se basa en la repetibilidad de este tipo de correspondencias y sugiere las correspondencias entre modelos en base al conocimiento que se ha ido adquiriendo. No obstante, en esta propuesta las funciones heurísticas juegan un gran papel y el usuario es el que debe tratar con estas correspondencias de una forma más o menos directa.

En varios trabajos se introducen las ontologías para manejar modelos de datos. En [1] se tratan los modelos como vistas locales de una misma ontología global, permitiendo la integración de modelos diferentes. En esta aproximación es importante el conocimiento previo de los modelos que van a ser especificados sobre una

ontología antes de su definición. De forma que una modificación en la ontología conlleva grandes costes de desarrollo debido a las reglas de reescritura establecidas entre la ontología y las fuentes XML que contienen la información. Por otro lado, su propuesta se centra en la explotación de metadatos semánticos mediante consultas pero no aportan ningún mecanismo para manipular los modelos ni las ontologías. Además se limita a la utilización de modelos XML con lo que se restringe su aplicación.

Nuestra aproximación integra la definición de modelos de datos con la definición de ontologías, hecho que nos permite aplicar las técnicas formales, citadas en la sección de introducción, sobre el incipiente mundo de la web semántica. Además las propiedades semiformales que proporciona el lenguaje RDF facilitan su aplicación. El uso de técnicas formales como los morfismos basados en reescritura de términos [10] aporta una aproximación automática y correcta a la hora de validar modelos y realizar correspondencias entre ellos, evitando el desarrollo de complejos algoritmos. Por otra parte, la plataforma permite definir tanto los metamodelos utilizados para gestionar modelos, como los modelos utilizados para gestionar información, además de establecer correspondencias entre elementos de un mismo nivel de abstracción. De esta manera, se ofrece la posibilidad de definir nuevos metamodelos y relacionarlos con otros existentes abriendo la herramienta a todo aquél que desee utilizarla.

5. Conclusiones y trabajo futuro

Se ha presentado una plataforma semántica que facilita la gestión de modelos. Esta plataforma está formada por cinco niveles de abstracción cuyos elementos se relacionan mediante mecanismos de reflexión y de reificación. De manera que la torre de reflexión tiene cinco niveles y en ella se colapsa reificación con tipado. La semántica se puede observar en dos aspectos: en la metainformación que proporciona un nivel a los elementos de un nivel inferior y la sintaxis RDF que permite expresar los modelos mediante términos constituidos por proposiciones de tres elementos.

La existencia de diferentes niveles de abstracción proporciona una gran versatilidad y potencia a la plataforma de gestión de modelos puesto que permite crear correspondencias entre esquemas conceptuales en diferentes niveles de abstracción. Además, las correspondencias entre los elementos de un esquema conceptual a un determinado nivel permite la generación automática de correspondencias en niveles más concretos. Este hecho se utiliza para la comparación de esquemas conceptuales en el Nivel de Modelo a partir de las correspondencias definidas entre metamodelos, generación de planes de migración entre esquemas conceptuales a Nivel de Información partiendo de las correspondencias establecidas a Nivel de Modelo, etc.

La representación de modelos mediante términos que respetan un orden canónico entre constructores permite aprovechar las ventajas de los lenguajes funcionales para procesar los modelos. De la misma forma aporta una gran modularidad que permite crear de forma automática conjuntos de términos mediante técnicas de patronaje y reglas de reescritura.

Una vez sentadas las bases de la plataforma, en los siguientes trabajos se va a dar un enfoque más aplicado a la plataforma de gestión de modelos. El primer paso será

desarrollar distintos metamodelos que nos permitan tratar la mayoría de los modelos existentes en el mercado. El orden canónico descrito sobre los términos de un modelo se establece de una forma muy próxima a la sintaxis RDF. Este hecho retrasa el proceso de validación del modelo hasta que esté completamente construido, en los niveles de Metamodelo e inferiores. Si el orden canónico se establece de una forma más cercana a un determinado metamodelo permitirá una mayor automatización en la validación de modelos. De igual forma, facilitará un proceso de creación de modelos más correcto puesto que se establece el orden entre los constructores de los conceptos básicos del metamodelo. Así pues, en cada metamodelo se debe definir el orden canónico entre sus términos. Otro trabajo inmediato es la aplicación de la plataforma de gestión de modelos a los proyectos de evolución de esquemas y de recuperación de sistemas legados citados en el apartado de introducción.

6. Bibliografía

1. Amann B., Beeri C., Fundulaki I., and Scholl M., *Ontology-Based Integration of XML Web Resources*, 1st International Semantic Web Conference (ISWC2002), Sardinia, Italia, June 9-12th, 2002.
2. Bernstein P. A., Levy A. Y., and Pottinger R. A., *A Vision for Management of Complex Models*, SIGMOD Record (ACM Special Interest Group on Management of Data), 2000.
3. Boronat A., Carsí J.A., Ramos I., *Una plataforma semántica para la getión de modelos*, Informe Técnico DSIC-II/17/03, Univ. Politécnica de Valencia, Junio 2003.
4. Brickley D., Guha R.V. (Editors), *RDF Vocabulary Description Language 1.0: RDF Schema*, World Wide Web Consortium, November 2002.
5. Carsí J.A., *OASIS como marco conceptual para la evolución del software*, Tesis doctoral, DSIC, Univ. Politécnica de Valencia, ISBN 84-699-3372-8. 1999.
6. Carsí J.A., Ramos I., Silva J., Perez J., Anaya S., *Un generador automático de planes de migración de datos*, Electronic Journal (I+D Computación) on Computer Science Research and Development of the International Congress on Computer Science Research (CIICC), The National Academy of Computer Science (Mexico), ISSN: 1665-238X, Julio 2002.
7. Hayes P. (Editor), *RDF Semantics*, World Wide Web Consortium, 12 November 2002 (work in progress). The latest version is http://www.w3.org/TR/rdf-mt/
8. Madhavan J., Bernstein P. A., Chen K., Halevy A., and Shenoy P., *Corpus-based Schema Matching*, at the Workshop on Information Integration on the Web at the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'2003), Acapulco, Mexico.
9. Melnik S., Rahm E., Bernstein P. A., Rondo: A Programming Platform for Model Management, to appear in Proc. ACM SIGMOD 2003, San Diego, June 2003.
10. Perez J., Anaya V., Cubel J.M., Domiguez F., Boronat A., Ramos I., Carsí J.A., *Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas*, SET 2002, Software Evolution Through Transformations: Towards uniform support throughout the software life-cycle, Barcelona - Spain, October 2002.
11. Pérez Benedí J., Carsí Cubel J. A., Ramos Salavert I., *On the implication of application's requirements*, DataBase Maintenance and Reengineering Workshop (DBMR), ISBN: 84-699-8920-0 pag. 3-16, IEEE International Conference of Software Maintenance, Montreal (Canada), November 2002.

Two experiences in software dynamics♦

Artur Boronat
(Polytechnic University of Valencia, Spain
aboronat@dsic.upv.es)

Jennifer Perez
(Polytechnic University of Valencia, Spain
jeperez@dsic.upv.es)

José Á. Carsí
(Polytechnic University of Valencia, Spain
pcarsi@dsic.upv.es)

Isidro Ramos
(Polytechnic University of Valencia, Spain
iramos@dsic.upv.es)

Abstract: This paper presents an outline of a formal model management framework developed in two projects, which provide breakthroughs for legacy systems recovery (RELS) and for data migration (ADAM). To recover a legacy system, we follow an algebraic approach by using algebras in order to represent the models and manipulate them. RELS also automatically generates a data migration plan that specifies a data transfer process to save all the legacy knowledge in the new recovered database. The data migration solution is also introduced as a support for the OO conceptual schemas evolution where their persistent layers are stored by means of relational databases, in the ADAM tool. Contents and structure of the data migration plans are specified using an abstract data migration language. The high abstraction level of this language allows us to be independent from the underlying DBMS technology. Our past experience in both projects has guided us towards the model management research field. We present a case study that illustrates the application of both tools.
Keywords: data reverse engineering, rewriting rules, data migration, migration patterns
Categories: D.2.7 [Software Engineering]: Maintenance, D.2.9 [Software Engineering]: Management, E.2 [Data Storage Representations], H.1.0 [Models and principles]: General, H.2.4 [Database Management]: Systems, I.1.1 [Algebraic Manipulation]: Expressions and Their Representation

1 Introduction

Information systems are dynamic by nature. One reason an information system could change is the inaccuracy of its requirements specification. This inaccuracy is usually produced by a misunderstanding between the user and the system analyst, inexperience of the analyst or the imprecise knowledge of the user. Other reasons for this variable behavior could be changes in the requirements of a software application, adaptation to new technologies or even the satisfaction of new standards. The consequence of these facts is the continuous evolution of the system from the beginning of its deployment. This evolution is necessary in order to achieve the information system that the user wants. Other factors that produce continuous changes

♦ This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

in information systems are the high level of competitiveness in the market place and their volatile business rules.
Statistics given regarding the time invested and the cost of people involved in the maintenance process are 80% of the total expense of software development [Yourdon 1996]. This fact has intensified interest in software evolution research in the past few years in order to cope with the problem and to reduce the costs.
A great deal of work has been done in this area. It has focused mainly on the automatic software development approach to improve the time and cost invested in the life cycle of the software system. Our work is based on the paradigm of automatic prototyping proposed by Balzer [Balzer 1985] (see Figure 1).

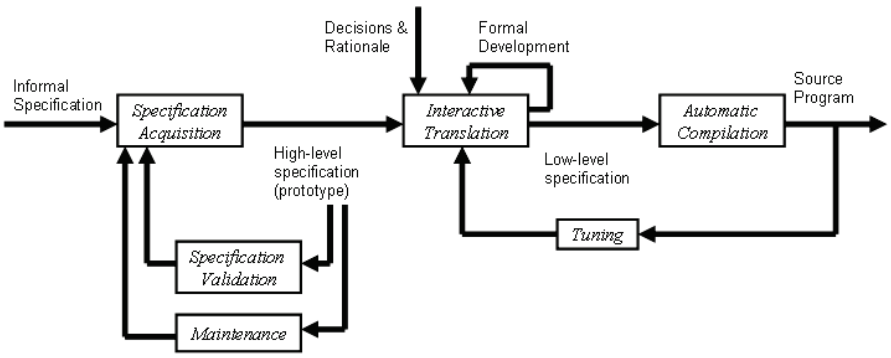


Figure 1: Paradigm of automatic prototyping

Existing CASE tools are able to generate applications from the systems specifications. These CASE tools are called model compilers. They use models to abstract a reality with some graphical notation and automatically generate the application code and the database schema from the conceptual schema of an information system. The automatic generation can be complete, such as Oblog Case ([Sernadas 1994]), or partial, such as Rational Rose ([Rational]), System Architect ([SystemArchitect]), Together ([TogetherSoft]) and others.
None of these tools provides full support for the volatile nature of an information system. Technologies used for software development become obsolete, when new technologies providing new and better features appear. However, software products are kept in use as long as they are useful and efficient, accumulating a lot of knowledge in their databases. Nevertheless, as they become obsolete they become more difficult to maintain. Systems of this kind are called legacy systems and their adaptation to new technologies or even the introduction of changes involves great economical efforts.
Another problem related to the dynamic behavior of an information system concerns to the addition of new requirements during the life cycle of the software product. Using one of the tools mentioned above, the change may be applied to the model and the new application and its database are automatically generated. Consequently, we have two conceptual schemas, the original schema and the evolved one, which contains the new features of the system. Additionally, we obtain two databases, the one corresponding to the first conceptual schema, which stores all the knowledge that

the original application has produced while it has been working, and the new, generated one, which remains empty. In this case, the problem solution focuses on finding a way to migrate the information produced by the first application to the new database.

In this paper, we present a solution for both model management problems. We explain how to solve them by means of two tools that use algebraic formalisms and pattern techniques: the legacy system recovery tool and the data migration tool.

In the next section, we present a motivating scenario involving both problems: a legacy system recovery and its evolution applying some changes. Section 3 presents the legacy system recovery tool and how it can resolve the first problem of the example shown in section 2. Section 4 presents the data migration tool and how it resolves the second problem of the problem. Section 5 discusses related works highlighting their differences with our tools. Finally, we present conclusions and future work.

2 Motivating Scenario

To promote the use of the two tools that we present in this paper, we will use a motivating scenario that is illustrated in Figure 2 and exemplifies the recovery of a legacy system and its later evolution.

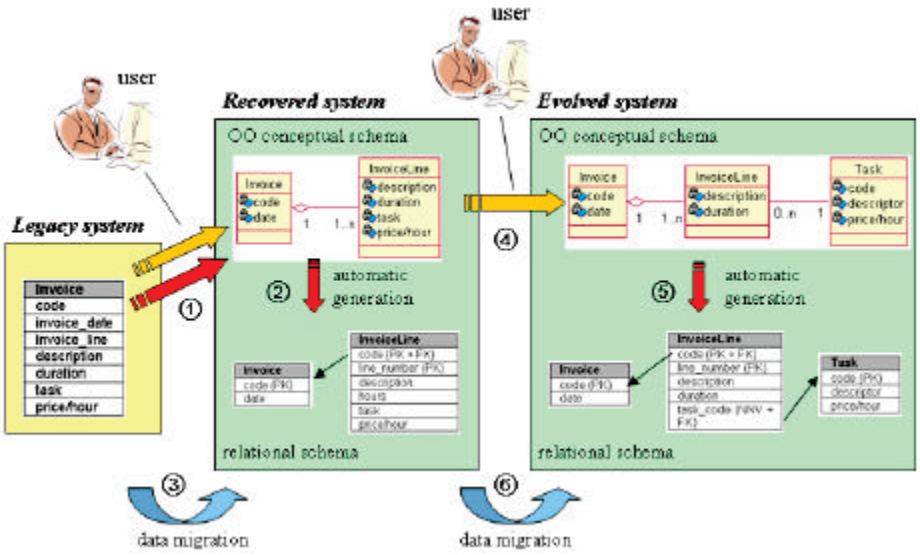


Figure 2: Motivating scenario

Consider a car maintenance company that has worked for a long time for a large car dealership. The maintenance company works with an old C application where the information is stored in a simple relational database that does not even consider integrity constraints. The car dealership has recently acquired the car maintenance company and its president has decided to migrate the old application to a new OO

technology in order to improve maintenance and efficiency. Therefore, the target application will be developed by means of an OO programming language although the database layer remains a relational database. This time, new integrity constraints are provided in the new relational database in order to improve maintainability.

Consider the part of the legacy system that stores information about invoices in the example. Each invoice contains data about the task performed in a specific period of time and at a specific price.

To recover the legacy system, a designer has to build a semantically equivalent OO conceptual schema that captures the semantics that is disorganized in the legacy system. This task is usually done manually and involves high development costs. What is worse is that the human factor does not guarantee an error-free process to obtain a correct OO conceptual schema. Step (1) constitutes a manual, reverse-engineering process where the designer detects that the legacy table can be broken down into two classes: one containing the information about a performed task during a period of time and another one representing the collection of performed tasks for a specific customer, i.e., the *InvoiceLine* class and the *Invoice* class. In this step, works like those by [Hainaut 1996], [Premerlani 1994] and [Ramanathan 1996] can be applied to obtain OO models from relational schemas.

Once the OO model is complete, the relational database has to be generated (2). Here the designer can use many CASE tools such as Rational Rose, Together or System Architect in order to generate the new relational schema automatically. Despite obtaining a relational schema, these tools do not take into account legacy data.

The cumulative experience of the maintenance company is collected in its database and, it is expected to be preserved in the new database (3). Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done by means of SQL statements or user defined scripts which can be executed on the database. Although ETL tools provide friendly interfaces to migrate data between databases, DB administrators must write migration code manually, and this is very costly in terms of people and time.

Once the OO conceptual schema has been obtained from the legacy database and its data has been migrated to the new database, a design problem is detected in the generated OO schema: information about the same tasks is repeated and appears in several instances of the *InvoiceLine* class. Then, the designer decides to change the OO conceptual schema (4) breaking the *InvoiceLine* class down into two classes: a new class *Task* representing a task with information such as the price per hour and the *InvoiceLine* class that keeps the remaining information. The change is applied to the conceptual schema and a new empty database (5) is generated by means of the same CASE tool used in step (2). The data migration problem comes up again (6). Although ETL tools can be used in the same way as in step (3), this situation differs from the first case in the sense that the data migration process has to be specified. Now the original OO conceptual schema must be compared with the evolved one in order to obtain mappings between their respective databases. Thus, the designer that has applied changes to the original OO conceptual schema must supervise the data migration process in order to provide knowledge about the system, and it becomes a very complex solution.

In the following sections, we present two tools that provide a solution for both problems using formalisms and pattern techniques. This solution consists of an automated process that backs up the designer's work in an easy and efficient manner.

3 RELS: Reverse Engineering of Legacy Systems

Legacy systems can be defined informally as "software we do not know what to do with, but it is still performing a useful job" [Ward 1995]. They are information systems that have been developed by means of methods, tools and database management systems that have become obsolete, but they are still being used due to their reliability. They are characterized by the following features:

1. Software architecture based on obsolete technology that has probably been patched in order to adapt to new changes in requirements. This fact complicates the maintenance of the application.
2. Poor, complex documentation that prevents effective maintenance or software updating, making it necessary to check the source code to understand the functionality of the system.
3. Cumulative experience working with the system that has filled its database with information that is significant for the company.

As in all complex systems with a medium life cycle, the requirements for this kind of applications go on changing at the same rate as technology does. There are two main approaches to performing changes in these systems. On the one hand, there is the patching of the legacy system that has obsolete technology code. The disadvantages to this approach are that the technology does not consider new features to improve either code reuse, quality or documentation generation, and that the staff that will develop the new part of the system needs to be trained. On the other hand, the whole system can be developed with a new technology taking advantage of all its features. Both approaches imply a high cost, but we prefer the second option because the first one only temporarily delays the translation into a new technology, making maintenance harder and harder each time the system is changed.

The tool RELS (Reverse Engineering of Legacy Systems) provides a solution to this problem by applying the second approach to the structure of an application. It uses a reengineering process to rebuild the legacy system into a semantically equivalent one with a new technology. This process is composed of two steps:

1. A data reverse engineering process that extracts an abstract description from the legacy system database in order to know its structure and its behavior. Changes can be applied to it in order to adapt the systems to new requirements or to new technologies, such as the change between the structured paradigm and the OO paradigm Our tool recovers a legacy database obtaining the static component of an equivalent OO conceptual schema using formal methods.
2. A forward engineering process that generates the software application (its structure in our case) based on a specific technology from the abstract description extracted from the legacy system. We use the Rose Data Modeler add-in [Boggs 2002] of the Rational Rose tool case to generate a new relational schema from the OO conceptual schema.

Our tool also allows for data migration from the legacy database to the recently generated one, keeping the knowledge stored in the old database. The data reverse engineering process and the data migration process followed by our approach will reduce the time invested and the number of people involved in the data evolution process. This optimization is reached by the automatic tasks that are performed by the tool in three phases. Despite the fact that these tasks are performed automatically, the results can be freely modified by the analyst. In this case, the process is semi-automatic. The tool performs the following three phases in order to reach this goal, as shown in Figure 3:

1. A UML conceptual schema is obtained by applying a data reverse engineering process in order to recover a relational legacy database. Both relational and UML conceptual schemas are represented as terms of an algebra and the correspondences between terms are specified using term rewriting rules.
2. The rewriting rules applied in the first phase and the patterns used by the Rose Data Modeler add-in to generate the new relational schema are used to describe a data migration plan which is specified using a declarative language.
3. The data migration plan is compiled into DTS[1] packages whose execution automatically migrates data from the legacy database to the new one.

In this section, we present the three phases in more detail, illustrating their application by means of the example of the motivating scenario.

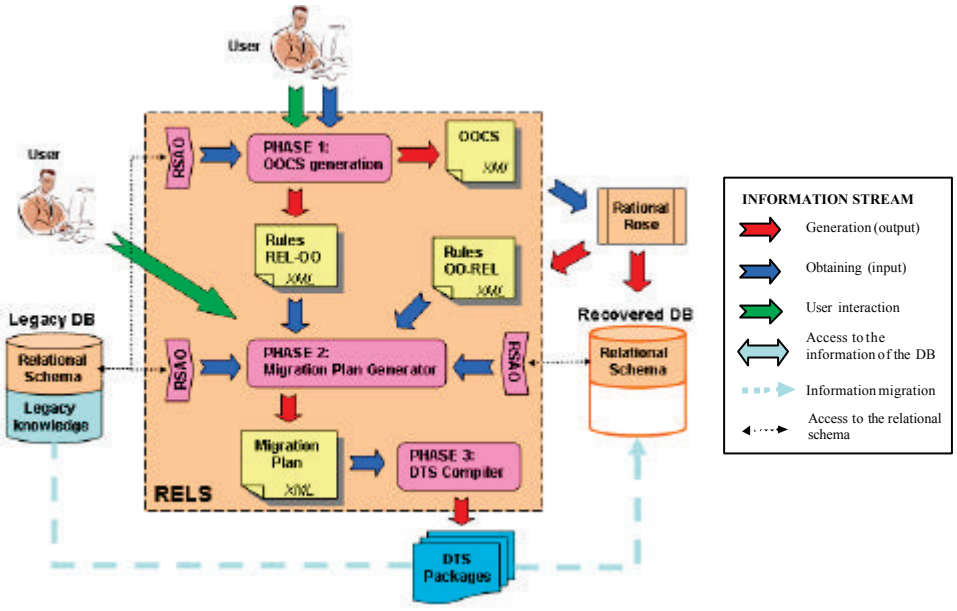


Figure 3: Legacy database recovery process in the RELS tool

[1] Data Transformation Services (DTS) is a technology of SQL Server that allows the transfer of data among heterogeneous databases.

3.1 Data Reverse Engineering Phase

The data reverse engineering phase takes the relational model of the legacy database as input and generates an object-oriented model which is equivalent to the previous relational one. These models are represented as terms of ADTs (Abstract Data Types) that are related by means of rewriting rules. These rules are applied to the term that represents a relational schema by a TRS (Terms Rewriting System) which obtains an OO conceptual schema. The following sections describe the term rewriting mechanism that translates a conceptual schema from the relational model to the OO model, and the whole process that constitutes this phase.

3.1.2 Term Rewriting Mechanism

An ADT is composed by a group of specification modules. Each one of them provides a set of operations (constructors and functions) to define terms and axioms to establish relations among these terms, such as order properties. These axioms specify how an algebraic term is correctly defined.
Therefore, we define two ADT in order to represent both relational and OO conceptual schemas:

a) A relational conceptual schema is represented as an algebraic term that is based on the syntactic and semantic rules provided by the relational ADT. This ADT consists of several specification modules, each of which is related to a relevant element of the relational model. The relational model specification module (m-rel in Table 1) is the core module that provides rules to define a relational conceptual schema term. In this ADT, a constructor is used to define an element of the relational model as a term, for instance a table. Additionally, axioms are used to specify the natural composition order between elements of the relational model; for instance, tables are formed by columns, indicating a compositional relation between the specification modules of the relational ADT.

```
SPEC   m-rel
USING  table + column + data_type + nut_nul + ...
SORTS  m-rel
OPS    create_database: --> m-rel
       add_table: table m-rel --> m-rel
       add_column: column data_type not_null table m-rel --> m-rel
       add_ctr_pk: col_list table m-rel --> m-rel ...
ENDSPEC
```

Table 1: Relational specification module

b) The OO model specification module (m-oo in Table 2) is the core module of the OO ADT. It is composed of other modules that represent elements of the OO model, such as a class or an aggregation, and it expresses how to

generate terms that combine the rules of its components. Thus, it provides the rules to generate OO conceptual schema terms.

```
SPEC   m-oo
USING  class + aggregation + specialization + ...
SORTS  m-oo
OPS    create_schema: --> m-oo
       add_class: class m-oo --> m-oo
       add_identif: attr_list class m-oo --> m-oo
       add_ctt_att: attribute data_type bool class m-oo --> m-oo
       add_vbl_att: attribute data_type bool class m-oo --> m-oo
       add_aggregation: aggregation class class rol rol nat nat
                        nat nat bool bool bool bool m-oo --> m-oo ...
ENDSPEC
```

Table 2: OO specification module

Once we can define conceptual schemas as terms of both relational and OO ADTs, we relate each element of the relational ADT with different elements of the OO ADT that are semantically equivalent.

An ADT consists of specification modules, and a module can be formed by other modules, as we have seen before in the cases of m-rel and m-oo modules. Hence, a new ADT is defined to relate elements of both relational and OO ADTs. The rules of the m-rel-oo relate the operators of the specific ADTs so that a term of the relational ADT is translated into a term of the OO ADT. These rewriting rules, which represent the correspondences between elements of the relational model and elements of the OO model, are automatically applied by a term rewriting system (TRS). Our TRS is finite and non-confluent, because we can obtain several OO terms from a relational term, i.e. several possible representations.

When the TRS applies the rewriting rules of the m-rel-oo specification module to a relational term, subterms of both relational and OO ADTs coexist in the intermediate terms that belong to m-rel-oo ADT. At the end of the rewriting process, the entire term belongs to the OO ADT. For further work, see [Pérez 2002c].

The rewriting process is automatic but the user can validate whether the applied rules are the most suitable, because an element of the relational model could be represented by several elements of the OO model. The tool supports decisions of this kind by providing the user with a set of potential rewriting rules that are syntactically correct in each rewriting step so that the user can choose one. In order to reduce the interactions required by the user, we have taken into account the criterion that legacy databases were usually designed to improve the efficiency of access to the data.

3.1.2 Data Reverse Engineering Process

The input of this phase is a relational schema of a legacy database and is the input to the main tool. This phase generates two XML documents as output: one representing the generated OO conceptual schema in XMI format ([OMG]) and another one that

contains the rewriting rules applied to obtain the final OO conceptual schema. In this phase the process that produces these outputs follows three steps:

a) The reading of the relational schema of the legacy database. The access to the relational schema is performed by means of an API, named RSAO in Figure 3, in order to access heterogeneous databases. This phase builds a term of the defined relational ADT that represents the relational schema obtained from the legacy database. It also considers features of the old DBMS or other repository forms which do not allow for the definition of constraints (either integrity or reference constraints). These constraints, implemented by code in the best case scenario, are not explicitly defined in the legacy database structure. Thus, user interaction may be necessary to provide additional information to obtain a complete relational conceptual schema. This extra information is added to the relational term obtained from the relational database by means of a graphical interface that hides the algebraic formalisms to the user.
b) Translation of the relational term into an OO term by means of the rewriting rules described in the previous section. Here the user may decide to apply other rewriting rule than the default rules chosen by the TRS in order to generate a more semantically accurate OO term.
c) Storage of the OO term as an OO conceptual schema following the UML notation by means of the XMI format. Storing the conceptual schema with this format allows the import of the generated OO conceptual schema from most of the CASE tools that manage UML diagrams, such as Rational Rose. The applied rewriting rules in the translation process are written down to a XML document that will be used in the second phase.

3.2 Relational Migration Plan Generator

This phase generates a migration plan that specifies what information must be copied from the legacy database to the database of the new OO application. Its inputs are two XML documents that contain the mappings:

- between elements of the legacy relational schema and elements of the recently generated OO conceptual schema.
- between elements of this OO conceptual schema and elements of the new relational schema generated by the Rose Data Modeler add-in.

The migration plan generator applies a set of patterns to the input correspondences and produces a migration plan that is specified using a relational declarative language indicating what information has to be copied from the legacy database to the new one. The use of a declarative language provides independence from the specific DBMSs used for supporting the databases. Additionally, this phase checks the constraints of the target database in order to avoid constraint violation.

3.2.1. Relational Migration Plan

A relational migration plan specifies the actions that must be performed in order to copy data from the legacy database to the new database, generated from the recovered

OO conceptual schema. The migration plan consists of a set of *migration modules*. There exists one migration module for each specific table of the target database. Therefore, a migration module assigns a view over the legacy database to a target table indicating where to find the source data.
To specify the data copy process, a migration module contains a set of mappings between columns of the source view and the target table. Those mappings constitute the *migration expressions* that can be used in a migration plan and they are specified by means of the relational declarative language.
The automatic generation of the migration plan considers its structure and its contents. Thus, two kinds of patterns are used: migration patterns and migration expression patterns. Migration patterns generate the structure of the migration plan following the ordering of the tables of the new database. Migration expression patterns generate their content by means of migration expressions that represent mappings between columns of the relational tables.
The migration plan generator gets the applied rewriting rules of the reengineering process from the two input XML documents, one from the data reverse engineering phase and another one from the Rose Data Modeler. These rules provide enough information to determine how many migration modules are needed and which tables of the legacy database form the source view for each module. Thus, the generator constructs the migration modules by applying the migration patterns. Then, it applies the migration expression patterns in order to link attributes of the source view with attributes of the target table in each migration module, reflecting the generation process followed to obtain the target database. Once the migration plan is finished, the generator writes it in an XML document, ready to be compiled into a specific technology in order to perform the data migration.

3.2.2. Constraint checking

Referential and integrity constraints in the target database involve an added problem to the migration process because the legacy database is not supposed to support them. The migration plan generator checks these constraints in order to avoid errors during the data migration execution. To obtain the metainformation required about referential and integrity constraints of a database in order to generate the data migration plan, we focus on the standard SQL99 [Türker 2001]. Relational DBMS that are compliant with SQL99 provide a set of tables with each database that contains information about its relational schema. We have developed an API that accesses these tables in any DBMS by means of the OLEDB interface [Lee 2002]. This API is based on the ADOX object model [Sussman 1999] and provides a way to explore the metadata that forms a relational schema.
The migration order is a sequence in which the tables of the target database must be filled to avoid violations of any of its referential constraints. In the example in Figure 2, there is a foreign key of the *InvoiceLine* table to the Invoice table. If the migration process fills the *InvoiceLine* table first, the underlying referential constraint to the foreign key would be violated for all the copied tuples. The correct migration order for these tables is then the *Invoice* table first and the *InvoiceLine* table afterwards.
The generator of the migration plan considers a correct migration order by analyzing the relational schema of the database as if it were a directed pseudo-graph in which

the tables constitute the nodes and the foreign keys become the arcs. In this step, the generator considers foreign keys to the same table (loops), several foreign keys between two tables (parallel arcs) and cycles among several tables. This order becomes the sequence in which the modules of the migration plan must be performed (each module affects a table of the target database).
Additionally, the migration plan generator also considers the integrity constraints of the legacy and the target database, because, in the first phase, the analyst might complete the relational schema manually. Thus, the generated schema might contain some constraints that are not considered in the legacy database. A simple not null constraint over a column that is added to the new database can provoke an error if there is a tuple that has a null value for its source column in the legacy data.
The generator compares the relational schema of the source view and the target table for each migration module. When the tool detects an inconsistency, it proposes several solutions to the user, such as population filters, default values and data transformations.
Therefore, we obtain a migration plan that is not based on any specific DBMS but rather considers the constraints of the target database in order to perform a secure data migration.

3.3 Migration Plan compiler

This phase performs the compilation of the migration plan to a specific technology and its execution in order to carry out the physical data migration. Each DBMS has its own ETL tool that allows for data migration among databases. We use the Data Transformation Services (DTS) of Microsoft SQL Server. This tool allows data migration between heterogeneous relational DBMS by applying data transformation services in order to fulfill target database requirements. The DTS code that performs data migration is structured in DTS tasks. These tasks perform several actions such as data copy between two tables or the execution of SQL commands and the data connection to other databases. These tasks are stored in DTS packages that become the execution units that guide the data migration process.
The compiler receives an XML document that describes the migration plan from the second phase of RELS and obtains a set of DTS packages that are able to perform the specified migration plan between the legacy database and the new one. The compiler parses the migration plan, module by module, generating the structure of a DTS package. For each type of input module, there is a specific pattern that produces a set of DTS tasks. Once the structure of the final DTS packages has been built, the compiler parses the migration expressions of each migration module and generates the contents of the DTS tasks of the corresponding DTS module. These contents perform the connections to the databases and the migration process among the source tables and the target tables.
The migration plan avoids target database constraint violations by means of several solutions; one of them is to ignore inconsistent data. In this way, there may be a lot of legacy information that is not copied to the target database. The execution environment provides an option to migrate the inconsistent data to error tables that have no constraints so that the designer can query them and recover more information

by means of a wizard that applies the solutions of the second phase to these error tables.

add_ctr_pk([code, invoice_line], LegacyInvoice, add_ctr_unique([code, invoice_line], LegacyInvoice, add_column(price/hour, currency, false, LegacyInvoice, add_column(task, string, false, LegacyInvoice, add_column(duration, int, false, LegacyInvoice, add_column(description, string, false, LegacyInvoice, add_column(invoice_line, int, false, LegacyInvoice, add_column(invoice_date, date, false, LegacyInvoice, add_column(code, int, false, LegacyInvoice, add_table(LegacyInvoice, create_database())))))))))))))	add_identif(line_number, InvoiceLine, add_identif(code, Invoice, add_unique(line_number, InvoiceLine, add_unique(code, Invoice, add_vbl_att(price_hour, currency, true, add_vbl_att(task, string, true, add_vbl_att(duration, int, true, add_vbl_att(description, string, false, add_ctt_att(line_number, int, true, add_vbl_att(invoice_date, date, true, add_ctt_att(code, int, true, (add_aggregation(agg_Invoice_InvoiceLine, Invoice, InvoiceLine, 1, 1, 1, n, false, true, false, true, add_class(Invoice, add_class(InvoiceLine, create_schema())))))))))))))))
a)	b)

Table 3.a): Relational term that represents the Invoice table of the legacy database.
Table 3.b): OO term that represents the two aggregated classes of the new OO conceptual schema.

3.4 Example

In the example of the motivating scenario, we start from a legacy system whose database consists of a table without any integrity constraint. RELS reads the legacy relational schema, and the designer adds metadata to the relational schema providing information about some integrity constraints. After that the tool generates the relational term that appears in Table 3.a representing the table of the legacy database plus information about the constraints. The rewriting process obtains the OO term that appears in Table 3.b. In this process, the designer has participated because the default rule that applies to a unique table generates a unique class. In this way, the user has chosen one of the rules proposed by the tool generating two aggregated classes.
The OO term that represents the OO conceptual schema is written in an XMI document so that any case tool that manages UML models could open the generated one. In Table 4, we show part of the generated document where there are two classes and the aggregation that relates them. We use Rational Rose to open the generated OO conceptual schema and its Data Modeler add-in to generate the corresponding relational schema. This new relational schema is different from the legacy one because the analyst has provided information about new integrity constraints and has also participated in the rule rewriting process.
Table 5 shows the migration module that specifies the data copy to the Invoice table of the target database. In this step, the tool has detected possible integrity constraint violations due to the not null value constraint over the columns of the table.

```
<UML:Association xmi.id='G.1' name='Invoice_InvoiceLine' visibility='public' isSpecification='false'
isRoot='false' isLeaf='false' isAbstract='false'>
    <UML:Association.connection>
        <UML:AssociationEnd xmi.id='G.2' name=''
        visibility='public' isSpecification='false' isNavigable='true' ordering='unordered'
        aggregation='none' targetScope='instance' changeability='changeable'
        type='S.363.1034.45.4'>
                ...
        </UML:AssociationEnd>
        <UML:AssociationEnd xmi.id='G.3' name='' visibility='public' isSpecification='false'
        isNavigable='true' ordering='unordered' aggregation='aggregate' targetScope='instance'
        changeability='changeable' type='S.363.1034.45.1'>
                ...
        </UML:AssociationEnd>
    </UML:Association.connection>
</UML:Association>
<UML:Class xmi.id='S.363.1034.45.1' name='Invoice' visibility='public' isSpecification='false'
isRoot='true' isLeaf='true' isAbstract='false' isActive='false' namespace='G.0'>
        ...
</UML:Class>
<UML:Class xmi.id='S.363.1034.45.4' name='InvoiceLine' visibility='public' isSpecification='false'
isRoot='true' isLeaf='true' isAbstract='false' isActive='false' namespace='G.0'>
        ...
</UML:Class>
```

Table 4. XMI document that contains information about the generated OO conceptual schema.

Finally, the migration plan is compiled into DTS code by means of the third phase of the tool. Depending on the user's choices the compiler generates one package containing all the migration modules of the plan (compiled mode) or two DTS packages (step-by-step mode). These packages can be opened from the same SQL Server DTS tool. If the migration plan is compiled by means of the step-by-step mode, the DTS packages can be executed one by one, following the migration order specified in the migration plan. The graphical interface of the tool allows querying the data in order to check the migrated information, and it provides several wizards to recover inconsistent data from the error tables generated during the migration process.

4 ADAM: Automatic DAta Migration

Nowadays, CASE tools can evolve applications by modifying their conceptual schema and regenerating the code and the database schema from the modified conceptual schema (see Figure 2). However, these model compilers do not take into account the data stored in the database during the evolution process.

When an information system undergoes an evolution, its conceptual schema is updated and a new schema results. A model compiler generates a new code and a new empty database from the new schema. The structure and the properties of the new database could be different from the old database. As the data remains compliant to the old database schema, the designer must preserve the data of the company by correctly migrating it in order to satisfy the properties of the new database.

The migration task is necessary and is normally done by hand. This task considerably increases the maintenance cost of a software product. For this reason, an important issue is the improvement of the database maintenance process.

```
<migration_plan>
    ...
    <target_table operation="insert">
        <target_name operation="empty">INVOICE</name _destino>
        <target_pk> <pk_field>CODE</pk_field></target_pk>
        <source_table>
            <source_field>LEGACY_INVOICE</source_field>
            <source_pk><pk_field>CODE</pk_field></source_pk>
            <target_field operation="insert">
                <target_name operation="empty">CODE</target_name>
                <source_field>
                    <source_field>LEGACY_INVOICE.CODE</source_field>
                    <condition>UNIQUE</condition>
                    <condition>NOT_NULL_VALUE</condition>
                </source_field>
            </target_field>
            <target_field operation="insert">
                <target_name operation="empty">date</target_name>
                <source_field>
                    <source_field>LEGACY_INVOICE.DATE</source_field>
                </source_field>
            </target_field>
        </source_table>
    </target_table>
    ...
<migration_plan>
```

Table 5. Fragment of the relational migration plan that specifies the data migration to the table Invoice of the target database.

Our proposal is the ADAM tool. The starting point was the work done by Carsí about OASIS reflection [Carsí 1999]. OASIS is a formal language to define conceptual models of object-oriented information systems [Letelier 1998], and it was extended in Carsí's work to support the evolution of models. As a result, the AFTER tool [Carsí 1998] was developed. This is a CASE tool prototype which is based on the logic formalism *Transaction-Frame Logic* [Kifer 1995] and allows the definition, validation and evolution of OASIS models. As the data model of OASIS and UML are basically the same, the solution applied to OASIS models can be applied to UML models.

In ADAM, the data migration process transfers and updates information system data from the old database to the new one. It improves the data maintenance process due to the automatic tasks that constitute the three steps of its migration process (see Figure 4). Despite the fact that these steps are performed automatically, the results can be easily modified by the designer. In this last case, the process will be semi-automatic. These three steps are described in the following subsections.

4.1 Matching between Conceptual Schemas

This phase of the data migration process is necessary to be able to discover the changes that have occurred in the old conceptual schema in order to obtain the new one. The changes can be obtained applying a *matching algorithm* whose result is the set of correspondences between the old and new conceptual schemas. The matching is done automatically [Silva 2002a].

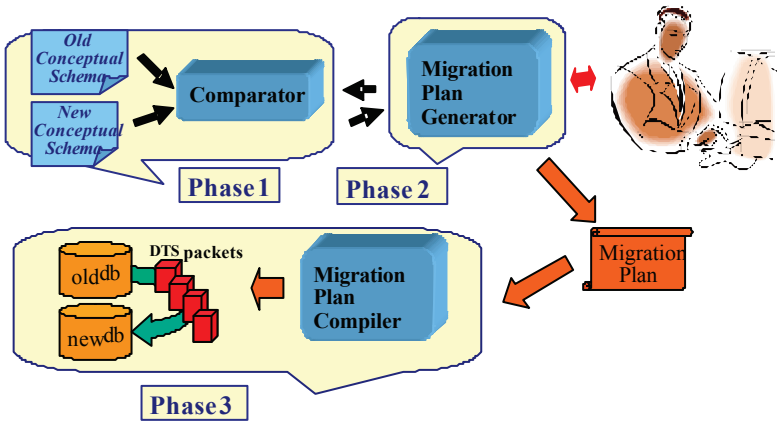


Figure 4. Data migration process of the ADAM tool

A great effort was done to study which kind of algorithms were most appropriate for this kind of problem and finally, a comparison algorithm for the ADAM tool was designed. This algorithm obtains the correspondences between both schemas and shows whether an element is a new one or a modification of an old one. The matching algorithmis based on dynamic programming techniques and graph theory.

In order to apply the algorithm, ADAM represents conceptual schemas as trees breaking cycles in relationships between their elements (see Figure 5). These elements are classified into the following categories: classes, aggregation and association relationships, specialization relationships and attributes. This categorization of the matching space reduces the complexity, thereby reducing the processing time of the algorithm.

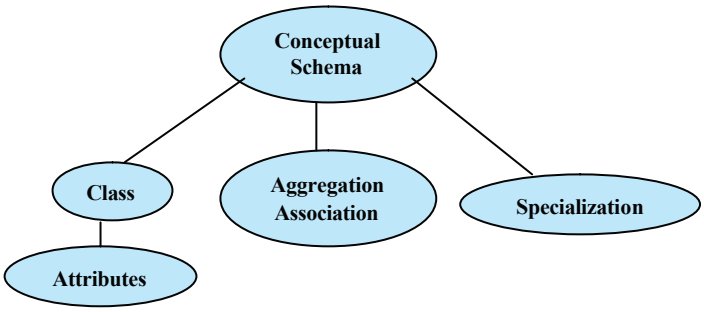


Figure 5. Tree representation of a conceptual schema

The algorithm can also use different *matching criteria* and different combinations of them [Silva 2002b]. The matching criterion allows us to distinguish whether two elements of two different conceptual schemas come from each other. Nowadays, the matching criteria that the algorithm applies are: the object identifier (OID), the name of the element, the number of attributes, the creation date, etc. Depending on the matching criterion selected by the user or the combination of them, the result of the algorithm is different. For this reason, the knowledge of the user about the system helps to select the most convenient criterion.
The matching of a sample of conceptual schemas provided by our industrial partners allowed us to validate the algorithm, achieving a high rate of correct matchings.
As a result of this phase, ADAM produces the correspondences between the elements of the conceptual schemas that have been compared. For instance, in Table 6, we show the correspondences of the algorithm using the *name matching criterion* for the classes of the example, presented in section 2.

OLD CONCEPTUAL	NEW CONCEPTUAL
Invoice	Invoice
InvoiceLine	InvoiceLine
Null	Task

Table 6: Correspondences between classes of the recovered and the evolving systems

4.2 Generation of a Data Migration Plan

From the correspondences detected in the first phase of the migration process, the second one automatically generates the first version of a *data migration plan*. A data migration plan is a set of data changes that are specified using "A Data Migration Language" (ADML) [Pérez 2002a]. Its execution allows the transfer of data from an old database to a new database one.
A data migration plan must include all the necessary changes to perform a correct migration in the right order. ADAM structures its data migration plans as a set of the following different elements: migration expressions, changes and modules; where migration modules have higher granularity than module expressions. The definition of these elements is as follows:

- *Migration expressions:* Migration expressions are those expressions that can be specified in a data migration plan. Each type of migration expression has different semantics and follows different syntactic constraints. Examples of migration expressions are: Data Source, Transformation Function, Filters, etc
- *Changes:* A change is the set of migration expressions that specify the updates undergone and the filters applied on the old database instances.
- *Migration modules:* A migration module contains the set of transformations applied to obtain a target element; it has a transactional behavior when it is executed by the ADAM tool. The composition of modules forms a data migration plan. Finally, it is important to note that there is one migration module for each class, aggregation and specialization of the new conceptual schema.

Each one of the changes that can be undergone in the old database is specified using a declarative language. This migration language follows the object-oriented metaphor. The main advantage of ADML is the independence from any DBMS due to its high abstraction level. For this reason, ADAM allows the expression of a migration plan in an easy and user friendly way and it does not need to take into account implementation details.
ADML is declarative and is used to specify all expressions that make up a data migration plan. Object-oriented conceptual schema elements are the data that are managed by the migration language. This language follows the object-oriented model. ADML makes use of path expressions to specify:

- The changes undergone in a conceptual schema element.
- The data belonging to the old conceptual schema.
- The filters that will be applied on the data, if necessary.

4.2.1 Inputs

The ADAM migration plan generation requires several data sources to automatically create the structure and the contents of the plan. The inputs of this process are the following:

- *Correspondences between conceptual schemas* produced by the first phase of the tool.
- *Properties of the elements of the conceptual schemas.* They are necessary in order to know the changes between the elements of a matching and to generate the implied transformations. These transformations must be applied on the data of the old element in order to be compliant with the new element. This information is in the conceptual schemas, where the properties of each element are defined. In our example, we are going to focus on the following correspondence:
  The *Price/hour* attribute of the *Task* class of the new conceptual schema obtains its data from the *Price/hour* attribute of the *InvoiceLine* class of the old conceptual schema. The properties of both attributes are included in the specification of the classes (see Figure 6).
  Figure 5 shows that the prices of the *InvoiceLine* class were in pesetas, and that they must be in euros in the new *Task* class. As a result, the data type of the *price/hour* attribute was *integer* in the old schema and is *double* in the new one. The *price/hour* value must also be converted to the equivalent in euros.

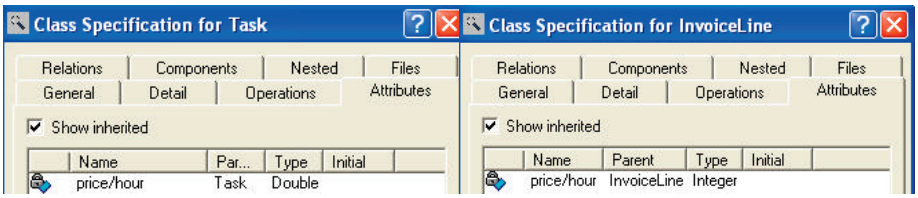


Figure 6. Price/Hour attribute of Task and InvoiceLine classes

– *Migration order*: The migration order is the sequence in which the data should be migrated. This order preserves the database in consistent states during the migration process. Moreover, this order facilitates the combination of migration modules as well as the migration order between non-related modules. The computation of this order is made by means of an algorithm. The migration order algorithm analyses the structure of the new conceptual schema to obtain the relationships between elements. These relationships imply dependencies determining the order to be followed in the data migration process. The migration order obtained by the algorithm for the new conceptual schema of our example is the following: *Invoice, Task, InvoiceLine.*

4.2.2 Process

The automatic generation of a data migration plan implies generating its structure and its contents. First of all, ADAM generates the structure creating an empty migration module for each element of the new conceptual schema and includes the modules in the data migration plan in the computed migration order. Next, the module content is automatically generated providing the migration expressions of each migration module and including them into their modules. Finally, a complete migration plan results. This automatic and complete generation of the migration plan is performed using two types of patterns: migration patterns and migration expression patterns [Pérez 2002b]. We have specified them using the patterns design criteria proposed by [Alexander 1979] and [Gamma 1994]. They are available in a pattern catalog, where they can be identified by a pattern number (**P-number**) and a title. Each pattern is composed of several sections that give different qualities of the pattern.

– *Migration expression patterns*

  There is a pattern for each one of the element properties that can be changed by the schema evolution process and for any of their possible combinations. Each pattern produces a migration expression or a set of migration expressions specifying the correct transformation of data.

  The generation of the migration expressions for a new element consists of determining which old element is related to it through mapping and consulting their different properties. Next, it applies the instantiated specific element pattern that specifies the migration expression code for the updated properties, and the resulting migration expressions are generated. Finally, these expressions are included in the new element module.

  When the data migration plan is executed, the generated migration expressions of an element will be evaluated and the instances migrated to the new database. An example of a migration expression pattern is the one for an attribute when the "name" and the "data type" properties change **(P-08)** (see Table 7).

P-08: Pattern for an attribute when the "name"and the "data type" properties change.

Solution

The solution presents the generic migration expressions that specify the attribute changes of "name", "data type" and "not null value" properties. In this case, as in the **P-04**[2] and **P-08**[3] patterns, it is necessary to perform a type conversion in the transformation function as follows:

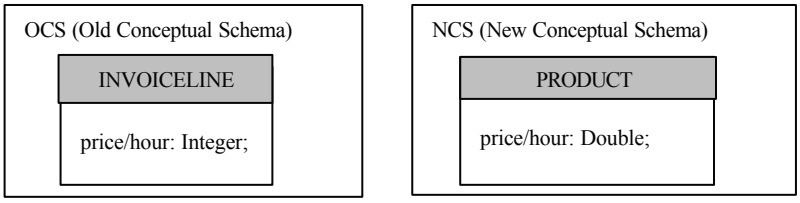
old_data_typeTOnew_data_type (old_attribute)

This pattern is a composition of the "name" and the "data type" property patterns (**P-03**[4] and **P-04**). The migrations expressions that express these changes are the following:

Transformation_Function: generic_func'('IDENT_clase'.'<IDENT_attr> ')'

Example

The prices of the products were in pesetas, and now they must be in euros. As a result, the data type of the price/hour attribute was *integer* in the old schema and is *double* in the new schema and the price/hour value must be converted to €.



Text Format

Transformation_Function: IntToDouble(OldCS.InvoiceLine.price/hour)

XML Format

<Transformation_Function> IntToDouble(OldCS.InvoiceLine.price/hour)
</Transformation_Function>

Table.7: Solution and Example sections of the migration expression pattern P-08

In our example, ADAM uses the necessary patterns for each one of the established correspondences between the attributes of the new *Task* class and the old *InvoiceLine* class. Moreover, we need take into account that the transformation function generated by the pattern (IntToDouble(OldCS.Product.UnitPrice)) must be modified by the user in order to add the currency conversion function[5]. As a result, the transformation function that will be included in the data migration plan is PtsToEuro(IntToDouble(OldCS.Product.UnitPrice)).

– *Migration patterns*

[2] **P-04:** Pattern for an attribute when the "data type" property change.

[3] **P-08:** Pattern for an attribute when the "name" and the "data type" properties change.

[4] **P-03:** Pattern for an attribute when the "name" property change.

[5] The PtsToEuros conversion finction is included in the ADAM set of built-in transformation functions.

For each type of target conceptual schema elements, migration patterns establish the way of migrating data. They also establish the necessary actions to migrate each type of conceptual schema element and the allowed migration expressions for each one. During the design process of a migration pattern, we must take into account the type of the conceptual schema element, because the transformations that may undergo each element are different.

A type of a conceptual schema element can have different associate patterns because there are different properties that influence the migration process. For example, the migration of a specialization relationship is different if its condition is more restrictive or less restrictive than the previous one, or is different because we must apply different types of filters on the data and different migration expressions in a different place. An example of a migration pattern is the pattern of the elemental class **(P-01)** (see Table 8):

P-01.Pattern: Elemental class

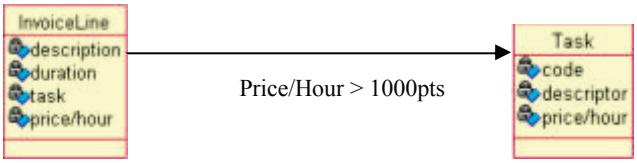
Solution

Let S be a set of schemas, C be an alphabet of classes, A be a set of attributes, G be a set of filters that are applied on old class population, GC be a set of conditions that are applied over old attributes, F be a set of transformation functions, SM be a set of matches between conceptual schemas, CM be a set of matches between classes of new and old conceptual schemas, and AM be a set of matches between attributes of conceptual schemas.

$S1, S2 \in S \wedge S1.C1, S2.C2 \in C \wedge S1.C1.a1, S2.C2.a2 \in A \wedge f_1, .., f_n \in F \wedge g_1,...,g_n \in G \wedge gc_1,...,gc_n \in GC \wedge SM1 \in SM \wedge CM1 \in CM \wedge AM1 \in CA \wedge SM1.old=S1 \wedge SM1.new=S2 \wedge CM1.old=S1.C1 \wedge CM1.new=S2.C2 \wedge AM1.old=S1.C1.a1 \wedge AM1.new=S2.C2.a2 \rightarrow data(S2.C2) = \{y \mid \exists x \in data(S1.C1) \wedge \forall i\ ?_x\ g_i\ i=1,..,n \wedge ((y.a2 = f_n of_{n-1}...of_1(x.a1)\ v\ y.a2 = cte) \wedge \forall i\ ?_{x\downarrow a1}\ gc_i)\}$

Example

The Task class of the new conceptual schema obtains its data from the InvoiceLine class of the old conceptual schema. However, the analyst of this system is only interested in the products that have a price that is higher than 1000. Moreover, all its attributes must be migrated with their transformations and conditions.



Text Format:
S1, S2 ∈ S ∧ S1.InvoiceLine, S2.Task ∈ C ∧ S1.InvoiceLine.task, S1.InvoiceLine.price/hour, S2.Task.code, S2.Task.descriptor, S2.Task.price/hour ∈ A ∧ IntTODouble, RightTrunc, PtsToEuro ∈ F ∧ {S1.InvoiceLine.price/hour > 1000pts} ∈ G ∧ SM1 ∈ SM ∧ CM1 ∈ CM ∧ AM1, AM2, AM3.AM4 ∈ CA ∧ SM1.old=S1 ∧ SM1.new=S2 ∧ CM1.old=S1.InvoiceLine ∧ CM1.new=S2.Task ∧ AM1.old=S1.InvoiceLine.task ∧ AM1.new=S2.Task.code ∧ AM2.old=S1.InvoiceLine.price/hour ∧ AM2.new=S2.Task.price/hour ∧ AM3.new=S2.Task.descriptor → data (S2.Task) = { y | ∃x ∈ data(S1.InvoiceLine) $?_x$ (S1.InvoiceLine.price/hour > 1000 ∧ (y.code = x.task) ∧ (y.descriptor = " ") ∧ y.Price = PtsToEuro(IntToDouble(x.UnitPrice)) }

XML Format:

```
<New_Conceptual_Schema>
 <Class>
  <Name> Task </Name>
  <Origin>
   <Name> InvoicedLine </Name>
   <Filtered>
    <Filter> OldCS.CS1.InvoicedLine.price/hour > 1000
    </Filter
    <Attribute>
     <Name> code </Name>
     <OriginAttribute> OldCS.InvoiceLine.Task </OriginAttribute>
     <Transformation_Function> OldCS. InvoiceLine.Task
     </Transformation_Function>
    </Attribute>
    <Attribute>
     <Name> Descriptor </Name>
     <OriginAttribute> Null </OriginAttribute>
     <Transformation_Function> " "
     </Transformation_Function>
    </Attribute>
    <Attribute>
     <Name> price/hour </Name>
     <OriginAttribute> OldCS.InvoiceLine.price/hour </OriginAttribute>
     <Transformation_Function>
          PtsToEuro(IntToDouble(OldCS.InvoiceLine.price/hour))
     </Transformation_Function>
    </Attribute>
  </Class>
</New_ConceptualSchema>
```

Table 8: Solution and Example sections of the migration pattern P-01

The first version of the data migration plan should be validated by the user after it is generated by ADAM. In addition, users can modify the plan if they want. ADAM provides a graphical user interface in order to perform these tasks in an easy, user-friendly way. This interface shows the correspondences between elements using a tree

and the differences between them by means of textual expressions, symbols and colors (see Figure 7).

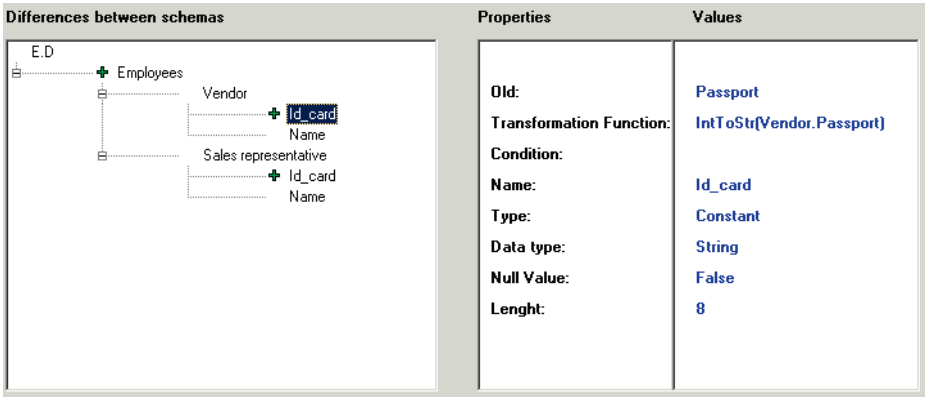


Figure 7. Graphical representation of the differences between elements

4.2.3 Outputs

After applying the necessary patterns to automatically generate the data migration plan, it is written in an XML document. This format makes the reading and translation of the data migration plan easier. This document makes the second and third phases of ADAM independent from each other.

4.3 Data Migration Plan Compiler

Finally, the third phase of the migration process compiles the data migration plan into code. The code execution migrates data from the old database to the new one [Anaya 2003]. This third phase of the ADAM tool automatically generates the code that a migration tool must manually produce using its script languages. In this way, ADAM considerably reduces the people and time invested in the creation of a migration plan between databases.

In ADAM, the target language was selected taking into account the capability of specifying complex expressions and the possibility of migrating data between heterogeneous databases such as Oracle [Oracle], Access, dBase, SQL Server, etc. SQL was excluded because it does not provide enough expressivity to specify complex expression transformations.

The compilation of the data migration plan produces a set of DTS packages. A DTS package includes a set of connections to the data sources, where data are read and stored, and a set of tasks to migrate the information. To generate the specific DTS packages that perform the data migration, we define a set of semantic correspondences between the object-oriented migration plan and the elements of a DTS package. These correspondences are shown in table 9.

Having taken into account these correspondences, the compiler reads the migration plan and the conceptual schemas to extract the necessary information to automatically generate the DTS packages. This information is stored into intermediate structures of

the main memory to be able to query them during the compilation process. The result of this process is a set of DTS packages, whose execution performs the migration of data.

Data Migration Plan	DTS Code
Migration Module	Package
Migration Sub module	Task
Data Filter	WHERE condition of task query
Transformation Function	Function specified using the script language and defined at the transformation section of a task
Attribute Condition	Condition specified using the script language and defined at the transformation section of a task

Table 9: Solution and Example sections of the migration pattern P-01

5 Experimental results

Results of both experiences in software dynamics are two tools that involve breakthroughs in legacy system recovery and in data migration. In this section, we indicate how we tested both tools.

5.1 Experimental results in the RELS tool

The RELS tool consists of several modules that are communicated by means of XML documents. This modularity has allowed us to use technologies that run on different operating systems, such as MAUDE system, which runs on Linux OS, and DTS, which runs on Windows OS. As Figure 3 shows, the RSAO API reads the metainformation that constitutes the relational schema of the legacy database and structures this information into an XML document. This document is read by the translation module (phase 1), which uses the MAUDE system and produces two more XML documents: one describing the generated OO model in XMI, and another one specifying the rules applied during the rewriting process.
The XMI document is used by Rational Rose to obtain the OO conceptual model in order to generate the relational schema of the target database. The migration plan generator module (phase 2) obtains the XML document that describes the rewriting process followed in phase 1, and obtains the generation rules applied by the Data Modeler add-in of the Rational Rose tool. This module (phase 2) generates an XML document that specifies the data migration plan, which is compiled by the DTS compiler module (phase 3), obtaining the DTS packages, whose execution performs the data migration from the legacy database to the target one.
One of the tests that we applied to the RELS tool was a free accounting application, which stored its information in a relational database. We used RELS to recover this database, obtaining an OO conceptual schema that could be edited in Rational Rose and a new relational database that contained the information of the legacy database. This process was carried out in an almost automatic manner. The user only interacted with RELS to indicate that the table could be broken down into several classes. By

doing so, the RELS tool saved us from having to use a developer team to build the new database and to migrate the information, decreasing costs in both staff and time.

5.2 Experimental results in the ADAM tool

The ADAM tool has a 3-tier architecture: client, server and database. The client layer includes the interface of the ADAM tool. The server layer implements services that provide ADAM in order to manage the data migration process. Finally, the database stores the information about schemas, the matchings between them and data migration plans.

Moreover, ADAM needs a checker of ADML migration expressions in order to syntactically and semantically validate the migration expressions defined by users. The ActiveX checker has been generated using VisualParse ++, and a file of rules has been designed.

The checker is invoked by the server layer using the function *fu_validate(string_formulae, type_formulae)*. Each time that the checker receives an invocation of *fu_validate*, it will reply to the server indicating wether the migration expression is valid and providing the decomposition of the migration expression in a XML syntactic tree. In addition, the checker needs information about the conceptual schema elements that includes the migration expression during its validation process. This information is achieved by querying the server.

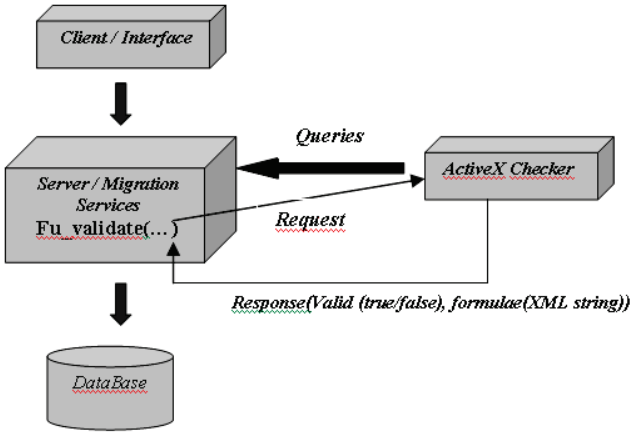


Figure 8. ADAM architecture

With regard to the server layer, it implements the three phases needed to create a data migration plan (see Figure 4). Input conceptual schemas are XMI documents that are loaded in the database of the ADAM tool. This information is used by a module that implements the comparison algorithm, and the results of the algorithm are stored in the database. Another module implements the generation of the data migration plan. This one uses the data stored in the database and its results are stored in a XML document, which follows a DTD that was designed for this purpose. The structure of these XML documents is briefly presented in the migration patterns (see table 8). Finally, the XML document that stores the data migration plan is used by the last

module that implements the server layer of the tool. This module is the one responsible for compiling this XML document into DTS packages in order to be executed. This execution allows the data migration from the old database to the new one. It is important to keep in mind that XMI documents allow us to manage any kind of conceptual schema that is able to be stored in accordance with this standard, and XML documents allows us to achieve independence among the phases of the ADAM tool.

ADAM tool was tested using several examples. These examples were provided by industrial partners and were used in order to evolve their data. As a result of these experiments, we obtained better results than our industrial partners doing these tasks by hand. Due to the privacy policies of the companies, we cannot publish these results; however, we can advance that one of our analyst who was not familiar with the information system, migrated the database using ADAM for two days and half; whereas the same database had been manually migrated by a databases expert and an expert familiar with the information system for one month.

6 Related work

The work presented in this paper involves data reverse engineering from relational schemas to OO schemas (using a relational persistent layer) and data migration between relational schemas and between relational and OO schemas.

In [Bisbal 1999], a general migration process is broken down into several phases. We compare RELS and ADAM and their application with the tools studied in this survey for each proposed phase. In [Bisbal 1999], the justification phase is when the benefits and risks of recovering a legacy system are discussed. Although there are software quality metrics for estimating the level of technical difficulty involved and there are tools like RENAISSANCE [ESPRIT] to support this task, we have not yet considered them.

An understanding process of the legacy system is necessary in order to know its functionality and how it interacts with its domain. [Müller 2000] presents a roadmap for reveerse engineering research building on the program comprehension theories of the 1980s and the reverse engineering technology of the 1990s. We focus on database understanding tools and we find the DB-Main CASE tool [DB-Main]. This tool applies a data reverse engineering process and recovers the conceptual schema from the logic schema to obtain traceability between different layers of the database, to create new databases in other DBMSs and to reduce the dependence on the technology. [Henrard 2002] describes and analyzes a serie of strategies to migrate data-intensive applications from a legacy data management system to a modern one, basing on DB-Main CASE tool. Rational Rose also obtains OO conceptual schemas from many DBMSs by means of the Data Modeler add-in. However, none of them takes into account legacy data recovery.

RELS does not only the support development of relational schemas by hiding their physical database design, but it also provides an automated translation across ontologies, i.e. the relational and the OO metamodels. In the target system development phase, which is based on three-layer target systems, we produce the persistent layer.

The testing phase ensures that the new recovered system provides the same functionality as the legacy system. This is a complex task that can be supported by a Back-to-Back testing process [Sommerville 1995]. We can shorten this task by generating a relational schema that is semantically equivalent to the legacy database schema, although we allow the introduction of new integrity constraints to the legacy relational schema that remains buried in the code of the old application.

For the migration phase, we contrast the use of our tool with the other approaches to migration presented in [Bisbal 1999]. The Big Bang approach [Bateman 1994], also referred to as the Cold Turkey Strategy [Brodie 1993], involves redeveloping a legacy system from scratch using modern software techniques and hardware of the target environment. This approach was severely criticized in [Brodie 1993]. In [Brodie 1995], Brodie and Stonebraker present their Chicken Little approach. This strategy proposes migration solutions for fully-, semi- and non-decomposable legacy systems by using a set of gateways that allow the recovery of the legacy system in an incremental way. These gateways relate the legacy and recovered databases during the migration process, so that both systems coexist during the migration process, sharing data. Nevertheless, [Wu 1997] presents the Butterfly methodology, which discredits the Chicken Little approach by arguing that the migration process maintained by means of gateways is too complex. Under the Butterfly approach, new subsystems are developed; however they are only taken into production once the whole system is finalized using the Cold Turkey approach. The last phase involves a data migration process eliminating the need for data gateways. RELS follows this last approach providing support for an automated generation of the new database by means of a formal data reverse engineering process. This reduces the fears that a legacy system migration provokes.

RELS and ADAM focuses on the data migration process taking into account heterogeneous relational DBMSs and managing inconsistencies that might be produced during the migration of the legacy data to the new database. Additionally, the high level of user involvement in these approaches is drastically reduced by means of data inconsistency wizards and automated support for schema generation. RELS focuses on relational DBMS, but it can also be applied to COBOL legacy systems whose persistent layer is based on a flat file by interpreting it as a relational table.

Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done by means of SQL statements or user-defined scripts that can be executed on the database. However, these tools do not provide automatic support for the generation of these statements and scripts as the data migration tool does. For this reason, DB administrators must write the migration code by hand.

There are several works that study new algorithms to perform data migration in a more efficient manner, such as [Anderson 2001] and [Khuller 2003]. These works focus on the physical consistency of the data persistence when the physical storage configuration must be changed, whereas we stay at a high logical level.

The most similar approach to ADAM on data migration between relational and OO schemas is the TESS tool [Staund 2000]. It involves an automatic process that is based on schema evolution. TESS uses an intermediate language that is generated from the relational schema code. This is an important difference to our approach, because we deal directly with the OO conceptual schemas, and we do not have to

translate them to an intermediate language. The OO conceptual schemas give us a higher level of abstraction and eliminate the translation process.
The Varlet Database [Jahnke 1998] provides support to transform a relational schema into an OO conceptual schema and migrates the legacy data to the new OO database. However, our approach considers a relational database as the persistence layer of an object society and migrates information to it. Also, in Varlet, the legacy relational schema is enriched with semantic information that is extracted from several sources as the application source code. In our approach, this semantic information is given by the user in an interactive way.

7 Final Remarks and Further Work

This paper presents two experiences in software evolution that provide support to legacy system recovery and data migration.
To recover a legacy system, we follow an algebraic approach by using algebra terms to represent models. RELS provides a data reverse engineering process supported by a Term Rewriting System that applies a set of rewriting rules, obtaining the term that represents the target OO model. RELS also generates a data migration plan that specifies the data copy process to keep all the legacy knowledge in the new recovered application database. This entire process should be checked by a designer who could intervene, if necessary in order to obtain a more accurate result.
The data migration problem is also introduced for the OO conceptual schemas evolution where persistent layers are formed by relational databases. In this case, a matching process is applied between both OO models to generate mappings between them that are used in the generation of the data migration plan. The automatic generation process gives us a first version of a data migration plan that can be modified later by the designer.
The contents and structure of the data migration plan are generated by means of a set of patterns. The high abstraction level of the migration language allows us to be independent from the underlying DBMS.
Our implementation experience in both projects has led us to dealing with generic model management tasks. A model is an abstract representation of reality that enables communication among heterogeneous stakeholders in order to understand each other, i.e. an OO conceptual schema, an interface definition, an XML DTD, or a semantic network.
RELS and ADAM work for several heterogeneous models by means of mappings between them that allows transformations between models of heterogeneous metamodels such as the automatic generation of OO conceptual schemas from relational schemas. Model management aims at solving problems related to model representation and its manipulation. This is done by considering models as first-class citizens that are manipulated by means of abstract operators. This approach permits the automation of model manipulation tasks. Therefore, it completely involves all the tasks carried out in our projects. In future projects, we will propose a model management platform that permits model representation and manipulation using an algebraic approach.

8 References

[Anderson 2001] Anderson E., Hall J., Hartline J., Hobbes M., Karlin A., Saia J., Swaminathan R., Wilkes J.: "An Experimental Study of Data Migration Algorithms". Workshop on Algorithm Engineering, pages 145-158, 2001.

[Alexander 1979] Christopher Alexander, "The Timeless Way of Building', Oxford University Press. 1979.

[Anaya 2003] Anaya V., Carsí J.A., Ramos I., "Automatic evolution of database data", Novática, ISSN: 0211-2124, July 2003 (in Spanish).

[Balzer 1985] Balzer R., "A 15 Year Perspective on Automatic Programming". IEEE Transactions on Software Engineering, vol.11, num.11, pages 1257-1268, November 1985.

[Bateman 1994] A. Bateman and J. Murphy, "Migration of Legacy Systems"; School of Computer Applications, Dublin City University, Working Paper CA-2894, 1994

[Bisbal 1999] Bisbal, J., Lawless, D., Wu, B., Grimson, J.: "Legacy Information Systems: Issues and Directions", IEEE Software, September/October 1999.

[Boggs 2002] Boggs, W., Boggs, M.: "Mastering UML with Rational Rose 2002"; Sybex. January 2002.

[Brodie 1993] Brodie M., Stonebraker M., "DARWIN: On the Incremental Migration of Legacy Information Systems"; Technical Report TR-022-10-92-165 GTE Labs Inc., March 1993

[Brodie 1995] Brodie, M. and Stonebraker, M., "Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach", Morgan Kaufmann, 1995.

[Carsí 1998] Carsí J.A., Camilleri S., Canós J.H., Ramos I., "Homogeneous graphical user interface to design and use information systems", JIS'98, III Workshop on Software Engineering, Murcia - Spain, November 1998.

[Carsí 1999] Carsí J.A., "OASIS as conceptual framework to treat the software evolution", PhD Thesis, Technical University of Valencia, Spain, November 1999 (in Spanish).

[Chaffin 2000] Chaffin M., Knight B., Robinson T., "Professional SQL Server 2000 DTS (Data Transformation Services)", Wrox, 2000

[DB-Main] DB-Main, http://www.fundp.ac.be/recherche/unites/publications/en/2987.html

[ESPRIT] ESPRIT Project - Lancaster University, "RENAISSANCE Project - Methods & Tools for the evolution and reengineering of legacy systems"; http://www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/RenaissanceWeb/, December 1997

[Gamma 1994] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addyson-Wesley.1994.

[Hainaut 1996] Hainaut, J. L., Henrard, J., Roland, D., Englebert, V., Hick, J.M.: "Structure Elicitation in Database Reverse Engineering"; WCRE'96 3 (1996), 131-140.

[Henrard 2002] Henrard J., Hick J.M., Thiran P., Hainaut J.L.: "Strategies for Data Reengineering"; WCRE 2002, Working Conference on Reverse Engineering October 29 - November 1, 2002. Richmond, Virginia, USA.

[Jahnke 1998] Jahnke, J.H., Zundorf, A.: "Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment". Theory and Application of Graph Transformations (TAGT'98), Paderborn, Germany, Technical Report tr-ri-98-201, University of Paderborn, 1998.

[Khuller 2003] Khuller S., Kim Y., Wan Y.: "Algorithms for Data Migration with Cloning"; SIAM Journal on Computing, Volume 33, Number 2, Pages 448 – 461, 2003.

[Kifer 1995] Kifer M., "Deductive and Object Data Languages: A Quest for Integration," Proc. of the 4th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, December 1995.

[Lee 2002] Lee W.: "The Evolution of Data-Access Technologies". SQL Server Magazine (April 2002). URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsqlmag02/html/TheEvolutionofData-AccessTechnologies.asp

[Letelier 1998] Letelier P., Sánchez P., Ramos I., Pastor O., "OASIS 3.0: A formal Approach for OO Conceptual Modelling". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721- 663-0, 1998. (In Spanish)

[Müller 2000] Müller H. A., Jahnke J. H., Smith D. B., et al.: "Reverse Engineering: A Roadmap"; In A. Finkelstein, editor, The Future of Software Engineering, ACM Press, 2000.

[OMG 2003] OMG: XML Metadata Interchange (XMI), v2.0. Object Management Group, formal/03-05-02 (2003)

[Oracle] Oracle Corporation, *Oracle*, http://www.oracle.com

[Pérez 2002a] Pérez J., Carsí J A., Ramos I., "ADML: A Language for Automatic Generation of Migration Plans", The First Eurasian Conference on Advances in Information and Communication Technology , Tehran, Iran, octubre 2002 http://www.eurasia-ict.org/ © Springer LNCS vol n.2510

[Pérez 2002b] Pérez J., Carsí J A., Ramos I., "On the implication of application's requirements changes in the persistence layer: an automatic approach", Database Maintenance and Reengineering Workshop (DBMR'2002), IEEE International Conference of Software Maintenance, Montreal (Canada), October 1st, 2002, pag. 3-16, ISBN: 84-699-8920-0.

[Pérez 2002c] Pérez, J., Ramos, I., Anaya, V., Cubel, J.M., Domínguez, F., Boronat, A., Carsí, J.A.: "Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas"; Software Evolution Through Transformations: Towards Uniform Support throughout the Software Life-Cycle Workshop (SET'02), First International Conference on Graph Transformation(ICGT2002) , Barcelona (Spain), 2002.

[Premerlani 1994] Premerlani, W.J., Blaha, M.: "An approach for reverse engineering of relational databases"; Communications of the ACM 37,5 (1994), 42-49.

[Ramanathan 1996] Ramanathan, S., Hodges, J.: "Reverse Engineering Relational Schemas to Object-Oriented Schemas"; Technical Report MSU-960701, Mississippi State University, Mississippi, 1996.

[Rational] Rational Software, "Rational Rose", http://www.rational.com/products/rose/.

[Sernadas 1994] Sernadas A., Costa J.F., Sernadas C., "Object Specifications Through Diagrams: OBLOG Approach" INESC Lisbon 1994

[Silva 2002a] Silva J.F., Carsí J.A., Ramos I., "An algorithm to compare OO-Conceptual Schemas" Proceeding of the International Conference on Software Maintenance (ICSM'2002), Montreal, Canada, 2002.

[Silva 2002b] Silva, J.F., Carsí, J.A., Ramos, I., "Theoric analyze of the criteria of OO conceptual schemas comparison", Ingeniería Informática Magazine, ISSN:0717-4197, January, http://www.inf.udec.cl/revista/edicion7/jsilva.htm (in Spanish)

[SQL] Microsoft, *SQL Server*, http://www.microsoft.com/sql

[Sommerville 1995] Sommerville, I., "Software Engineering", Addison-Wesley, 1995

[SystemArchitect] "System Architect", http://www.popkin.com/products/sa2001/systemarchitect.htm

[Staund 2000] Staund Lerner, B.: "A Model for Compound Type Changes Encountered in Schema Evolution"; ACM Transactions on Database Systems (TODS) March 2000, Vol. 25 number 1.

[Sussman 1999] Sussman, D.: "ADO 2.1 Programmer's Reference"; Wrox Press Inc; 2nd edition (June 1999).

[TogetherSoft] "TogetherSoft Corporation", http://www.togethersoft.com/

[Türker 2001] Türker, C., Gertz, M.: "Semantic Integrity Support in SQL-99 and Commercial (Object-)Relational Database Management Systems". The VLDB Journal — The International Journal on Very Large Data Bases archive. Volume 10 , Issue 4 (December 2001). Pages: 241 - 269.

[Versant] Versant Object Technology, “Versant”, http://www.versant.com/
[Ward 1995] Ward, M.P., Bennett, K.H.: “Formal Methods to Aid the Evolution of Software”; Journal of Software Maintenance: Research and Practice, Vol 7, no 3, May-June 1995, pp 203-219.
[Wu 1997] Wu, B., Lawless, D., Bisbal, J., Richardson, R., Grimson, J., Wade, V., O'Sullivan, D.: "The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems", in Proceedings of the 3rd 1EEE Conference on Engineering of Complex Computer Systems, Italy, September 1997.
[Yourdon 1996] Yourdon, E.: “Rise and Resurrection of the American Programmer”; Yourdon Press, Upper Saddle River, NJ, 318 pp., 1996.

Replace this file with `prentcsmacro.sty` for your meeting, or with `entcsmacro.sty` for your meeting. Both can be found at the ENTCS Macro Home Page.

An Algebraic Baseline for Automatic Transformations in MDA

Artur Boronat [1] José Á. Carsí [2] Isidro Ramos [3]

Information Systems and Computation Department
Polytechnic University of Valencia
Valencia, Spain

Abstract

Software evolution can be supported at two levels: models and programs. The model-based software development approach allows the application of a more abstract process of software evolution, in accordance with the OMG's MDA initiative. We describe a framework for model management, called MOMENT, that supports automatic formal model transformations in MDA. Our model transformation approach is based on the algebraic specification of models and benefits from mature term rewriting system technology to perform model transformation using rewriting logic. In this paper, we present how we apply this formal transformation mechanism between platform-independent models, such as UML models and relational schemas. Our approach enhances the integration between formal environments and industrial technologies such as .NET technology, and exploits the best features of both.

Key words: Graph-based models, MDA and model transformation, consistency and co-evolution, term rewriting systems.

1 Introduction

The development of information systems is getting increasingly complex as these systems become both more widely distributed and pervasive in influence [1]. New technologies that enable these capabilities allow for a wide range of choices which the software developer must take into account. Such choices involve technologies for describing software such as object-oriented programming languages, XML, database definition, query languages, etc. These technologies

[1] Email: aboronat@dsic.upv.es
[2] Email: pcarsi@dsic.upv.es
[3] Email: iramos@dsic.upv.es

©2004 Published by Elsevier Science B. V.

Boronat, A., Carsí, J. A., Ramos, I. An Algebraic Baseline for Automatic Transformations in MDA. Proc. of the Workshop on Software Evolution Through Transformations at ICGT2004. 2004.

have different levels of abstraction. For instance, there are technologies that support requirements engineering such as DOORS or RequisitePro. There are also technologies that support modeling approaches such as UML and others that permit the implementation of a specific solution such as .NET Framework, Java, etc. On the one hand, modeling approaches provide mechanisms to find the most effective representation of real-world concepts in the domain space of a software project. On the other hand, the majority of technical frameworks offer a large number of mechanisms to build solutions in the computer space.

In general, the initial idea was that the models had to represent reality from the user perspective, starting from the problem space. However, these models actually come from the solution space (such as UML) abstracting the features of object-oriented programming language and constraining the logic representation of the problem space. This approach was taken in order to be able to automatically generate software artifacts from a layer that is more abstract than the final code. Unfortunately, the automatic generation of software has not yet reached maturity, provoking the crisis of the CASE tools that appeared in the mid 90s. Contrary to what we expected ten years ago, we have arrived to a complicated situation where the design and implementation of a software product requires an enormous effort involving several technologies.

However, all the software artifacts that we have mentioned above can act as models with a specific level of abstraction. The generation of code that CASE tools are supposed to perform (model compilers) can also be considered as the transformation of a model of a high level of abstraction into one with a more specific level of abstraction. In accordance with this approach, a research field has emerged, providing a solution to problems of this kind: model management. A model is an abstract representation of a domain that enables communication among heterogeneous stakeholders so that they can understand each other. Model management [2] aims at solving problems that require model representation and its manipulation in an automated way.

The OMG's Model Driven Architecture (MDA) initiative [3] is set in this context and provides several proposals to define models, through the standard Meta-Object Facility (MOF) [4]. It also offers proposals to perform model transformations by means of the Query/View/Transformations (QVT) language, which is still in its early stages [5]. While a lot of attention has been given to the transformation of platform-independent models into platform-specific models, the scope of MDA goes beyond this in an attempt to model all the features of a software product throughout its life cycle. Nonetheless, a precise technique to provide formal support for the entire process of model transformation has not yet been developed.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. MOMENT is based on an algebra that is made up of *sorts* and operators that permit the representation of any model as a term that can be automatically manipulated by means of operators. The MOMENT Framework

benefits from the best features of current visual CASE tools and the main advantages of formal environments such as term rewriting systems, combining the best of both industry and research.

This paper presents the generic model transformation mechanism of the MOMENT Framework, and its application to platform-independent models, obtaining a UML model from a relational schema in the MDA context. The paper is structured as follows: Section 2 presents other approaches to provide automatic support for model transformations; Section 3 indicates an example to illustrate the transformation process through the paper; Section 4 provides an overview of the MOMENT Framework; Section 5 presents the model transformation mechanism of the MOMENT Framework, focusing on the use of a Term Rewriting System (TRS) as a formal environment to perform automatic model transformations. Finally, Section 6 summarizes the work and indicates the future directions of our research tasks.

2 State of the Art

The essentials of a model theory for generic schema management are presented in [6]. This model theory is applicable to a variety of data models such as the relational, object-oriented, and XML models, allowing model transformations by means of categorical morphisms. RONDO [2] is a tool based on this approach. It represents models by means of graph theory and a set of high level operators that manipulate such models and the mappings between them by using category theory. Models are translated into graphs by means of specific operators for each metamodel. These algebraic morphisms are implemented using imperative algorithms such as CUPID [7]. CUPID is an algorithm for matching schemas in the RONDO tool.

In the MOMENT platform, we follow the framework that is proposed in the Meta-Object Facility specification (MOF). MOF is one of the OMG family of standards for modeling distributed software architectures and systems. It defines an abstract language and a four-layer framework for specifying, constructing and managing technology neutral metamodels. A metamodel is an abstract language for different kinds of metadata. MOF defines a framework for implementing repositories that hold metadata described by the metamodels. This framework has inspired our platform for model management, although we do not use the same vocabulary to describe metamodels. In the case of MOF, the two most abstract layers offer an abstract view of a specific model. This allows for the definition of generic operators to manipulate models and metamodels.

The MétaGen project [9] has dealt with model engineering since 1991, aiming at a fully automatic generation of a conventional application from a description given by its intended user. Such a description is performed by means of PIR3, a variant of what is known in the Database community as Entity-Relationship Model. In [10], Revault et al. compared three metamod-

eling formalisms to share the experience they acquired during the MétaGen project. In that paper, they presented a way to transform MOF-based metamodels into PIR3-based metamodels so that the metamodels could benefit from the MétaGen tools. This proposal constrains the expressivity of the source metamodels because the Object-Oriented paradigm is richer than the Entity-Relationship paradigm. Our model management approach supports the definition of several metamodeling languages such as MOF or PIR3, considering them as models at the same abstraction level and using generic operators. This makes automated transformations between models of both metamodels possible, without loss of expressivity.

XML [18], the standard for data communication between applications is also used to represent models and metamodels by means of the XMI specification [19]. MOF defines a meta-metamodel, while XMI indicates the physical representation for the metamodels and the models that can be defined with it. In the model transformation field, there is a XML specification that allows the transformation of a XML document into another one, called XSLT [20]. It could be used to transform models that are represented in the XML format. Comparing XSLT to term rewriting systems, there are some differences that should be pointed out:

- Although XSLT is said to be a declarative language, control instructions, such as jumps and loops, can be used to guide the transformation process. In contrast, a term rewriting system takes over the transformation rule evaluation process.
- Writing an XSLT program is a long and painful process which implies poor readability and high maintenance cost for associated programs. Also, writing an XSLT program requires good skills in the MOF and XMI specification, because when using XPATH and XSLT, the developer must take into account the structure of models which depends on metamodels. These metamodels are in turn widely influenced by MOF and XMI. By expressing metamodels as algebras, we can deal with a more specific syntax that reflects their semantics using the algebras as domain specific languages. Therefore, writing models (and consequently transformation rules) becomes easier and more comprehensible.
- Transformation rules in XSLT are applied without taking into account the target XML schema (metamodel when transforming models), implying a posterior checking to determine whether the obtained document is a valid XML document that conforms to the target schema. Rewriting rules in an algebra take into account the source and the target algebras so that a posterior checking is no longer needed.
- Executing an XSLT program is not user-friendly for model transformation because there are no error messages to advise the user about an incorrect transformation.

The MTRANS Framework [21] provides an abstract language to define

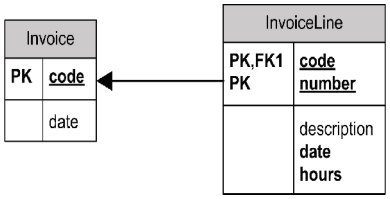


Fig. 1. Relational schema

transformation rules that are compiled to XSLT. Even though this language is more compact and easier to understand than XSLT, it still keeps instructions to manage the transformation rules evaluation. Nevertheless, transformations using XML technology imply the use of standard specifications that are industrially supported while term rewriting systems usually remain within the field of research.

3 A Running Example

Consider a car maintenance company that has worked a long time for a large car dealership. The maintenance company has always worked with an old C application where the information is stored in a simple relational database that does not take into account integrity constraints. The car dealership has recently acquired the car maintenance company and the president has decided to migrate the old application to a new OO technology in order to improve maintenance and efficiency. Therefore, the target application will be developed by means of an OO programming language.

Suppose that a part of the original database are two tables related by means of a foreign key, representing the information of an invoice and its lines, as shown in Fig. 1. To obtain a UML model that is semantically equivalent to this relational schema, a designer usually builds it manually, which involves high development costs, since the entire initial database must be taken into account. What is worse is that this process is error-prone due to the human factor.

4 The MOMENT Framework

The MOMENT (MOment manageMENT) Framework is a modular architecture divided into the three traditional layers: interface, functionality and persistence. In each one of them, the environment benefits from mature tools, such as graphical CASE tools at the interface layer, term rewriting systems at the functionality layer, and RDF repositories at the persistence layer. Hence, the MOMENT Framework aims at using the best features of each environment, bringing industrial modeling tools closer to more formal systems. Fig. 2 shows an overview of the MOMENT Framework.

The functionality layer permits the representation of models and the per-

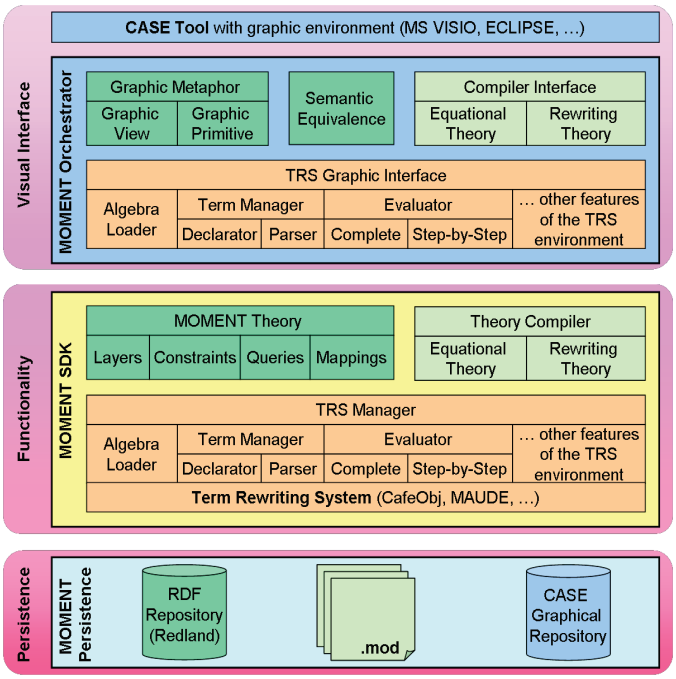


Fig. 2. The MOMENT Framework

formance of transformations over them. The core of the functionality layer is a module called *MOMENT Theory*, which allows model representation and manipulation by means of an algebraic approach. We use the expressiveness of the algebra that the platform is based on to define and represent a model as an algebraic term. This algebra represents models by means of terms of a *sort* called *Schema*. These terms are made up of by concepts and properties. The concepts are the main entities of a model, and the properties either describe them with values or establish relationships between them. The properties contain information about cardinality, indicating how many concepts can be related to the owner of the property.

The MOMENT platform uses several metadata layers to describe any kind of information including new metadata types. This architecture is based on both the classical four-layer metamodeling architecture (following standards such as ISO [11] and CDIF [12]) and on the more modern four-layer framework proposed in the MOF specification [4]. In our work, we divide the platform into four abstract layers:

- The M0-layer collects the examples of all the models, i.e., it holds the information that is described by a data model of the M1-layer.
- The M1-layer contains the metadata that describes data in the M0-layer and aggregates it by means of models. This layer provides services to collect examples of a reality in the lowest layer.
- The M2-layer contains the descriptions (meta-metadata) that define the structure and semantics of the metadata located at the M1-layer. This

layer groups meta-metadata as metamodels. A metamodel is an "abstract language" that describes different kinds of data. The M2-layer provides services to manage models in the next lower layer.

- The M3-layer is the platform core, containing services to specify any metamodel with the same common representation mechanism. It is the most abstract layer in the platform. It contains the description of the structure and the semantics of the meta-metadata, which is located at the M2-layer. This layer provides the "abstract language" to define different kinds of metadata.

The *MOMENT Theory* module also provides a mechanism to define transformations between metamodels. The *TRS Manager* module wraps a TRS, which carries out the model transformation by applying a set of rewriting rules automatically. We have used the CafeOBJ environment as TRS [13]. The *Theory Compiler* module permits the compilation of the algebraic specification of a metamodel into a theory based on equational logic. It also compiles the defined mappings between the elements of the metamodels into a theory based on rewriting logic in order to perform the model transformation on the wrapped TRS.

Some of these modules have been developed using the functional language F# [14], which provides convenient features to work with algebraic specifications and with imperative programming environments such as .NET technology. A combination of functional languages and algebraic specification languages has permitted us to reach our goals. On the one hand, the MOMENT algebra is implemented in F#, which provides efficient structures for navigation and specification manipulation. On the other hand, a TRS provides a suitable environment to support automatic model transformation.

5 PIM-to-PIM Transformation

MDA raises the level of abstraction in the software development process by treating models as primary artifacts. Models are defined using modeling languages, but when those languages are intended to be used for anything more sophisticated than drawing pictures, both their syntax and their semantics must be specified. In this case, the use of formal languages usually involves dealing with their complex syntax, making them unpopular in industry. In this sense, the MOMENT Framework is user-friendly and permits the use of formal techniques from well-known CASE tools to both define models by means of algebraic specifications and to perform model transformations using rewriting logic [15].

In this section, we present how MOMENT provides formal support for generic model transformation in the MDA context, by generating a UML model from a relational schema. First, we explain a general overview of the transformation mechanism, and later, we focus on the most formal phases of the process.

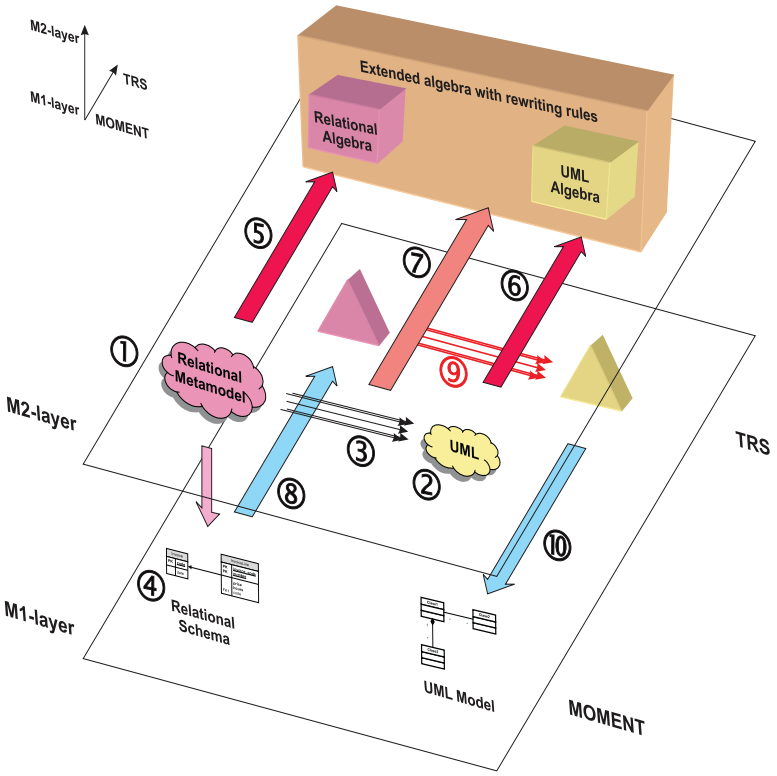


Fig. 3. Model Transformation

5.1 *Overview of the MOMENT model transformation process*

Transforming any model using the MOMENT Framework constitutes a process that is detailed in Fig. 3. To obtain the corresponding UML model from the relational schema of the motivating example, we perform the following steps:

(i) (1) and (2):
We specify both relational and UML metamodels, respectively, at the M2-layer of the MOMENT platform using the operations of the MOMENT algebra. Each one of the metamodels is a schema made up of concepts, which describe the main entities of the ontology, and by properties, which describine the concepts by specifying values and establishing relationships between them. These algebraic specifications are performed through visual wizards that are embedded in a specific CASE tool to disguise the equational logic formalism.

(ii) (3):
Mappings are specified between the concepts of both metamodels at the M2-layer by means of a script language, indicating semantic relationships. There are two kinds of equivalence mappings that can be expressed in this language:
(a) *Simple mappings*, which define a simple correspondence between two concepts that belong to different metamodels; for instance, between a table and a class, or between a column of a table and an attribute

of a class.

(b) *Complex mappings*, which define correspondences between elements of a source metamodel and a target metamodel. These mappings relate two structures of concepts that represent a similar semantic meaning. For instance, to define an equivalence relationship between a foreign key of the relational metamodel and an association of the UML metamodel, we have to relate the foreign key, the unique constraint and the not null value constraint concepts to the association concept. This is because all three of these concepts of the relational metamodel provide the necessary knowledge to define an association between two classes in the UML metamodel, such as the cardinalities of the association.

(iii) (4):
The original relational schema is specified by means of concepts and properties in a schema of the M1-layer of the MOMENT platform. Both concepts and properties are instances of the elements of the relational metamodel defined in step 1.

(iv) (5) and (6):
Both relational and UML metamodels, respectively, are compiled into algebraic theories by means of the *Theory Compiler* module of the Framework. The compilation uses the concepts to define the sorts of the new theory and the properties to define constructors and query operators. The generated theories are interpreted by the CafeOBJ TRS, providing the respective algebras to define models in the TRS as algebraic terms.

(v) (7):
The semantic mapping that is specified between the concepts of both metamodels at the M2-layer is also compiled into another theory that extends the theories described above with a set of rewriting rules. This theory indicates how to transform a model of the source metamodel (relational metamodel) into a new model of the target metamodel (UML) in an automatic way.

(vi) (8):
The original relational schema, which is defined in step (4) at the M1-layer of the MOMENT platform, is compiled into a term of the relational algebra in the CafeOBJ TRS by means of the *Term Manager* module of the Framework.

(vii) (9):
The TRS evaluates the term that represents the initial relational schema in the algebra obtained in step (7). The user can manage this process through the *Evaluator* module of the Framework. The evaluation process can be carried in a step-by-step mode or in only one step with the full-evaluation mode, benefitting from the evaluation features of the TRS. The TRS reduces the initial term by applying the rewriting rules obtained

in step (7), generating a term of the target algebra.

(viii) (10):
This is the last step of the model transformation process. It parses the obtained term in step (9), defining a model in the M1-layer as an instance of the target metamodel defined at the M2-layer. There, it is disguised with the visual metaphor associated to the target metamodel in the graphical CASE tool.

In the model transformation process, the user only interacts with the MOMENT platform when defining the source and target metamodels (step (1) and (2)), the semantic mappings between the elements of both metamodels (step (3)) and the initial model (step (4)). The other steps are automatically carried out by the Framework, although the user can participate in the evaluation process by specifying the rewriting rules to be applied by the TRS at each step of the term reduction.

In the following sections, we explain phases (5), (6), (7), (8) and (9) in more detail, indicating how the TRS is able to perform model transformations providing formal support to the objectives of MDA.

5.2 Compilation of equational logic based theories

The relational and UML metamodels defined at the M2-layer of the MOMENT platform are compiled into theories based on equational logic in steps (5) and (6), respectively. The compilation of MOMENT metamodels into equational theories uses the concepts of the metamodel to obtain the sorts of the theory; for instance, the sorts Table, Field, ForeignKey for the relational metamodel, as well as the identifiers for these sorts, i.e., the sorts TableId, FieldId and ForeignKeyId. The properties of a MOMENT metamodel provide information about the structure of the term of a sort by means of the cardinalities. Thus, when a concept A is related to a concept B by means of a property that has cardinality 1..1, the constructor of the sort A looks like this : $op\ a\_ : B \rightarrow A$. Nevertheless, if the minimum cardinality is zero or the maximum cardinality is *many*, then the constructor for a term of the sort A looks like this: $op\ a\_ : ListB \rightarrow A$, where $ListB$ is a *sort* that permits the definition of lists, whose items are terms of *sort B*. As CafeOBJ belongs to the OBJ language family, it permits equational specification through several equational theories, such as associativity, commutativity, identity, idempotence and combinations between all these. This feature is reflected at the execution level by term rewriting by means of such equational theories.

Fig. 4 shows the constructors of the compiled theory for the relational metamodel; and Fig. 5 shows the constructors for the UML metamodel. We obviate the definition of sorts and other constructors in the theory, as well as the definition of query operators, focusing on the elements of the metamodels that permit us to illustrate the example. We must point out that the constructors obtained for the UML theory permit us to define terms that represent

```
-- FIELD: id, type, is nnv, is pk, tableid, dbid
op field _ _ _ _ _ _ : FieldId Datatype Bool Bool TableId DatabaseId -> Field {constr}
-- FOREIGNKEY: id, field list, related table, is unique, table that contains the fk, database
op foreignKey _ _ _ _ _ _ : ForeignKeyId ListField TableId Bool TableId DatabaseId -> ForeignKey {constr}
-- TABLE
op table _ _ _ _ : TableId ListField ListForeignKey DatabaseId -> Table {constr}
-- DATABASE
op database _ _ : DatabaseId ListTable -> Database {constr}
```

Fig. 4. Part of the relational theory

```
-- ATTRIBUTE: id, type, required, constant, identifier, ClassId, schemaid
op attribute _ _ _ _ _ _ _ : AttributeId Datatype Bool Bool Bool ClassId SchemaId -> Attribute {constr}
-- ASSOCIATION
op association _ _ _ _ : AssociationId AssociationEndId AssociationEndId SchemaId -> Association {constr}
-- ASSOCIATIONEND: id, id of the class, id of the association, isNavigable, ordering, aggregation, min
card., max card, changeability, visibility, id of the schema
op associationEnd _ _ _ _ _ _ _ _ _ _ _ : AssociationEndId ClassId AssociationId Bool OrderingKind
AggregationKind Cardinality Cardinality ChangeableKind VisibleKind SchemaId -> AssociationEnd {constr}
-- CLASS
op class _ _ _ : ClassId ListAttribute SchemaId -> Class {constr}
-- OOSCHEMA
op ooSchema _ _ : SchemaId ListClass -> OOSchema {constr}
```

Fig. 5. Part of the UML theory

UML-compliant models.

5.3 Compilation of rewriting logic-based theories

To transform the relational schema of the example, semantic mappings are defined between the concepts of both source and target metamodels in step (3). These mappings are compiled into an algebra that extends both relational and UML algebras (steps (5) and (6)) with a set of rewriting rules describing the guidelines for the model transformation. These rules are automatically applied by the TRS rewriting the initial term into a term of the target algebra. The new algebra constitutes the context where semantical relationships between the source and target ontologies are defined. To allow the transformation process, the new algebra must relate the *sorts* of the initial algebra (relational metamodel) to the *sorts* of the target algebra (UML metamodel). Relationships between the sorts of both algebras result in a subsort order that involves all the sorts. For instance, in the example, the sort Table is a subsort of the sort Class, indicating that a class can take the place of a table that was there before. Subsort relationships affect all the sorts of both algebras, even identifiers and lists, because they are the related concepts in the MOMENT algebra.

The properties that relate concepts in the MOMENT algebra define a canonical order among the sorts of the compiled algebras. This order is taken

into account to generate the rewriting rules. We present the rewriting rules that are applied to the relational schema of the example to obtain a semantically equivalent UML model in CafeOBJ syntax:

(i) Field

A field of a table becomes an attribute of a class in the term that represents a UML model. The rule reuses the features of the field (datatype, whether it is null or not and whether it is primary key) to generate an attribute. Field features also indicate the attribute datatype, whether it is required or not and whether it is the identifier of the class to which it belongs:

op field _ _ _ _ _ _ : FieldId Datatype Bool Bool TableId DatabaseId -> Attribute
eq field FI D NNV PK TI DBI = attribute FI D NNV false PK TI DBI .

(ii) Foreign Key

A foreign key can define an association between two classes in the UML context. The following rule is applied when the foreign key is unique and not null, obtaining an association 1..1 - 0..* between the classes generated from the related tables.

op foreignKey _ _ _ _ _ _ : ForeignKeyId ListAttribute TableId Bool TableId DatabaseId -> ListClass
ceq foreignKey FkI LA RTI U TI DBI =
(association FkI TI RTI DBI)(associationEnd TI TI FkI true unordered aggregate card 0 many frozen public DBI)(associationEnd RTI RTI FkI true unordered none card 1 card 1 frozen public DBI)
if U and isRequired (LA) .

(iii) Table

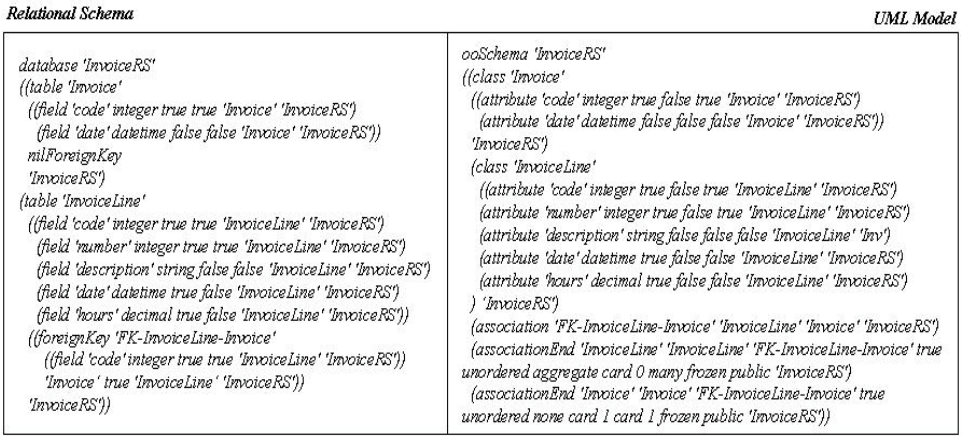
A table becomes a class. The rewriting rules must take into account the fact that a table is made up of fields and foreign keys, so that a field will become an attribute of the new class and a foreign key will become a set of elements of the UML model, i.e., an association and two association ends, according to the UML metamodel.

op table _ _ _ _ : TableId ListAttribute ListClass DatabaseId -> ListClass
eq table TI LA nilForeignKey DBI = (class TI LA DBI) .
eq table TI LA LC DBI = (class TI LA DBI) LC .

(iv) Database

Finally, a database is rewritten into a term of the sort OOSchema, representing the target UML model, by means of the following rule:

op database _ _ : DatabaseId ListClass -> OOSchema
eq database DBI LC = ooSchema DBI LC .



Relational Schema

```
database 'InvoiceRS'
((table 'Invoice'
  ((field 'code' integer true true 'Invoice' 'InvoiceRS')
    (field 'date' datetime false false 'Invoice' 'InvoiceRS'))
  nilForeignKey
  'InvoiceRS')
(table 'InvoiceLine'
  ((field 'code' integer true true 'InvoiceLine' 'InvoiceRS')
    (field 'number' integer true true 'InvoiceLine' 'InvoiceRS')
    (field 'description' string false false 'InvoiceLine' 'InvoiceRS')
    (field 'date' datetime true false 'InvoiceLine' 'InvoiceRS')
    (field 'hours' decimal true false 'InvoiceLine' 'InvoiceRS'))
  ((foreignKey 'FK-InvoiceLine-Invoice'
     ((field 'code' integer true true 'InvoiceLine' 'InvoiceRS'))
     'Invoice' true 'InvoiceLine' 'InvoiceRS'))
  'InvoiceRS'))
```

UML Model

```
ooSchema 'InvoiceRS'
((class 'Invoice'
  ((attribute 'code' integer true false true 'Invoice' 'InvoiceRS')
    (attribute 'date' datetime false false false 'Invoice' 'InvoiceRS'))
  'InvoiceRS')
  (class 'InvoiceLine'
    ((attribute 'code' integer true false true 'InvoiceLine' 'InvoiceRS')
    (attribute 'number' integer true false true 'InvoiceLine' 'InvoiceRS')
    (attribute 'description' string false false false 'InvoiceLine' 'Inv')
    (attribute 'date' datetime true false false 'InvoiceLine' 'InvoiceRS')
    (attribute 'hours' decimal true false false 'InvoiceLine' 'InvoiceRS')
  ) 'InvoiceRS')
  (association 'FK-InvoiceLine-Invoice' 'InvoiceLine' 'Invoice' 'InvoiceRS')
  (associationEnd 'InvoiceLine' 'InvoiceLine' 'FK-InvoiceLine-Invoice' true
unordered aggregate card 0 many frozen public 'InvoiceRS')
  (associationEnd 'Invoice' 'Invoice' 'FK-InvoiceLine-Invoice' true
unordered none card 1 card 1 frozen public 'InvoiceRS'))
```

Fig. 6. Original term representing the source relational schema, and the generated term representing the target UML model

5.4 Term rewriting process

Step (8) compiles the relational schema defined in the M1-layer of the MOMENT platform, obtaining the algebraic term in Fig. 6.a. The TRS applies the rewriting rules specified above to this initial term, obtaining a term of the target algebra (UML), shown in Fig. 6.b. This term is parsed and is defined as a UML model in the M1-layer of the MOMENT platform. There, it is automatically related to graphical pictures in a specific CASE tool.

During the term rewriting process, some additional information could be required in order to perform a correct transformation, as the metamodels do not have the same expressive power. For instance when a transformation case has not been taken into account or when several rewriting rules can be applied to the source model. In this cases a visual wizard helps the user to chose one option or even to add a new transformation rule, providing a visual interface for the CafeOBJ interpreter.

To benefit from the MOMENT features, we have integrated the functionality of the MOMENT Framework into a visual modeling environment [22]. In this way, we can relate algebraic specifications to visual notations so that the user can use these graphics to build a model. The CASE tool we have chosen is MS Visio [23]. We have developed an add-in that permits the definition of metamodels with concepts and properties. Fig. 7 shows the interface that permits the definition of the graphical symbol of a class in the MOMENT platform.

In visual CASE tools, models are usually defined by dropping graphical primitives on a sheet where the model is defined. By means of the developed add-in, dropping a primitive on the sheet does not only add a figure to the model but it also defines it algebraically, specifying the model so that it can be manipulated afterwards.

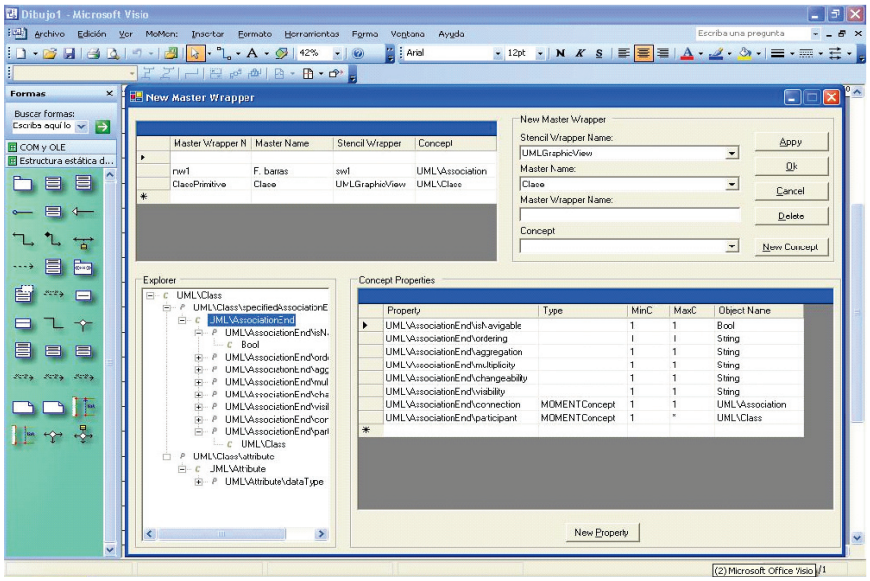


Fig. 7. Visual interface to define a graphical primitive algebraically

6 Conclusions and further work

Nowadays, software applications have become complex combinations of technology, which have to be well understood in order to manage them. The development of software artifacts involves models that can be mixed with others to obtain an entire system from partial views or that can be interconnected with others in order to guarantee both interoperability in a distributed environment and their implementations.

Model management [8] is an emerging research field whose aim is to resolve data model integration and interoperability by means of generic operators. Similarly, MDA raises the level of abstraction in the software development process by treating models as primary artifacts. MDA potentially covers the modeling of all aspects of a system throughout its life cycle, making software development processes easier and more automated.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. The MOMENT Framework benefits from the best features of current visual CASE tools and from the main advantages of formal environments such as term rewriting systems, combining both industrial and research features.

In this paper, we have presented the generic model transformation mechanism provided by the MOMENT Framework, focusing on the use of the CafeOBJ TRS to perform automatic translations of models. This mechanism has been applied to platform-independent models in the MDA context, obtaining a UML model from a relational schema. The functionality of TRSs permits us to deal with model management from a more abstract point of view, since the application of rewriting rules can be performed in a trans-

parent way. This fact allows us to focus all the efforts on the specification of models without having to take the evaluation logic into account. Our approach constitutes an algebraic baseline to cope with the future model transformation language QVT, providing a user-friendly environment to manipulate models from a visual CASE tool [22]. In [16], we present the fundamental mainstay on which we have built our MOMENT platform taking into account our previous experience in the industrial project RELS, a tool for the recovery of legacy systems.

Currently, we are working with transformations between relational schemas and UML models. In the near future, we will also take into account software architecture specifications by means of PRISMA ADL [17], studying semantic interoperability between software architectures and other types of software artifacts represented through UML models.

7 Acknowledgments

This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

References

[1] S. Cook, *Domain-Specific Modeling and Model Driven Architecture*, MDA Journal, (January 2004).

[2] S. Melnik, E. Rahm, P. A. Bernstein, *Rondo: A Programming Platform for Generic Model Management*, (Extended Version), Technical Report, Leipzig University, 2003. Available at http://dol.uni-leipzig.de/pub/2003-3.

[3] OMG, *The Model-Driven Architecture*, Guide Version 1.0.1, OMG Document: omg/2003-06-01. Available from www.omg.org.

[4] OMG, *Meta Object Facility 1.4*, OMG Document: formal/02-04-03. Available from www.omg.org.

[5] OMG, *MOF 2.0 Query/Views/Transformations RFP*, OMG Document ad/2002-04-10. Available from www.omg.org.

[6] Alagic, S. and Bernstein, P.A., *A Model Theory for Generic Schema Management*, in Proceedings of DBPL'01, G. Ghelli and G. Grahne (eds), Springer-Verlag, (2001).

[7] Madhavan, J., P.A. Bernstein, and E. Rahm, *Generic Schema Matching using Cupid*, MSR Tech. Report MSR-TR-2001-58, 2001, http://www.research.microsoft.com/pubs (short version in VLDB 2001).

[8] Bernstein, P.A., Levy, A.Y., Pottinger, R.A., *A Vision for Management of Complex Models*, Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00)).

[9] N. Revault, H.A. Sahraoui, G. Blain and J.F. Perrot, *A Metamodeling technique: The MÉTAGEN system*, TOOLS 16: TOOLS Europe'95, Prentice Hall, pp. 127-139. Versailles, France. Mar. 1995.

[10] N. Revault, X. Blanc and J. F. Perrot, *On Meta-Modeling Formalisms and Rule-Based Model Transforms*, Comm. at workshop, In Iwme'00 workshop at Ecoop'00, Jean Bézivin and Johannes Ernst (ed), Sophia Antipolis and Cannes, France, June, 2000.

[11] ISO/IEC 10746-1, 2, 3, 4 — ITU-T Recommendation X.901, X.902, X.903, X.904, *Open Distributed Processing - Reference Model*. OMG, 1995-96.

[12] CDIF Technical Committee, *CDIF Framework for Modeling and Extensibility*, Electronic Industries Association, EIA/IS-107, January 1994. See http://www.cdif.org/.

[13] Razvan Diaconescu, Kokichi Futatsugi, *An overview of CafeOBJ*. Electronic Notes in Theoretical Computer Science vol. 15: (2000)

[14] Microsoft Research (Don Syme), *The F# official web site*: http://research.microsoft.com/projects/ilx/fsharp.aspx.

[15] N. Martí-Oliet and J. Meseguer, *Rewriting logic: roadmap and bibliography*, Theoretical Computer Science, vol. 285, issue 2 (August 2002), pp. 121-154. Preprint version available at http://maude.cs.uiuc.edu.

[16] A. Boronat, J. Pérez, J. Á. Carsí, I. Ramos, *Two experiencies in software dynamics*. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. April 2004.

[17] J. Pérez, I. Ramos , J. Jaén, P. Letelier, E. Navarro , *PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*, 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 IEEE Computer Society Press pp. 59-66.

[18] Elliotte Rusty Harold, *XML Bible*, IDG Books Worldwide, 1999.

[19] OMG, *XML Metadata Interchange (XMI) Specification*, OMG Document formal/02-01-01.

[20] W3C, *XSL Transformations (XSLT) v1.0. W3C Recommendation*, http://www.w3.org/TR/xslt, Nov. 1999.

[21] M. Peltier, J. Bézivin, and G. Guillaume, *MTRANS: A general framework, based on XSLT, for model transformations*. In WTUML01, Proceedsings of the Workshop on Transformations in UML, Genova, Italy, Apr. 2001.

[22] Artur Boronat, José Á. Carsí, Isidro Ramos, Julián Pedrós, *Soporte Formal para Entornos Visuales de Modelado*, I Workshop Metodologies for Dynamic User Interfaces Development. Ed. Computer Science Department. Castilla-La-Mancha University. Albacete (Spain), July, 2004. (In Spanish).

[23] Graham Wideman, *Microsoft Visio 2003 Developer's Survival Pack*, Trafford, 2004.

Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos♦

Artur Boronat, Julián Pedrós, José Á. Carsí, Isidro Ramos

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
C/Camino de Vera s/n
E-46071 Valencia- España
{aboronat | jpedros | pcarsi | iramos}@dsic.upv.es

Resumen. El modelado específico a dominio se está consolidando como una forma de desarrollo de software con herramientas como GME (Generic Metamodel Environment) y MetaEdit+, entre otras. La clave del éxito de estas herramientas es la proximidad de la ontología de modelado aportada al problema que intenta solucionar el usuario analista y/o programador, de manera que los modelos definidos son fácilmente interpretables en el dominio del problema. MOMENT es una herramienta que permite definir este tipo de modelos siguiendo una aproximación algebraica. Con el fin de obtener una herramienta de gestión de modelos que utilice metáforas gráficas cercanas al usuario, se presenta una arquitectura que integra MOMENT en herramientas CASE que posean un entorno gráfico, pudiendo así utilizar sus capacidades visuales en la definición de modelos soportados por MOMENT. Con tal fin se indican los elementos gráficos de una herramienta CASE que se asocian con elementos de un metamodelo definido en MOMENT y cómo se almacenan esas asociaciones en el propio repositorio de MOMENT. Se ha realizado una implementación de esta arquitectura para una herramienta CASE concreta, y se presenta cómo se define la vista gráfica de un metamodelo definido algebraicamente utilizando su interfaz.

Palabras clave: gestión de modelos, herramientas CASE, modelado, extensibilidad, especificaciones algebraicas.

1. Introducción

El desarrollo de software implica el estudio del sistema de información que constituye el problema a resolver (espacio del problema), las tecnologías que permiten implementar una solución (espacio de la solución) y el proceso de desarrollo del software que obtiene una solución tecnológica a los requisitos del sistema de información.

♦ Este artículo ha sido financiado por el Proyecto Nacional de Investigación, Desarrollo e Innovación DYNAMICA TIC 2003-07804-C05-01 y el Proyecto Nacional PBC-03-00 de "Metodologías de desarrollo de interfaces de usuario dinámicas".

Desde el punto de vista de la solución, los lenguajes de programación han ido incrementado paulatinamente su nivel de abstracción con el objetivo de aproximarse al espacio del problema. Así pues del lenguaje ensamblador se pasó a los lenguajes procedurales, posteriormente a la tecnología orientada a objeto, desembocando en los lenguajes orientados a aspectos y componentes en estos últimos años. Estos esfuerzos han permitido mejorar el proceso de desarrollo del software incrementando la interoperabilidad, la reutilización de artefactos software y disminuyendo los costes de desarrollo. No obstante, aún no se ha conseguido alcanzar un paradigma que permita representar cualquier problema y que pueda generar su solución de forma automática. Además, difícilmente se alcanzará este punto, puesto que cada dominio dispone de unas propiedades específicas, que se deben tener en cuenta para poder definir el problema con precisión. El hecho de conseguir una tecnología universal suficientemente abstracta como para solucionar cualquier problema equivaldría a afirmar que la lengua "esperanto" será una especie de *lingua franca* en la que nos expresaremos los seres humanos en el futuro. Y esta situación parece improbable puesto que cada lengua posee su riqueza expresiva en el contexto socio-cultural en el que es usada, al igual que cada dominio de problema necesita mantener su vocabulario específico.

Desde el punto de vista del dominio, la Ingeniería del Software también ha conocido una serie de técnicas de modelado que permiten representar el problema aproximándolo hacia el espacio de la solución. Las técnicas de modelado se pueden clasificar en dos tipos, tomando como criterio de decisión la proximidad de la metáfora del modelo respecto al dominio del problema: modelado independiente de dominio y modelado específico a dominio. El modelado independiente de dominio intenta representar cualquier problema mediante una notación gráfica basada en un paradigma determinado, como por ejemplo UML [2] que se centra en el paradigma orientado a objeto. El modelado específico a dominio permite utilizar los conceptos del propio dominio para especificarlo. El problema en ambas aproximaciones es la falta de un soporte formal que garantice un proceso unívoco que permita obtener un producto software consistente y eficiente, y que permita validar los requisitos.

MOMENT es una plataforma [1] que permite definir modelos como especificaciones algebraicas de una teoría expresada en lógica ecuacional condicional [22]. La semántica operacional de estas especificaciones algebraicas se evalúa mediante un sistema de reescritura de términos, lo que permite soportar las siguientes aplicaciones [3]: ejecución (animación) de especificaciones formales, análisis *narrowing*, *model checking* y demostraciones formales, entre otras.

A pesar que el uso de formalismos aporta propiedades ventajosas al proceso de desarrollo del software, usualmente no han tenido éxito fuera del ámbito investigador. Esto es debido al grado de preparación teórica que necesitan los usuarios de herramientas formales.

En este artículo, se presenta un enfoque genérico para integrar MOMENT en una herramienta CASE con un entorno gráfico, utilizando su interfaz como mecanismo para asociar una metáfora visual a un metamodelo descrito mediante una especificación algebraica. Como herramienta CASE para realizar la primera implementación de la arquitectura presentada se ha elegido MS Visio [4], porque proporciona buenas propiedades de extensibilidad, tanto en las facilidades gráficas, ya que dispone de un editor gráfico, como en la funcionalidad interna. De esta manera, el

usuario puede especificar modelos de forma algebraica basándose en una metáfora visual conocida a través de una interfaz sencilla.

En esta sección se ha introducido la problemática tradicional existente en el proceso de desarrollo del software que constituye la motivación de nuestro trabajo y hemos presentado nuestro objetivo. En el apartado 2 se indica la estructura de la plataforma de gestión de modelos MOMENT y los elementos principales que se utilizan en ella. En el apartado 3, se describe los elementos principales del entorno gráfico de una herramienta visual de modelado. En el apartado 4, se indica la arquitectura que hemos utilizado para añadir el soporte formal a una herramienta CASE, identificando las asociaciones entre los elementos de su editor gráfico y los sorts (o nombres de tipos) de la teoría algebraica utilizada en MOMENT, y se describe el mecanismo que permite asociar una metáfora gráfica a un metamodelo especificado algebraicamente en MOMENT. En el apartado 5 se discuten trabajos relacionados, y finalmente, en el apartado 6, se resumen una serie de conclusiones y trabajos futuros.

2. La plataforma MOMENT (MOdel manageMENT)

MOMENT es una plataforma de gestión de modelos implementada algebraicamente usando el modelo computacional λ-cálculo mediante el lenguaje funcional F# [5]. MOMENT utiliza también el entorno CafeObj [6] como sistema de reescritura de términos para realizar transformaciones automáticas de modelos entre diferentes metamodelos (transformaciones intermodelo) o de un mismo metamodelo (transformaciones intramodelo).

La plataforma posee cuatro niveles de abstracción, siguiendo la cultura de metamodelado presentada en el estándar MOF [7], aunque no implementa directamente su lenguaje abstracto para definir metamodelos. Cada nivel está formado por un conjunto de esquemas que a su vez están constituidos por un conjunto de conceptos y propiedades. Los conceptos permiten representar entidades descriptibles en un dominio determinado, y las propiedades establecen dichas descripciones, pudiendo asociar dos conceptos mediante una propiedad o bien describir un concepto mediante una propiedad que contiene un valor básico.

Los niveles que consituyen la plataforma MOMENT son los siguientes:

- Nivel M3: contiene los elementos (conceptos y propiedades) necesarios para poder describir cualquier metamodelo en el siguiente nivel. Es el nivel más abstracto de la plataforma.
- Nivel M2: sus esquemas constituyen metamodelos, donde sus respectivos conceptos y propiedades indican como definir un modelo en el siguiente nivel.
- Nivel M1: sus esquemas representan modelos de un determinado metamodelo del nivel M2. Los conceptos y propiedades que forman un modelo permiten definir información en el siguiente nivel.
- Nivel M0: sus esquemas representan instanciaciones de modelos del nivel M1 que contienen datos concretos.

3. Elementos principales del entorno visual de herramientas CASE de modelado

El área de trabajo de las herramientas CASE de modelado visual tienen cuatro elementos en común, que son descritos a continuación:

- *Figuras*: Símbolos que pueden ser prediseñados o pueden ser de creación propia, ajustándose a las necesidades del modelo que el usuario necesita representar. Dichos símbolos pueden contener información de valor añadido mediante propiedades.
- *Hoja de dibujo*: Zona de trabajo donde se depositan (pegan) las figuras, constituyendo un diagrama o modelo gráfico concreto.
- *Primitivas gráficas*: Distintos tipos de figuras que se pueden definir en la hoja de dibujo de la herramienta CASE. Una primitiva gráfica permite definir una figura en la hoja de dibujo. Poseen unos atributos o propiedades por defecto que definen las características de sus figuras.
- *Vistas gráficas*: Plantillas que agrupan a las primitivas gráficas según el modelo o tipo de diagrama al que pertenecen, encapsulando la ontología gráfica que permite definir un modelo.

MS Visio 2003 es una herramienta visual de modelado fácilmente personalizable y extensible mediante la tecnología .NET. En el vocabulario propio de la herramienta, los elementos que hemos identificado anteriormente de forma genérica reciben los siguientes nombres: las figuras son denominadas *shapes*; la hoja de dibujo *shapesheet*; una primitiva gráfica recibe el nombre de *master*; y la vista que agrupa una serie de primitivas gráficas recibe el nombre de *stencil*.

La personalización del entorno visual de Visio se realiza mediante add-ons, es decir, conjuntos de *stencils* que proporcionan la información que constituye un metamodelo completo. La extensión de la herramienta se consigue mediante add-ins, que son módulos que permiten añadir funcionalidad a la herramienta. Dada la facilidad de extensión aportada por Visio 2003, ha sido la herramienta escogida para construir la interfaz gráfica de usuario de MOMENT.

4. Arquitectura de la solución

Partiendo de la plataforma formal de gestión de modelos MOMENT, se ha desarrollado un módulo, llamado *MOMENT Integrator*, que permite asociar una metáfora gráfica a un metamodelo formal mediante la interfaz de herramientas CASE con entornos visuales que proporcionen cierto grado de extensibilidad, como MS Visio 2003.

4.1. Arquitectura

En la Fig. 1, se muestra la arquitectura que permite integrar la plataforma MOMENT en una herramienta CASE con un entorno gráfico de forma genérica. Aunque se ha contextualizado su implementación en el entorno Visio 2003, se está pensando en

desarrollar la misma arquitectura para integrar MOMENT en la herramienta de desarrollo Eclipse aprovechando su entorno gráfico Graphical Editor Framework [20]. La arquitectura utilizada es la tradicional separación en tres capas: una interfaz que permite representar gráficamente metamodelos y modelos, la funcionalidad que permite asociar la metáfora gráfica a un metamodelo especificado algebraicamente y la capa de persistencia en la que se almacena toda la información utilizada.

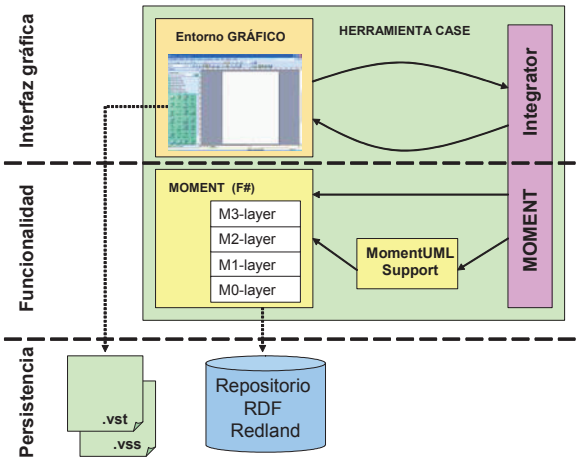


Fig. 1. Arquitectura de la integración de MOMENT en una herramienta CASE.

En la interfaz gráfica, la herramienta CASE ofrece la funcionalidad necesaria para definir los gráficos que se van a utilizar en un metamodelo, y permite acceder a la funcionalidad del módulo *MOMENT Integrator*.

En la capa de funcionalidad, el módulo *MOMENT Integrator* permite definir asociaciones entre los elementos gráficos que se han definido mediante el editor gráfico de la herramienta CASE y los términos algebraicos mediante los cuales MOMENT representa un metamodelo. Como veremos en la siguiente sección, dichas asociaciones se almacenan en la propia plataforma como instancias de clases UML a través de la librería MomentUMLSupport.

En la capa de persistencia se pueden distinguir dos unidades principales de almacenamiento: la de los elementos gráficos de la herramienta CASE y la de la plataforma MOMENT.

Las herramientas CASE suelen almacenar la información gráfica mediante una serie de ficheros. En el caso específico de la herramienta Visio, los modelos gráficos se almacenan mediante dos tipos de ficheros. Los ficheros con extensión *.vss* almacenan los documentos que se crean, o sea el área dibujada con figuras (*shapesheet*), junto con las plantillas o templates a partir de las cuales se ha creado el diagrama o modelo. Los ficheros con extensión *.vst* almacenan las plantillas con *masters* (*stencils*). Por defecto, Visio ya incorpora un gran número de plantillas con todo tipo de formas para todo tipo de modelos, pero un usuario puede crearse y guardar sus propias plantillas con una colección de formas de otras plantillas o con formas creadas por el mismo.

Por otro lado, MOMENT almacena toda la información en un repositorio RDF puesto que los conceptos y propiedades manejados en la plataforma son un reflejo de

los recursos y propiedades RDF. El repositorio utilizado es Redland [19] y permite almacenar la información sobre esquemas de todos los niveles de la plataforma y asociaciones entre elementos gráficos del entorno visual y términos de MOMENT, explotando las facilidades que nos ofrece el entorno gráfico de la herramienta CASE.

4.2. Diseño de la capa de funcionalidad

Mediante el módulo *MOMENT Integrator*, los elementos gráficos de la herramienta CASE se vinculan con los elementos de la plataforma MOMENT. Cómo nuestro objetivo consiste en la definición de la metáfora gráfica asociada a un metamodelo, nos centramos en el uso de los esquemas del nivel M2 de la plataforma, que son los metamodelos. De manera que, un esquema de este nivel se asocia con una vista gráfica (*stencil* en Visio). Por otro lado, para representar gráficamente los conceptos y las propiedades de los esquemas que constituyen un metamodelo en el nivel M2 de la plataforma, hacemos uso de las primitivas gráficas que componen la vista gráfica elegida (*masters* en el contexto de Visio).

Para asociar la información de cada elemento de un metamodelo específico de MOMENT a las figuras gráficas que proporciona el entorno visual, se ha definido una composición, expresada en notación UML en la Fig. 2. En esta composición participan las siguientes clases:

– *GraphicViewWrapper*: es la clase agregada y encapsula información sobre una vista gráfica determinada y sobre el esquema correspondiente del nivel M2 de la plataforma. Es en esta clase, por tanto, donde se asocia el esquema del metamodelo a la plantilla gráfica. Una instancia de la clase *GraphicViewWrapper* está formado por un conjunto de instancias de la clase *GraphicPrimitiveWrapper* que definen las primitivas gráficas del metamodelo.

  La información que contiene esta clase es el identificador o nombre del contenedor, el de la vista gráfica (*stencil* en el contexto de Visio) al que se le asocia el metamodelo junto con la ruta completa donde se encuentra el fichero en el que se almacena la vista gráfica, y finalmente los atributos que identifican la definición del metamodelo en la plataforma, es decir, los identificadores del nivel y del esquema. El tipo de datos *Repository* permite abstraer el mecanismo de almacenamiento de una herramienta CASE específica en el módulo coordinador *MOMENT Integrator*.

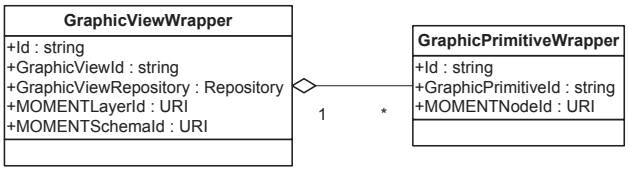


Fig. 2. Diagrama de clases UML que modela las asociaciones de elementos gráficos con elementos de la plataforma MOMENT.

– *GraphicPrimitiveWrapper*: es la clase componente que encapsula la información sobre una primitiva gráfica específica (*master* en el contexto de Visio) y el concepto correspondiente de un metamodelo de la plataforma MOMENT. Este

concepto puede disponer de una serie de propiedades, definidas como términos en MOMENT.

Los atributos de esta clase son un identificador o nombre para el contenedor, el identificador de la primitiva gráfica que pertenece a la vista gráfica especificada y el identificador del nodo MOMENT (concepto o propiedad) que se asocia.

La asociación entre elementos gráficos de la herramienta CASE con elementos de la plataforma definida en el diagrama de clases UML de la Fig. 2, se almacena utilizando la propia plataforma MOMENT como repositorio orientado a objetos, puesto que permite definir modelos UML y sus extensiones.

Para llegar a tal fin, se aprovecha la parte del metamodelo UML, especificado como un esquema en el nivel M2 de la plataforma, que permite la definición de clases y de asociaciones. Por tanto, se ha definido el modelo UML de la Fig. 2 como un esquema del nivel M1 de la plataforma. De esta manera, MOMENT ya dispone de la información necesaria para almacenar instancias de las clases del diagrama de clases definido.

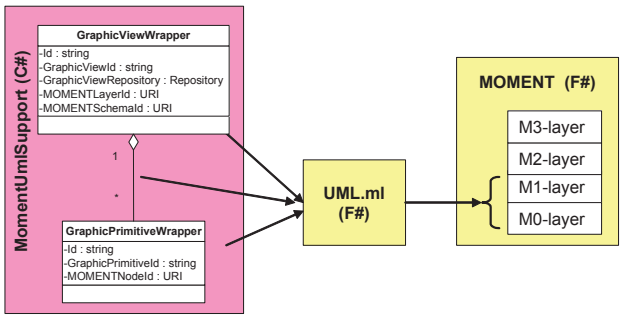


Fig. 3. Estructura de la librería MomentUMLSupport

Para facilitar el uso de MOMENT como repositorio orientado a objetos, se ha desarrollado una API que se comporta como la interfaz directa entre la plataforma de gestión de modelos y el módulo *MOMENT Integrator*. Esta librería, llamada MomentUMLSupport, ha sido desarrollada en C# y utiliza las funciones de un módulo F#, llamado UML.ml, que utiliza la funcionalidad de MOMENT para insertar instancias de las clases de un modelo del nivel M1, en una instanciación de modelo del nivel M0. La Fig. 3 muestra la estructura de la librería MomentUMLSupport.

El API de la librería MomentUMLSupport permite manipular instacias de clases de cualquier modelo UML, involucrando la instanciación, consulta o destrucción de objetos en el nivel M0 de la plataforma. De manera que cuando se instancia una clase mediante esta librería, se está almacenando la información en la plataforma.

4.3. Definición de la metáfora gráfica de un metamodelo

Para construir un metamodelo mediante una herramienta CASE asociamos un metamodelo MOMENT (un esquema del nivel M2) a una vista gráfica. Una vez definido el significado de esa vista, se especifica cada uno de sus primitivas gráficas con conceptos y propiedades de un esquema del nivel M2 de la plataforma, completando la metáfora gráfica asociada al metamodelo. A continuación se detallan

estos dos pasos en el contexto de la implementación que se ha realizado para la herramienta MS Visio 2003:

1. En primer lugar, para definir el *stencil* se accede a la interfaz de definición que se muestra en la Fig. 4, donde se selecciona el *stencil* que se desea asociar y el esquema del nivel M2 de la plataforma que define el metamodelo.

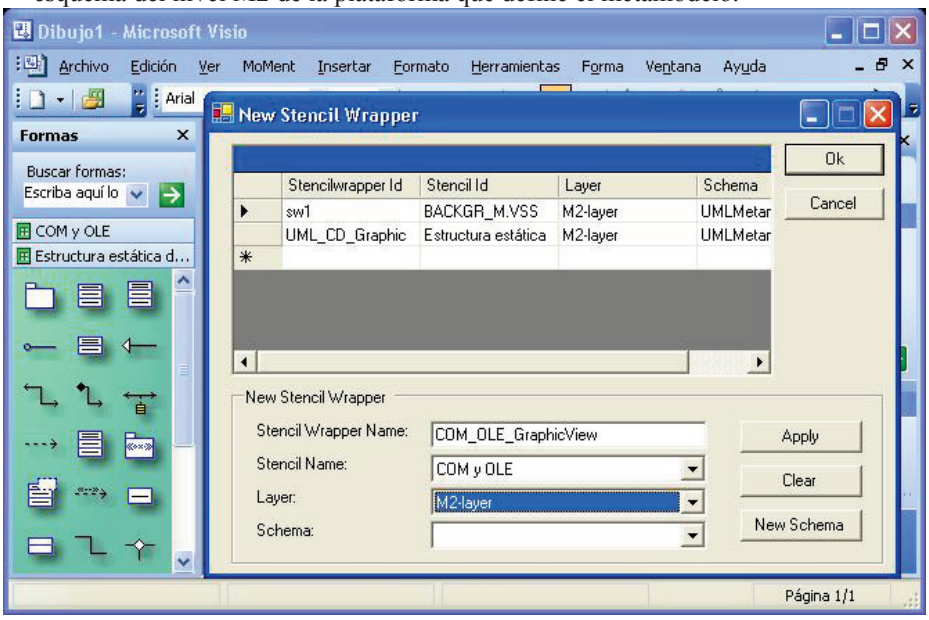


Fig. 4. Creación de un metamodelo mediante VisioMoment

2. Una vez asociada una vista gráfica a un metamodelo MOMENT mediante una instancia *GraphicViewWrapper*, ya sea por haber seguido los pasos anteriores o por haber cargado la plataforma MOMENT de un repositorio Redland, se definen gráficamente los conceptos y propiedades que posee el metamodelo en MOMENT.

   Para ello se accede a la interfaz de la Fig. 5, donde se selecciona un metamodelo definido gráficamente. Acto seguido, se selecciona una primitiva gráfica de la vista seleccionada y un nodo (concepto o propiedad) del metamodelo MOMENT especificado.

   Después de haber definido una primitiva gráfica, se pueden consultar las propiedades ocultas que describen al nodo MOMENT, es decir, aquéllas que no están definidas gráficamente. La información de esas propiedades aparece en la lista inferior, y además se puede navegar recursivamente por las propiedades mediante el árbol situado en la parte izquierda inferior de la ventana.

Una vez se ha definido un metamodelo en la herramienta CASE, se puede definir un modelo de la forma convencional mediante la técnica *drag-and-drop* característica de este tipo de entornos de modelado, soltando primitivas gráficas de la vista gráfica sobre la hoja de dibujo, creando una nueva figura. El módulo *MOMENT Integrator* también enriquece esta funcionalidad definiendo los términos del esquema correspondiente del nivel M1 de forma automática y transparente al usuario. De manera que cuando un nodo del metamodelo tiene asignado un conjunto de

propiedades ocultas, éstas se instancian tomando como valor el valor por defecto especificado para la propiedad o uno asignado de forma automática por el módulo *MOMENT Integrator*. De manera que no sólo definimos un modelo gráfico, sino que además utilizamos la información semántica asociada a un metamodelo, tanto para los conceptos como para las propiedades.

Además, cuando disponemos de un metamodelo en MOMENT que ya ha sido asociado a una metáfora gráfica, se puede cargar un modelo de ese metamodelo, directamente del nivel M1 de MOMENT. Automáticamente se asocian las figuras, que son instancias de las primitivas gráficas de la vista asociada al metamodelo, a los términos concepto y propiedad del modelo formal, dando lugar a un diagrama que representa gráficamente el modelo cargado mediante la metáfora del metamodelo.

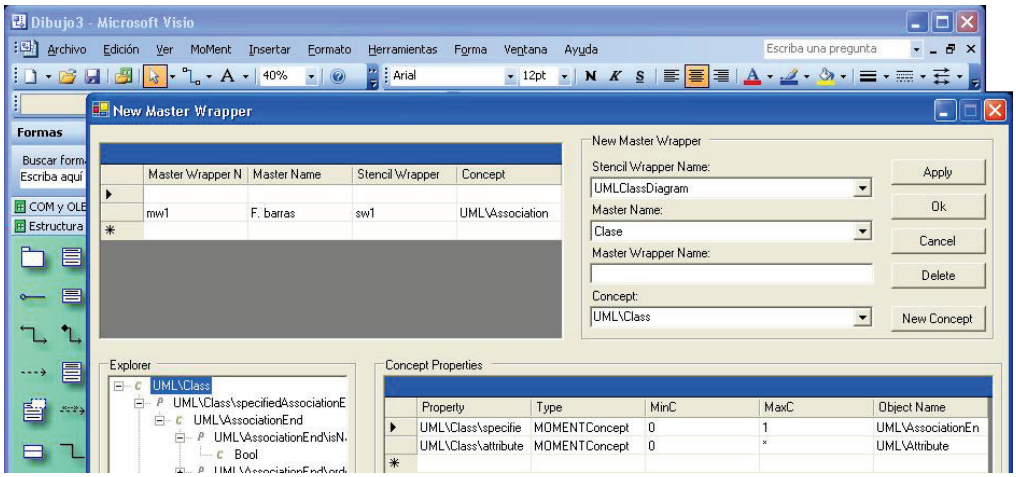


Fig.5. Creación de los conceptos y propiedades de un metamodelo

5. Trabajos relacionados

La definición de metáforas gráficas para técnicas de modelado no es una idea nueva en Ingeniería del Software como se indica en [8], pero pocas son las herramientas que permiten especificaciones formales, por no decir ninguna de las herramientas CASE más conocidas. En esta sección tratamos las herramientas CASE más conocidas en dos ámbitos de modelado: el independiente de dominio y el específico a dominio.

La técnica de modelado más representativa de las independientes de dominio es, sin duda alguna, UML [2]. UML aporta una notación gráfica para diseñar productos software siguiendo el paradigma orientado a objetos, pero no proporciona una especificación formal y sólida que garantice la producción de software de calidad y consistente.

La falta de una definición formal de los elementos del estándar UML provoca que el propio estándar vaya evolucionando sin garantía alguna sobre compatibilidad entre las diferentes versiones. De esta manera, una herramienta CASE que soporte UML, perfectamente puede quedar desfasada tras la publicación de una nueva versión del estándar. La existencia de una definición formal permitiría utilizar herramientas para

demostrar que la nueva versión es compatible con la anterior o incluso facilitar la migración automática hacia la nueva versión.

No obstante, hoy en día, existen herramientas CASE que permiten generar un producto software total o parcial a partir de un modelo UML, como Rational Rose [9], MS Visio [10], ArgoUML [11]. Estas herramientas únicamente suelen generar el esqueleto de la aplicación software final y las herramientas que proporcionan productos finales suelen ser difícilmente extensibles, lo que no permite seguir la contiua evolución de UML, y se suelen centrar en la generación de un tipo específico de aplicaciones. La funcionalidad de algunas de estas herramientas ha sido embebida en entornos de desarrollo (como Rational Rose XDE Developer [12] para Visual Studio .NET y EMF (Eclipse Modeling Framework) [13] para Eclipse) explotando los modelos UML para proporcionar facilidades durante la implementación del producto software. En [14] se realiza un amplio estudio de las propuestas de formalización de UML, pero ninguna de ellas ha llegado a sensibilizar a OMG para enriquecer el estándar UML.

Por otra parte, las técnicas de modelado específicas a dominio se están consolidando como una fuerte alternativa a las independientes de dominio. Esto es debido a que permiten mejorar el proceso de producción de software especializando a los usuarios en el espacio del problema (su área de trabajo) y no en el espacio de la solución (implementación tecnológica). La computación integrada en modelos (MIC – Model Integrated Computing) [15] es una aproximación de desarrollo de sistemas y de software que permite el uso de modelos específicos a dominio para representar aspectos relevantes de un sistema. El ciclo de desarrollo MIC consiste en identificar los conceptos de un dominio, sus atributos y relaciones obteniendo un metamodelo concreto. Este metamodelo se traduce a un entorno de diseño específico del dominio (DSDE – Domain Specific Design Environment) que permite definir modelos en un dominio. Además los DSDE suelen obtener código ejecutable, realizar análisis o animar los modelos. Algunas de las herramientas que destacan en este campo son GME [16], Atom3 [17] y MetaEdit+ [18]. Todas estas herramientas constituyen un marco de metamodelado basadas en un lenguaje abstracto para definir la sintaxis, semántica y visualización de lenguajes específicos a dominio.

En cambio, muchas de ellas utilizan lenguajes abstractos propietarios o UML (como GME) para definir metamodelos en lugar de utilizar un formalismo adecuado. De esta manera, los intérpretes de modelos obtenidos a partir de un metamodelo concreto suelen tener una complejidad considerable. Además, los transformadores de modelos se basan en las capacidades de estos lenguajes para realizar dichas traducciones de modelos a metamodelos diferentes. Por otra parte, ninguna de ellas consta de un mecanismo para definir metáforas gráficas como el que proporciona el entorno gráfico de MS Visio.

Mediante el *add-in* que hemos desarrollado para dicha herramienta CASE, no somos sólo capaces de definir metamodelos específicos a dominio de una forma sencilla y consiguiendo una buena representación gráfica de los conceptos relevantes de un dominio, sino que también ofrecemos soporte formal para notaciones gráficas utilizadas masivamente para el modelado independiente de dominio, como UML. Esta aportación se consigue mediante las especificaciones algebraicas que representan artefactos software en MOMENT.

6. Conclusiones y trabajo futuro

El modelado constituye un punto clave en el proceso de desarrollo del software, contando con diferentes aproximaciones. El modelado independiente de dominio que permite diseñar soluciones basándose en un determinado paradigma, como UML se centra en la orientación a objetos. Por otra parte, el modelado específico a dominio permite diseñar un producto software utilizando los conceptos del dominio del problema. Ambos tipos de técnicas de modelado suelen proporcionar una representación gráfica y una serie de facilidades para generar el producto software, total o parcialmente, y para transformar modelos que representan una determinada implementación tecnológica.

La teoría de grafos es el formalismo más utilizado para representar modelos y realizar transformaciones entre ellos pero su aceptación viene limitada por la complejidad de los algoritmos que proporcionan una solución a este tipo de problemas. En cambio, el enfoque algebraico de la plataforma de gestión de modelos MOMENT, permite abordar el problema de una forma genérica utilizando sistemas de reescritura de términos para realizar dichas transformaciones de una forma sencilla puesto que la implementación consiste únicamente en definir un conjunto de reglas de reescritura.

En este artículo presentamos un mecanismo de representación gráfica para los metamodelos especificados algebraicamente en MOMENT. De esta manera, permitimos definir metáforas gráficas asociándolas a metamodelos formales utilizando una herramienta CASE de gran difusión y fácilmente extensible. Este hecho permite acercar la gestión formal de modelos al usuario final a través de una interfaz gráfica fácilmente personalizable.

Para llegar a tal fin, hemos descrito la plataforma MOMENT y los principales elementos del entorno gráfico de una herramienta CASE. A continuación hemos descrito la arquitectura de un módulo que permite extender dichas herramientas, las asociaciones entre sus elementos gráficos y los elementos de la plataforma MOMENT, contextualizando la implementación en la herramienta Visio 2003. Finalmente, detallamos la interfaz que permite especificar dichas asociaciones para un metamodelo específico.

En los trabajos relacionados hemos citado herramientas que soportan el modelado independiente de dominio y específico a dominio, con mecanismos de manipulación de modelos mucho más complejos que los que ofrece MOMENT. Además con la aproximación seguida, MOMENT se beneficia del entorno gráfico de herramientas CASE, evitando el coste de desarrollo que sería necesario invertir para obtener un editor gráfico de las mismas características. Siguiendo este enfoque, se está estudiando la posibilidad de desarrollar un plug-in para la herramienta Eclipse que se beneficie de su módulo Graphical Editor Framework [20] y que utilice MAUDE [21] como sistema de reescritura de términos para manipular artefactos software en MOMENT.

7. Bibliografía

1. Boronat A., Carsí J.A., Ramos I., *Una plataforma semántica para la gestión de modelos*. Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2003. Alicante, 2003.
2. Object Management Group, *Unified Modeling Language (UML) 1.4 draft*, February 2001. http://www.omg.org/cgi-bin/doc?ad/2001-02-11
3. Clavel M., Durán F., Eker S., Lincoln P., Martí-Oliet N., Meseguer J. y Quesada J., *Maude: Specification and programming in rewriting logic*. Theoretical Computer Science, 2001.
4. Wideman G., *Microsoft Visio 2003 Developer's Survival Pack*, 2004.
5. F# web site: http://research.microsoft.com/projects/ilx/fsharp.aspx
6. Diaconescu R., Futatsugi K., Ishisone M., Nakagawa A. T. y Sawada T. *An Overview of CafeObj*. In: C. Kirchner and H. Kirchner (eds), Proceedings of WRLA '98, Electronic Notes in Theoretical Computer Science, Vol. 13, 1998.
7. Object Management Group, *Meta-Object Facility (MOF) version 1.4*, April 2003. http://www.omg.org/technology/documents/formal/mof.htm
8. Bruno G., *Model Based Software Engineering*, Chapman & Hall, 1995.
9. Rational Rose Developer. Official web site: http://www-306.ibm.com/software/rational/
10. MS Office Visio 2003. Official web site: www.microsoft.com/office/visio/
11. ArgoUML. Official web site: http://argouml.tigris.org/
12. Rational Rose XDE Developer. Official web site: http://www-306.ibm.com/software/awdtools/developer/rosexde/
13. Eclipse Modeling Framework. Official web site: http://www.eclipse.org/emf/
14. Fernández J.L., *Una Propuesta de Formalización de la Arquitectura en Cuatro Capas de UML*. Tesis Doctoral, Departamento de Informática y Sistemas, Univ. de Murcia, 2001.
15. Sztipanovits J. y Karsai G., *Model-Integrated Computing*, Computer, Apr. 1997, pp. 90-92.
16. Ledeczi. A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason C., Nordstrom G., Sprinkle J. y P. Volgyesi. *The Generic Modeling Environment*. In Proc. Workshop on Intelligent Signal Processing. 2001.
17. Lara J., Vangheluwe H., Alfonseca M., *Using Meta-Modelling and Graph Grammars to create Modelling Environments*. Workshop on Graph Transformations and Visual Modellig Techniques GT-VMT at ICGT'2002. Barcelona, Octubre 2002. ENTCS Vol.72, (3).
18. Kelly S., Lyytinen K. y Rossi M., *Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment.* In Proceedings of CAiSE'96, LNCS 1080, Springer-Verlag, Heraklion, Creta, Grecia, Mayo 1996, pp. 1-21.
19. Beckett D.. *The Design and Implementation of the Redland RDF Application Framework*. Tenth International World Wide Web Conference. Mayo 2-5, 2001, Hong Kong. http://www10.org/cdrom/papers/490/
20. Graphical Editing Framework. Official web site: http://www.eclipse.org/gef
21. MAUDE. Official web site: http://maude.cs.uiuc.edu/
22. Boronat A. Carsí J. A., Ramos I., An Algebraic Baseline for Automatic Transformations in MDA. Workshop Software Evolution Through Transformations: Model-based vs. Implementation-level Solutions (SETra'04), 2nd International Conference on Graph Transformation (ICGT2004), ENTCS, Roma (Italia). Octubre 2004.

An Approach for Cross-Model Semantic Transformation on the .NET Framework

Artur Boronat, José Á. Carsí, Isidro Ramos, Julián Pedrós
Department of Information Systems and Computation
Technical University of Valencia
Camí de Vera s/n
46022 Valencia (Spain)
{aboronat | pcarsi | iramos | jpedros}@dsic.upv.es

ABSTRACT

Model-Driven Development is a suitable approach for improving productivity and quality in the software development process by raising the level of abstraction of software artifacts from code to models. In this context, code generation has traditionally been the star feature. Working on models also provides more reusable solutions to problems that have to be solved in an ad-hoc manner using the .NET technology: interoperability between applications, integration of applications, legacy system recovery, software evolution, maintainability, etc. One mechanism for dealing with models is model transformation. Although several tools follow this approach to generate code that targets the .NET platform, there are no tools based on .NET technology that provide model manipulation such as transformations. In this paper, we present a platform that permits the formal representation of models and an operator to transform models in a declarative way. This platform has been implemented using the F# functional programming language, presenting its advantages over an implementation using an imperative programming language such as C#. The platform has been integrated into the Visio modeling environment by means of an add-in to deal with formal models through visual metaphors (visual notation). To our knowledge, this solution is the first approach for dealing with cross-model semantic interoperability on the .NET technology.

Keywords

Model-driven development, model transformation, graphical notation, MS Visio 2003, F#, Office managed COM add-in, cross-language interoperability.

1. INTRODUCTION

Model-Driven Development (MDD for short) [Sel03] is a suitable approach to combat the complexity of software development by means of principles such as abstraction and modularity, which improve the quality, reuse, and scalability of software artifacts. This discipline also improves the productivity and quality in the software development process to obtain automatically error-free code that is easy to maintain. Following this approach, a software artifact is modeled at a high level of abstraction where technical details are not as important as semantics. The structure and semantics of a software artifact are modeled by using an ontology or metamodel (a vocabulary that provides constructs to specify a model in a determined manner). A metamodel can be domain-independent such as UML, or domain-specific, taking into account specific types of software systems, such as banks, electronical circuits, business modeling, etc.

In accordance with [Cza00], the MDD approach based on UML-like metamodels is called Object-Oriented Analysis and Design, while the MDD approach based on domain-specific metamodels is called Domain Engineering. Microsoft has shown a growing interest in the MDD discipline by adding designers to the Visual Studio environment in order to build software artifacts by means of models. This technology should be expanded to cover domains of interest to their customers, such as code visualization, business modeling, etc., thereby

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2005 *conference proceedings,*
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

Boronat, A., Carsí, J. A., Ramos, I., Pedrós, J. An Approach for Cross-Model Semantic Transformation on the . NET Framework. Proc. of the .NET Technologies'2005 Conference. 2005.

applying Domain Engineering from a commercial standpoint [Coo04].

In MDD, models defined by means of metamodels are usually transformed into code, providing the final application that can be directly compiled and executed on a specific platform, such as .NET. There are lots of tools that provide code generation based on models in the .NET world, called model compilers: from visual modeling environments (such as Visio, Rational XDE [Rat] among others) to development environments (such as the Visual Studio .NET).

However, generating code from models forces the programmer to work on code in order to face well-known problems in the software engineering field: round-trip, application integration, legacy system recovery, refactorization, software evolution, maintainability, etc. These problems can be solved at a more abstract level by dealing with models directly, obtaining the same advantages that the MDD discipline obtains for the software development process. This is the point where model transformations come into play [Sen03]. To provide support for model transformations, two main issues have to be taken into account: model representation to structure the information in some accessible manner and a transformation mechanism to manipulate such models. Although this issue is becoming well-known in the research field [Cza03], to our knowledge, there are no tools based on the .NET technology to achieve transformations of this kind. A solution of this nature would improve the productivity and the quality in the integration of .NET-based applications at a high level of abstraction, rather than just benefiting from the cross-language interoperability that the .NET Framework provides at code level. Therefore, a solution of this nature would achieve cross-model interoperability.

In this paper, we provide a solution along these lines. We present a mechanism that takes advantage of the MS Visio modeling tool in order to describe the structure of visual models in a formal manner. This mechanism uses a platform to represent and store software artifacts in four layers, where metamodels and models are taken into account. Models defined on the platform can be transformed in a declarative fashion by using the platform operator *generate*, which permits the translation of a model between different metamodels.

This platform has been developed using the functional language F# [Fsh]. Taking into account its advantages over conventional OO languages such as C#, models are formally described in an algebraic fashion.

Our solution takes advantage of the .NET cross-language interoperability and the Office extension mechanism by means of managed COM add-ins. It extends the Visio tool by using the most suitable language in each context: F# to implement the definition of formal models and their manipulation, and C# to integrate this functionality into the Visio tool.

The structure of the paper is as follows: in Section 2, we discuss and compare the C# and F# programming languages, evaluating their suitability in our solution; Section 3 presents a platform that enables the definition of models in an algebraic fashion; Section 4 presents the add-in that integrates this platform into the Visio modeling environment, enabling the manipulation of formal models by means of graphical metaphors (graphical notation); Section 5 describes the F# definition of the model transformation mechanism that is provided by the platform; finally, Section 6 summarizes our contributions.

2. F# versus C#

F# is a functional programming language that targets the .NET platform. F# has been developed at MS Research Cambridge and is a version of the Caml programming language [Cah00], which belongs to the ML languages family. F# is well integrated in the Visual Studio environment[1] and provides certain features that are inherited from Caml, which make it interesting for our purposes.

F# is based on the lambda-calculus model [Rea93] by means of a strict (eager) evaluation strategy. Therefore, it permits the definition of a program independently from the evaluation strategy used, that is without mixing functionality and control logic as is necessary in C#.

F# provides richer constructs to declare types like sum types, among others. A sum type permits the definition of a type by means of constructor patterns, each of which may have arguments. Sum types allow us to describe the signature of an algebraic specification [Ehr85], where the name of the type is the sort, and the constructor patterns are the constructors of a sort. This comparison allows us to deal with models from an algebraic point of view, where semantics of models can be described formally by means of Abtract Data Types. This feature is not feasible in C# intuitively, although it can be simulated in the same way that such constructors are invoked from C# code by means of static methods.

[1] Although we used F# version 0.6.4.1 for this solution.

F# provides a conditional pattern matching mechanism that enables the definition of functions over sum types in an intuitive way by applying a pattern to each constructor in order to perform a task. This mechanism can be simulated in a more complex way in C# by means of the *switch* statement and the addition of *if* statements inside each *case* of the *switch* statement.

The F# compiler infers the types of the declaration of a function statically (the types of its arguments and the type of its closure, i.e. the type of the returned value), so that these types do not have to be indicated in the definition of the function. This feature makes the definition of F# programs easier.

As all values are functions in F#, we can use lists of functions whenever we need them, rather than using delegates, as it happens in C#. This also provides parametric polymorphism that is used to provide some parametric functions that deal with lists without knowing the types of their elements: *map* to apply a function to the elements of a list, *find* to search the elements of a list that validate a condition, *exists* to know if some elements of a list validate a condition, etc. This feature, called generics, has not been added to the current release of the .NET Framework, although it will be added to the next release [Yu04].

Although F# is a functional language, it also provides imperative features such as references (pointer to a value), which allow us to manipulate the memory state whenever necessary for the sake of efficiency. Furthermore, the last and the most important feature of F# is its full interoperability with languages that target the .NET platform, such as C#, by means of the ILX extensions [Sym01] to the IL language.

All these considerations have encouraged us to use F# for the implementation of our solution to deal with models from a formal standpoint, on the grounds that we can use C# to integrate our solution to tools based on the .NET technology, such as the Visio modeling environment.

3. ALGEBRAIC REPRESENTATION OF MODELS BY MEANS OF F#

Our approach constitutes a platform that uses several metadata layers to describe any kind of information. In our work, we consider software artifacts in four abstract layers (as shown in Figure 1):

- The M0-layer collects the examples of all the models, i.e., it holds the information that is described by a data model of the M1-layer.
- The M1-layer contains the metadata that describes data in the M0-layer and aggregates it by means of models. This layer provides services to collect examples of a reality in the lowest layer.
- The M2-layer contains the descriptions (metamodels) that define the structure and semantics of the models located at the M1-layer. A metamodel is an "abstract language" that describes different kinds of data.
- The M3-layer is the platform core, containing services to specify any metamodel with the same common representation mechanism. It is the most abstract layer in the platform. It contains the description of the structure and the semantics for metamodels. This layer provides the "abstract language" to define different kinds of metadata.

The core of the prototype is an algebra that provides a set of sorts and constructors to define models and a set of operators to manipulate them. To implement this algebra, we have used the F# programming language for two main reasons: to bring a formal model transformation approach closer to an industrial programming environment, such as .NET, and to benefit from the functional programming advantages presented above.

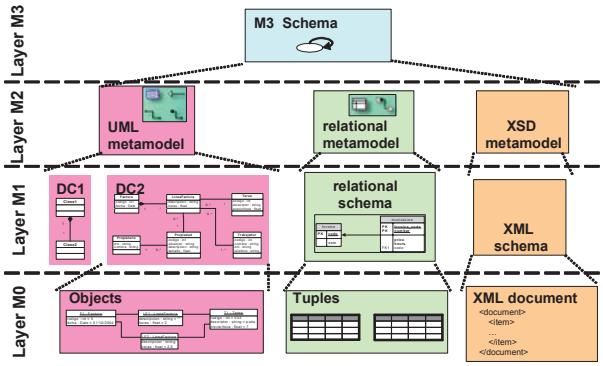


Figure 1. Graphical representation of the four-layered platform.

An Algebra for Representing Models

The algebra aims to represent models of any kind as algebraic terms in order to automate model transformation tasks in a precise, formal way. Achieving this objective implies choosing a basic specification language that permits us to describe any piece of data.

We have developed a platform based on this algebra that permits the representation of software artifacts in the four meta-layers explained above. Four main sorts permit the definition of a model as a term in the algebra:

1. *Concept*

   A concept represents an entity that can be described by means of properties. The constructor of this sort is defined in F# notation as follows:

```
Concept = NilConcept
        | Concept of (Concept * string)
```

where *NilConcept* represents a null concept term; the first argument of the constructor *Concept* is a term of the sort *Concept* that represents its metaconcept in the next upper abstraction layer, and the second argument is its identifier.

2. *Property*

   A property is a relationship that relates either a concept or a property (subject of the property) to a concept (the object of the property), following the RDF philosophy to describe metadata [W3C]. Such relationships are specified by means of the Property sort.

   We express the constructor of this sort in F# notation as follows:

```
Property = NilProperty
    | Property of (Property * string * Cardinality *
    Cardinality * Node * Concept)
```

where *NilProperty* represents the null property term and the arguments of the constructor *Property* are the following elements in order of appearance:

- Parent property indicating its type.
- Identifier of the property.
- Minimum cardinality of the property that indicates the minimum amount of instances of the range concept, which must be related to the subject node.
- Maximum cardinality of the property that indicates the maximum amount of instances of the range concept that can be related to the subject node.
- Subject element that receives the property. This can be a concept or another property, because a property may involve other properties.
- Object element that constitutes the value of the property. A property cannot be the object of another property on the grounds that it does not provide additional information.

3. *Schema*

   In our context, a schema term represents a collection of concepts and properties that describe such concepts.

4. *Level*

   A level term represents a layer in the platform. Four terms of this sort constitute the four-layer structure of the platform. The term M3-layer represents the most abstract layer in the platform and contains a basic vocabulary to define metamodels at the M2-layer, i.e. a simplified meta-metamodel. This schema contains the term Concept and the term Property; the latter relates two concept terms, constituting the minimal structure that we use to represent a model at a lower layer. The four layers of the platform are defined as values that can be accessed by means of references (pointers to a value). This simplifies the definition of transformation rules and enhances efficiency.

For instance, the *Relational Metamodel* is a schema term that contains the concepts and properties that constitute the terminology to define a relational schema, as shown in Figure 2. For instance, *Table* and *Column* are represented by means of concept terms, which are related to each other through a property *table/Column* in the relational metamodel at the M2-layer. This metamodel allows the definition of the concept *Invoice* as a table. In an identical way, the concept *Code* is defined as a column, which is related to the table *Invoice* by means of an instance of the property *table/Column*, i.e. by means of the invoice/Code property.

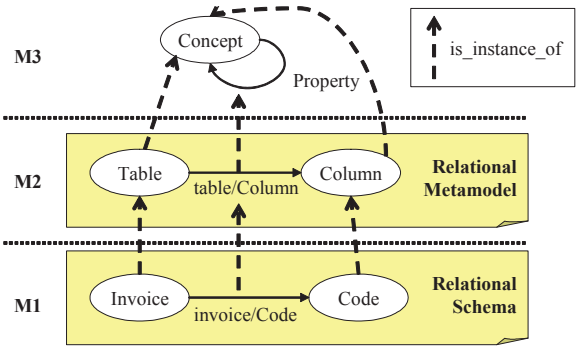


Figure 2. Definition of metamodels and models on the platform.

4. VISIO AS A VISUAL ENVIRONMENT FOR DEALING WITH ALGEBRAIC MODELS

Taking into account the four-layered platform based on the functional implementation of the presented algebra, we have developed an add-in for MS Visio 2003, called *Platform Integrator*. This add-in permits the association of a graphic metaphor with a formal metamodel in the Visio modelling environment and the automatic definition of its models as algebraic terms.

The customization of the Visio modeling visual environment is performed by means of add-ons, i.e. sets of stencils that provide the graphical information needed to define the graphical notation for a metamodel. To extend the tool, a type of module, called an add-in, is used to add functionality. Given the easy extension that such add-ins provide by means of managed COM (Component Object Model), Visio is the selected tool to embed our model repository. The formal definition of models, which our add-in provides, allows us to transform models as we present in the following section, rather than merely defining the models graphically.

Outside the Add-in

The add-in architecture is divided into three layers: the interface that graphically represents metamodels and models; the middle layer that permits the association of such graphics to algebraic representation of models; and the persistence layer that stores all the information.

In the middle layer, the module *Platform Integrator* enables the definition of associations between the graphical elements of the interface of Visio and the algebraic terms that define software artifacts in the four-layered platform. Such associations are stored in the same platform as instances of UML classes at the M0-layer by means of the *UMLSupport* library.

The persistence layer consists of two types of storage units: the one provided by Visio and the one provided by model repository of the platform.

In Visio, graphical models are stored by means of two types of files: .vss files that store the model defined in the shapesheet, and .vst files that provide the templates with masters (stencils), which enable the definition of shapes in the shapesheet. Visio provides several templates with several kinds of masters to define a large variety of models by default. Nevertheless, a user can define new templates to define other types of models.

The four-layered platform stores the information in a RDF repository on the grounds that the concepts and properties used in the platform are equivalent to RDF resources and properties, respectively. The repository used is Redland [Bec01], which we have embedded in a visual studio project and compiled on the .NET platform by means of the managed C++ programming language. In this repository, we store schemas that belong to any layer of the platform, and we store associations between graphical elements of the modeling environment and algebra terms.

Inside the Add-in

The graphical elements of the Visio interface are related to platform elements by means of the module *Platform Integrator*. To present both the definition of a graphical metaphor related to a metamodel and the definition of formal models by means of this association, we focus on the M2-layer and the M1-layer of the platform. These layers store information of metamodels and models, respectively. In this way, a schema of the M2-layer is related to a Visio stencil, while a schema of the M1-layer is related to a Visio shapesheet. To graphically represent the concepts and the properties that constitute a metamodel at the M2-layer, we use the masters that define the chosen stencil. In the case of the graphical representation of elements that constitute a model at the M1-layer, we use shapes that are defined by means of masters of a stencil.

The association mechanism that relates a formal model to a graphical representation has been modelled in Figure 3 using UML notation. In this model, the *SchemaWrapper* class contains the information needed to relate a schema to its visual representation, while the class *NodeWrapper* contains the information that relates a concept or a property to a specific image. Specializations of both classes identify whether a schema is a metamodel (*GraphicViewWrapper*) or a model (*GraphicModelWrapper*), and whether either a concept or a property is a metamodel element (*GraphicPrimitiveWrapper*) or a model element (*PictureWrapper*). In the case of a metamodel, an instance of the *GraphicViewWrapper* class relates a schema of the M2-layer to a stencil, and an instance of *GraphicPrimitiveWrapper* class relates a node of the schema to a master. In the case of a model, an instance of the *GraphicModelWrapper* class relates a schema of the M1-layer to a shapesheet, and an instance of the *PictureWrapper* class relates a node of the schema to a shape.

Storage of UML Software Artifacts

The association between Visio graphical elements and platform elements (defined in the UML class diagram in Figure 3) is stored in the same four-layered platform. In this way, the platform is used as an object-oriented repository on the grounds that it enables both the definition of UML models at the M1-layer and the definition of their instances at the M0-layer.

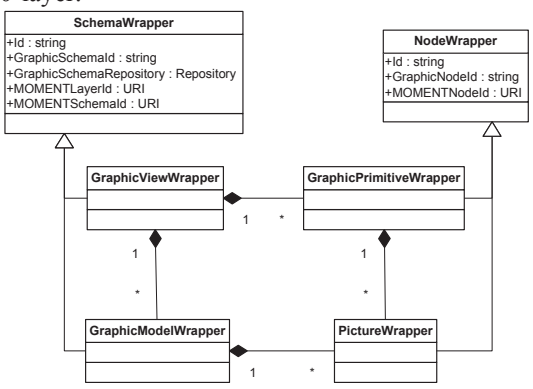


Figure 3. UML Model of the association mechanism between graphical elements and algebraic terms.

To achieve this, we have specified part of the UML metamodel as a schema at the M2-layer of the platform, taking into account classes and associations. The class diagram in Figure 3 has been specified in a schema at the M1-layer of the platform as an instance of this UML metamodel. Therefore, to

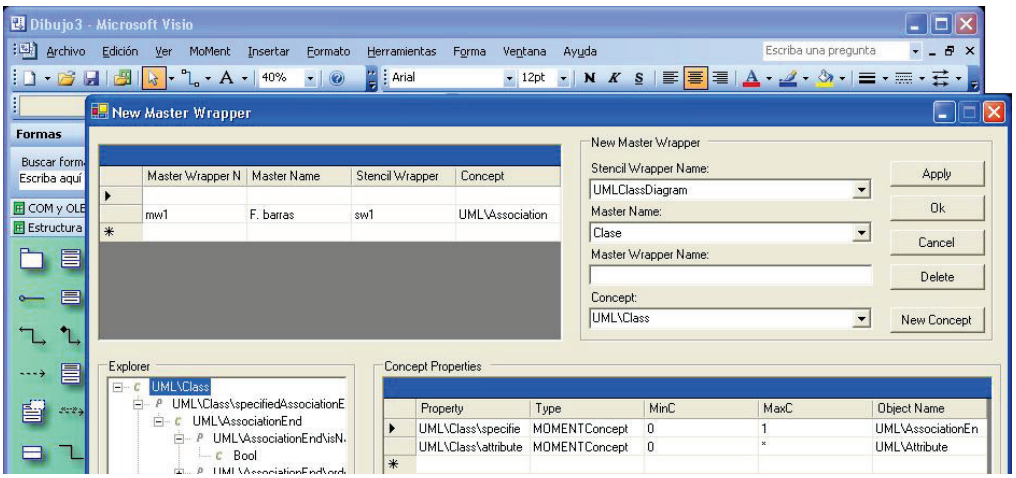


Figure 4. Interface to graphically define concepts and properties of a metamodel.

define associations between elements of the platform and Visio graphical elements, a schema can be defined at the M0-layer of the platform by instantiating the classes that constitute the model at the M1-layer.

Definition of Graphic Metaphors for Metamodels

To define a metamodel by means of Visio, we associate a schema of the M2-layer of the platform to a stencil. Then, each of its masters is related to a node of the schema by means of the interface in Figure 4, completing the graphical metaphor related to the metamodel. The formal metamodel can be directly defined on the platform by means of the Visio interface; it can also be loaded from the platform.

To define a master in the interface shown in the Figure, an association between a metamodel of the platform and a stencil must be selected. Then, a master of the stencil and a node of the schema are selected and related by means of a new association. After this, the hidden properties of a node (i.e. the properties that are not related to a master directly) can be accessed. They are shown in the list that appears at the bottom of the interface, and they can also be navigated recursively by means of the tree, placed on the left part of the interface.

Once a metamodel is graphically defined in Visio, it can be used to define visual models by means of the *drag-and-drop* mechanism, by dropping masters of the stencil onto the *shapesheet*. We enrich this mechanism so that the shape is not only graphically defined in the *shapesheet* but also its contents are defined in the platform. This functionality is embedded in the *Platform Integrator* module, providing this functionality automatically in a transparent manner to the user. Thereby, we not only define a graphical model but also provide the semantic information related to a metamodel.

Therefore, we can define formal models in a visual manner so that they can be manipulated by means of transformations as we explain in Section 5.

5. TRANSFORMATIONS

The operator *generate* permits the translation of a model of a specific metamodel into a model of a different metamodel. The semantics of the operator is defined denotationally by means of the pattern matching mechanism of F#.

The operation *generate* defines an evaluation strategy operationally (by means of the pattern matching mechanism) in order to enable the definition of transformation rules in a declarative manner. In this way, transformation rules are defined like axioms that do not take into account the rule evaluation strategy embedded in the operator *generate*. In this section, we introduce the *likeness* relation[2], which enables the definition of transformation rules based on metamodels. Then, we describe the structure of a transformation rule. Finally, we define the operational semantics of the operator *generate*.

Semantical Relationships between Metamodels

In our approach, transformations are based on metamodels. Applying a transformation to a source model involves two metamodels: a source metamodel that describes the structure and semantics of the source model, and a target metamodel that provides the structure and semantics for the new model to be generated. Two types of transformations can be distinguished taking into account the target metamodel:

[2] We have chosen the name *likeness* instead of *equivalence,* on the grounds that the equivalence relation is defined between elements of different metamodels, which cannot be equal.

– Intra-metamodel: when both the target and the source metamodels are the same. In this paper, we do not discuss this type of transformation.
– Inter-metamodel: when both the target and the source metamodels are different.

By basing our transformations on metamodels, we can specify them from an abstract point of view taking into account the metainformation that constitutes both the source and the target metamodels. This way, transformation rules can be viewed as patterns that can apply to any model of the source metamodel. To define such rules, we introduce the *likeness* relation between elements of two metamodels.

The *likeness* relation is based on mappings between the elements of both the source and the target metamodels. A mapping is a property that is an instance of a property, called *Likeness*, defined at the M3-layer. A mapping relates a concept of the source metamodel to a concept of the target metamodel. For example, indicating that a Table in the relational metamodel is like a Class in the UML metamodel. A set of mappings conforms a *likeness* relationship indicating that the concepts, which participate in these mappings, represent a similar semantic meaning in their respective metamodels.

A *likeness* relationship between elements of two metamodels may involve more than one element of either the source or the target metamodels. Thus, we distinguish between:

– Simple *likeness* relationships: specified by means of only one mapping.
– Complex *likeness* relationships: specified by a set of mappings involving several elements from either the source or the target metamodels. For example, to define an equivalence relationship between a foreign key of the relational metamodel and an aggregation of the UML metamodel, we have to relate the foreign key, the unique constraint and the not null value constraint concepts to the aggregation concept. This is because these three concepts of the relational metamodel provide the necessary knowledge to define an aggregation between two classes in the UML metamodel, such as the cardinalities of the aggregation.

A *likeness* relation between two metamodels is defined as the union of all *likeness* relationships established between the elements of both metamodels. As a first approach to provide cross-model semantic interoperability on the .NET platform, we only focus on simple *likeness* relationships.

Transformation Rules

The operator *generate* is applied to a schema of the source metamodel and defines a new schema of the target metamodel. To achieve the transformation, this operator is based on a *likeness* relation defined between both the source and the target metamodels. By means of this relation, the operator knows what should be generated from a set of concepts of a source model.

To process the elements of the source schema, the operator *generate* makes use of transformation rules defined declaratively. Each one of them is divided into two functions: a condition and a body. When a group of elements of the source model is processed, the condition checks their properties in order to select which generation function should be applied. These conditions take into account the order of precedence that exists between the concepts of a specific metamodel when this order is used to define a model. For instance, when we define a relational schema, we cannot define a column if the table that it belongs to is not defined previously. On the other hand, the body of the transformation rule involves the definition of concepts and properties in the target schema.

The operator *generate* automatically generates models among different metamodels taking into account a set of transformation rules, which are applied following a specific evaluation strategy. The set of transformation rules is defined independently of the evaluation strategy chosen. To achieve this issue, all the transformation rules must have the same declaration so that the operator *generate* knows how to apply them. By declaration, we mean the declaration expression of a function in a F# interface (.mli), which involves the symbol that identifies the function value, the types of the arguments and the type of the function value (i.e. the type of the closure). Therefore, we denote the declaration expression of a function as follows:

```
val function_name :
    arg1_type -> ... -> argn_type
-> closure_type
```

This is the value declaration inferred statically by the compiler where *val* is a reserved construct that indicates the declaration of a value; *function_name* is the symbol that identifies the function; *arg1_type -> ... -> argn_type* are the types of the argument list of the function; and *closure_type* is the type of the closure (which is viewed as the type of the returned value in imperative programming).

As a transformation rule is divided into a condition function and a body function, we present their declarations in the following subsections.

5.1.1 Condition function

A condition function is the mechanism that indicates when a transformation rule can be applied. There is one, and only one, condition function for each function body. This means that a condition can indicate the suitability of only one transformation rule in a specific context, although many transformation rules can be applied to the same group of elements of a source model. The declaration expression for a condition function is as follows:

```
val condition_name : Schema -> Node
-> bool
```

where *condition_name* is a symbol that identifies the condition, the first argument is the source schema to be translated, and the second argument is the current node of the source schema to be translated.

The condition function checks wheter or not the specified node validates a set of requirements in order to determine if it can be translated into the target schema by means of the body function of the transformation rule. Finally, the closure of the condition function is a Boolean value indicating the suitability of the transformation rule that contains the condition.

5.1.2 Body function

The body function of a transformation rule materializes a *likeness* relationship, defined between elements of both the source and the target metamodels. This materialization involves both the definition of new elements into the target schema and specific mappings between the elements of the source schema, which are involved in the transformation rule. Such mappings provide support for traceability. Given a transformation process between two models, traceability [Got94] enables the identification of elements that are related by means of the application of a transformation rule and that belong to different models. Traceability support enhances mechanisms such as change propagation and round-trip between models.

To explain the semantics of the operator *generate*, we use the generation of a UML model from a relational schema as an example. The materialization of a *likeness* relationship at the M1-layer (between models) is obtained by four steps, shown in Figure 5:

1. The concept is reified in its metaconcept; that is, if the concept to be processed is the table *Invoice*, we obtain the metaconcepts *Table* of the relational metamodel.
2. Once we know the corresponding metaconcept of the source metamodel, we navigate the *likeness* relationship that relates it to a concept of the target metamodel. In the case of a table of the *Relational Metamodel*, we obtain the concept *Class* of the *OO Metamodel*.
3. The operator *generate* instantiates the concept of the target metamodel, which becomes a metaconcept for its instance, i.e., the concept *Class* of the *OO metamodel* becomes the metaconcept for its instance *OO-Invoice*. The new concept, which has been generated in the new target schema at the M1-layer, is similar to the original concept in step 1, through the *likeness* relationship that we have defined before.
4. Finally, the operator instantiates the *likeness* relationship defined at the M2-layer between the *Metaconcept* of the source *Concept* and the *Metaconcept'* of the new generated *Concept'*. The instantiation defines a new mapping in the traceability schema at the M1-layer, which is a property that has the source *Concept* as domain and the target *Concept'* as range.

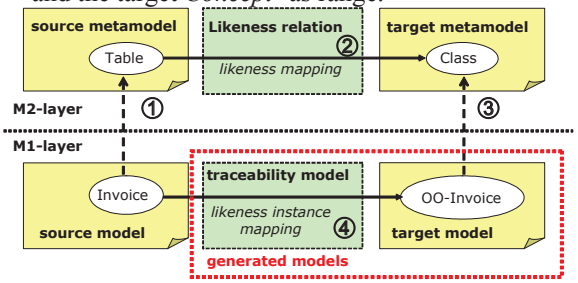


Figure 5. Description of the transformation process

The declaration of a body function is as follows:

```
val body_name : Schema -> Node
-> unit
```

where *body_name* is the symbol that identifies the body function, the first argument is a schema that contains the specific mappings between elements of both the source and the target models, and a node is the element of the source model to be translated. A body function knows the source and the target models by means of the mapping schema, which contains this information.

The type of the closure of a body function applied to a mapping schema and a node is the unit type[3]. A body function carries out side effects by accessing the layers of the platform (term of type Level) by means of references to them. Although these side effects decrease the level of abstraction of our functional approach, they avoid having to pass a whole layer as an argument for each transformation rule in order to improve efficiency. Side effects

[3] This type describes a set which possesses only a single element, which is denoted by (). This means that this function simulates the notion of procedure, just as the type *void* does in the C language.

produced by a body function involve the insertion of new elements into the target schema and new mappings into the mapping schema, both of which are located at the M1-layer. To understand the application of a transformation rule in more detail see [Bor04].

The Operator *generate*

The operator *generate* carries out the evaluation of a set of transformation rules on a source model, which is defined at the M1-layer of the platform. This obtains a new target model and a traceability model between the elements of the source models and the elements of the new generated model. The generated models are also defined at the M1-layer, as shown in Figure 5. The operator *generate* is a function whose declaration is as follows:

```
val generate :
    (bool * unit) list ->
    string -> string -> string
-> bool
```

where the first argument is a list of pairs of functions, in which the first element is a condition function and the second is a body function (i.e. each pair is a transformation rule); the second argument is the name of the source model placed at the M1-layer; the third argument is the name of the schema that contains the *likeness* relationship between the source and the target metamodels (i.e. the schema that provides the *likeness* relation); and the fourth is the name of the new target schema to be generated. This function returns a Boolean value indicating whether or not the model transformation has been performed correctly.

The evaluation process carried out by the operator *generate* is split into three steps: initialization of new schemas at the M1-layer, solution search, and transformation.

First, the operator defines two empty schemas at the M1-layer of the platform:

- Definition of the traceability model.
- Definition of an empty target schema as instance of the target metamodel with the name specified as the fourth argument. The target metamodel is known by means of the model of *likeness* mappings of the M2-layer, which is specified as the third argument.

Second, the operator searches for a solution for the source model transformation. This solution consists of a list of ordered nodes of the source model. The application of transformation rules to the ordered nodes produces the target schema. This step is needed because F# does not provide any mechanism to support the evaluation of the nodes of a schema in an automated and intuitive way. This inconvenience is due to the definition of a schema as a list of nodes, or even as a set. In other languages, this problem is avoided by means of a backtracking mechanism, such as in CLIPS [Cli], or by means of the commutativity property, such as in the algebraic language Maude [Cla02]. A solution is reached when all the nodes of the source model, whose parent participates in a *likeness* relationship in the specified *likeness* relation, have been added to the solution list. In the case that no solution is found, the transformation process is stopped and the operator returns a false value.

```
let rec apply_solution list_solution_concepts
list_transformation_rules sch_m1_source
sch_m1_mappings =
    match list_solution_concepts with
    | [] -> true
    | h::t ->
        let _ = apply_axiom
            list_transformation_rules
                sch_m1_source sch_m1_mappings h
        in
            apply_solution t list_transforamtion_rules
                sch_m1_source sch_m1_mappings
```

Figure 6. F# definition of the apply_axiom function.

Last, the list of transformation rules given as first argument is applied to the nodes of the solution list provided by the second step. The application of transformation rules is reached by means of the *apply_solution* function, which uses the pattern matching mechanism of the F# programming language, as shown in the code in Figure 6. The *apply_solution* function is recursive (indicated by the construct *rec*) and applies the *apply_rule* function to the first node h (head) of the list. This function searches for a suitable transformation rule in the list by means of its respective condition function, and applies the body of the rule to the node h. It inserts a set of nodes into the target schema and inserts the corresponding mappings into the traceability schema. To transform the entire list of nodes of the source model, the *apply_solution* function is applied to the rest of nodes of the solution list t (tail) recursively. When no node is left, the transformation is concluded.

The function *generate* produces side effects due to the application of transformation rules to the elements of the source model. These side effects are changes to the state of the M1-layer, which involve the addition of the generated target model and the traceability model to the M1-layer.

6. CONCLUSION

In this paper, we have presented a solution for transforming models by means of the Visio modeling

environment following a MDD approach. To achieve this, we built a platform that permits the definition of software artifacts following a four-layered approach, which involves metamodels and models, from an algebraic point of view. The platform provides a mechanism to transform models in a declarative way between two metamodels. This mechanism is embodied by the operator generate that receives a list of transformation rules that are applied to a source model in order to translate it into a model of a target metamodel. The application of a transformation provides support for traceability between the source model and the generated one.

The platform has been implemented with the F# programming language. We have also discussed its advantages over other languages that target the .NET platform, such as C#.

The platform has been integrated into the Visio modeling environment by means of an Office managed COM add-in. This allows us to deal with formal models in a visual manner through graphical metaphors. The platform also acts as a repository of formal models. This feature has been used to store the associations between the graphical elements of the Visio interface with the formal definitions stored in the platform, in a UML-based manner.

To our knowledge, this is the first approach to support cross-model semantic interoperability from a modeling environment based on .NET technology.

7. ACKNOWLEDGEMENTS

This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01, and the National Project PBC-03-00 "Methodologies of dynamic user interfaces development".

8. REFERENCES

[Bec01] Beckett, D. The Design and Implementation of the Redland RDF Application Framework. 10th WWW Conf. May 2-5, 2001, Hong Kong.

[Bor04] Boronat, A., Ramos, I., Carsí, J. Á. Automatic Model Generation in Model Management. Springer-Verlag GMBH. Proceedings of CIT 2004. India, 2004.

[Cah00] Cahilloux, E., Manoury, P., Pagano, B.: Developing Applications With Objective Caml, Éditions O'Reilly, 2000.

[Cla02] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F. Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.

[Cli] Clips documentation. http://www.cis.ksu.edu/ VirtualHelp/Info/clips.html

[Coo04] Cook, S. Domain-Specific Modeling and Model Driven Architecture, MDA Journal, January 2004

[Cza00] Czarnecki, K. and Eisenecker, U.W. Generative Programming: Methods, Tools, and Applications. Addison Wesley, Boston, 2000.

[Cza03] Czarnecki, K., Helsen, S. Classification of Model Transformation Approaches. In Proceedings OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Ehr85] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Springer-Verlag Berlin Heidelberg New York Tokio (1985). ISBN: 3-540-13718-1.

[Fsh] Fsharp home page. http://research.microsoft. com/ projects/ilx/fsharp.aspx

[Got94] Gotel, O. C. Z., Finkelstein, A. C. W. An Analysis of the Requirements Traceability Problem. In Proceedings of First International Conference on Requirements Engineering (ICRE). Colorado, USA. 1994.

[Rat] Rational Rose XDE Developer. http://www-306.ibm.com/software/awdtools/developer/rosexde/

[Rea93] Reade, C. Elements of Functional Programming. Addison-Wesley, 1993.

[Sel03] Selic, B.: The Pragmatics of Model-Driven Development. *IEEE Software*, ISSN 0740-7459. September 2003, pp. 19-25.

[Sen03] Sendall, S., Kozaczynski, W. Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software. September/October 2003 (Vol. 20, No. 5), pp. 42-45.

[Sym01] Syme, D. ILX: Extending the .NET Common IL for Functional Language Interoperability. In Proceedings of Workshop Babel 01, Florence, Italy. September 2001.

[Wid04] Wideman G., *Microsoft Visio 2003 Developer's Survival Pack*, 2004.

[W3C] W3C, Resource Description Framework (RDF), http://www.w3.org/RDF/

[Yu04] Yu, D., Kennedy, A., Syme, D.. Formalization of Generics for the .NET Common Language Runtime. In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy, January 2004.

Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos

A. Boronat, J. Pedrós, J. Á. Carsí e I. Ramos

***Resumen*-- El modelado específico a dominio se está consolidando como una forma de desarrollo de software con herramientas como GME (Generic Metamodel Environment) y MetaEdit+, entre otras. La clave del éxito de estas herramientas es la proximidad de la ontología de modelado al problema que intenta solucionar el usuario analista y/o programador, de manera que los modelos definidos son fácilmente interpretables en el dominio del problema. MOMENT es una herramienta de gestión de modelos que permite definir este tipo de modelos siguiendo una aproximación algebraica. Con el fin de obtener una herramienta de gestión de modelos que utilice metáforas gráficas cercanas al usuario, se presenta una arquitectura que integra MOMENT en herramientas CASE que posean un entorno gráfico, pudiendo así utilizar sus capacidades visuales en la definición de modelos soportados por MOMENT. Con tal fin se indican los elementos gráficos de una herramienta CASE que se asocian con elementos de un metamodelo definido en MOMENT y cómo se almacenan esas asociaciones en el propio repositorio de MOMENT. Se ha realizado una implementación de esta arquitectura para una herramienta CASE concreta, y se presenta cómo se define la vista gráfica de un metamodelo definido algebraicamente utilizando su interfaz.**

***Palabras clave* -- Software engineering, Software maintenance, Software management, Software reusability, Data models, System modeling, Graphical user interfaces, Iconic languages**

I. Introducción

El desarrollo de software implica el estudio del sistema de información que constituye el problema a resolver (espacio del problema), las tecnologías que permiten implementar una solución (espacio de la solución) y el proceso de desarrollo del software que obtiene una solución tecnológica a los requisitos del sistema de información.

Desde el punto de vista de la solución, los lenguajes de programación han ido incrementado paulatinamente su nivel de abstracción con el objetivo de aproximarse al espacio del problema. Así pues del lenguaje ensamblador se pasó a los lenguajes procedurales, posteriormente a la tecnología orientada a objeto, desembocando en los lenguajes orientados a aspectos y componentes en estos últimos años. Estos esfuerzos han permitido mejorar el proceso de desarrollo del software incrementando la interoperabilidad, la reutilización de artefactos software y disminuyendo los costes de desarrollo. No obstante, aún no se ha conseguido alcanzar un paradigma que permita representar cualquier problema y que pueda generar su solución de forma automática. Además, difícilmente se alcanzará este punto, puesto que cada dominio dispone de unas propiedades específicas, que se deben tener en cuenta para poder definir el problema con precisión. El hecho de conseguir una tecnología universal suficientemente abstracta como para solucionar cualquier problema equivaldría a afirmar que la lengua "esperanto" será una especie de *lingua franca* en la que nos expresaremos los seres humanos en el futuro. Y esta situación parece improbable puesto que cada lengua posee su riqueza expresiva en el contexto socio-cultural en el que es usada, al igual que cada dominio de problema necesita mantener su vocabulario específico.

Desde el punto de vista del dominio, la Ingeniería del Software también ha conocido una serie de técnicas de modelado que permiten representar el problema aproximándolo hacia el espacio de la solución. Las técnicas de modelado se pueden clasificar en dos tipos, tomando como criterio de decisión la proximidad de la metáfora del modelo respecto al dominio del problema: modelado independiente de dominio y modelado específico a dominio. El modelado independiente de dominio intenta representar cualquier problema mediante una notación gráfica basada en un paradigma determinado, como por ejemplo UML [1] que se centra en el paradigma orientado a objeto. El modelado específico a dominio permite utilizar los conceptos del propio dominio para especificarlo. El problema en ambas aproximaciones es la falta de un soporte formal que garantice un proceso unívoco que permita obtener un producto software consistente y eficiente, y que permita validar los requisitos.

MOMENT es una plataforma [2] que permite definir modelos como especificaciones algebraicas de una teoría expresada en lógica ecuacional condicional con el fin de manipularlos mediante operadores genéricos. La semántica operacional de estas especificaciones algebraicas se evalúa mediante un sistema de reescritura de términos, lo que permite soportar las siguientes aplicaciones [3]: ejecución (animación) de especificaciones formales, análisis *narrowing*, *model checking* y demostraciones formales, entre otras.

A pesar que el uso de formalismos aporta propiedades ventajosas al proceso de desarrollo del software, usualmente

Este artículo ha sido financiado por el Proyecto Nacional de Investigación, Desarrollo e Innovación DYNAMICA TIC 2003-07804-C05-01 y el Proyecto Nacional PBC-03-00 de "Metodologías de desarrollo de interfaces de usuario dinámicas".

A. Boronat, J. Pedrós, J. Á. Carsí e I. Ramos pertenecen al Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia, Camí de Vera s/n, 46022, Valencia (España) (e-mails: {aboronat | jpedros | pcarsi | iramos}@dsic.upv.es).

Boronat, A., Boronat, A., Carsí, J. A.,Ramos, I. Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos. Revista IEEE América Latina, Special Edition - JISBD'2004. 2005.

no han tenido éxito fuera del ámbito investigador. Esto es debido al grado de preparación teórica que necesitan los usuarios de herramientas formales.

En este artículo, se presenta un enfoque genérico para integrar MOMENT en una herramienta CASE con un entorno gráfico, utilizando su interfaz como mecanismo para asociar una metáfora visual a un metamodelo descrito mediante una especificación algebraica. Como herramienta CASE para realizar la primera implementación de la arquitectura presentada se ha elegido MS Visio [4], porque proporciona buenas propiedades de extensibilidad, tanto en las facilidades gráficas, ya que dispone de un editor gráfico, como en la funcionalidad interna. De esta manera, el usuario puede especificar modelos de forma algebraica basándose en una metáfora visual conocida a través de una interfaz sencilla.

En esta sección se ha introducido la problemática tradicional existente en el proceso de desarrollo del software que constituye la motivación de nuestro trabajo y hemos presentado nuestro objetivo. En el apartado 2 se presenta un contexto de aplicación de la herramienta MOMENT en el cual se proporciona soporte automático, formal y genérico a la transformación de modelos. En el apartado 3 se indica la estructura de la plataforma de gestión de modelos MOMENT y los elementos principales que se utilizan en ella. En el apartado 4, se describe los elementos principales del entorno gráfico de una herramienta visual de modelado. En el apartado 5, se indica la arquitectura que hemos utilizado para añadir el soporte formal a una herramienta CASE, identificando las asociaciones entre los elementos de su editor gráfico y los *sorts* (o nombres de tipos) de la teoría algebraica utilizada en MOMENT, y se describe el mecanismo que permite asociar una metáfora gráfica a un metamodelo especificado algebraicamente en MOMENT. En el apartado 6 se discuten trabajos relacionados, y finalmente, en el apartado 7, se resumen una serie de conclusiones y trabajos futuros.

II. Contexto

La gestión de modelos [5] es una disciplina que persigue la manipulación genérica de artefactos software, entendidos como una estructura de datos definida a partir de un determinado metamodelo. Por ejemplo, un documento XML, un esquema relacional, un diagrama de clases, la especificación arquitectónica de un componente software, etc.

La manipulación de este tipo de artefactos se realiza mediante una serie de operadores que son aplicables sobre los modelos de forma independiente al metamodelo utilizado para su definición. Además estos operadores deben proporcionar la propiedad de composición de manera que el resultado de la aplicación de un operador sobre un determinado modelo debe poder ser utilizado como entrada para otro operador. De esta manera, la gestión de modelos constituye una aproximación genérica y reutilizable para dar soporte a las transformaciones automáticas de modelos en el contexto del desarrollo de software guiado por modelos, como se propone en la iniciativa Model-Driven Architecture [6] de OMG.

La plataforma MOMENT ofrece una serie de operadores genéricos aplicables a modelos como la unión, la intersección, la diferencia y la transformación de modelos, entre otros. Estos operadores han sido especificados en una presentación algebraica Maude [7], definida en un módulo paramétrico (llamado MOMENT-OP). El parámetro formal de este módulo viene definido por una teoría que permite representar cualquier modelo mediante un conjunto de tipos (*sorts* según la terminología de los lenguajes de especificación algebraica) y un conjunto de operadores que permiten definir términos para dichos *sorts* y manipularlos. Esta teoría (llamada MOMENT-REP) constituye un metamodelo simple, a partir del cual se puede definir cualquier tipo de metamodelo, como el de UML, el relacional, etc. Los metamodelos definidos como presentaciones algebraicas a partir de la teoría MOMENT-REP siguiendo unas pautas simples, pueden ser pasados como parámetros actuales al módulo MOMENT-OP. Esta acción permite aplicar los operadores definidos de forma genérica a los modelos definidos mediante el metamodelo concreto pasado como parámetro, de forma automática.

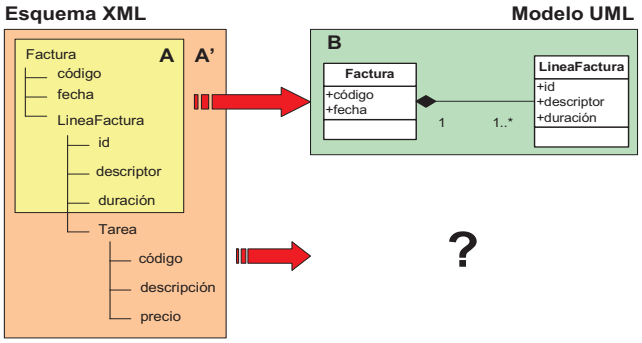


Fig. 1. Ejemplo de transformación de modelos

Por ejemplo, supóngase una aplicación desarrollada a partir del diagrama de clases B de la Fig. 1 almacena la información en documentos XML siguiendo el esquema XML A. Si se desea ampliar la información recogida por el sistema se podría modificar directamente el esquema XML A (al esquema XML modificado se le etiqueta como A'). El problema ahora consiste en reflejar los cambios sobre el modelo UML que actúa como base de la generación del código de la aplicación. Según la aproximación de gestión de modelos seguida en la plataforma MOMENT, la aplicación de tres operadores sería suficiente para obtener el diagrama de clases modificado de forma automática:

1. En un primer paso se contextualiza el operador diferencia (-) al metamodelo XML Schema Definition, XSD, (MOMENT-OP(XSD)). Con él, se determina la parte del esquema XML A' que ha sido añadida al esquema A (Fig. 2 – paso 1).
2. Acto seguido se utiliza el operador de transformación contextualizado a la unión de los metamodelos XSD y UML (MOMENT-OP(XSD+UML)) y se aplica sobre el esquema XML que aún no está considerado en el diagrama de clases. Como resultado se obtiene el diagrama de clases de la Fig. 2 (paso 2). Las transformaciones entre ambos metamodelos deben de ser especificadas previamente.

3. Finalmente se utiliza el operador de unión, contextualizado al metamodelo UML para combinar el modelo UML B y el obtenido en el segundo paso, obteniendo el modelo UML manipulado, a partir del cual se puede generar el código de la aplicación mediante técnicas de patronaje (Fig. 2 – paso 3).

Estos tres pasos podrían ser compuestos en uno dada la modularidad de los operadores:

$$(ModelGen_{XSD2UML}\ (A' - A))\ U\ B$$

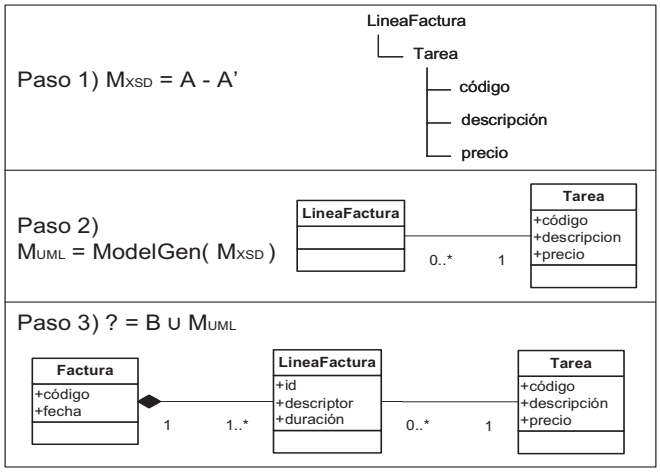


Fig. 2. Aplicación de operadores para obtener un modelo UML a partir de una extensión de un esquema XML.

Para poder aplicar estos operadores los modelos deben estar expresados como términos algebraicos de los *sorts* de la presentación algebraica de un metamodelo. Este artículo se centra en esta línea presentado una arquitectura para asociar metáforas gráficas a un metamodelo descrito algebraicamente que permita obtener la definición algebraica de un modelo a través de una representación visual cercana al dominio de trabajo del usuario.

En este apartado se ha contado de una forma intuitiva la aproximación de gestión de modelos que soporta MOMENT, con el fin de ilustrar la aplicabilidad de la herramienta a la manipulación genérica de modelos. La presentación de los fundamentos formales de esta aproximación así como la descripción de los operadores con más nivel de detalle quedan fuera del alcance de este artículo.

III. La Plataforma MOMENT (MOdel managEMENT)

La plataforma posee cuatro niveles de abstracción, siguiendo la cultura de metamodelado presentada en el estándar MOF [8], aunque no implementa directamente su lenguaje abstracto para definir metamodelos. Cada nivel está formado por un conjunto de esquemas que a su vez están constituidos por un conjunto de conceptos y propiedades. Los conceptos permiten representar entidades descriptibles en un dominio determinado, y las propiedades establecen dichas descripciones, pudiendo asociar dos conceptos mediante una propiedad o bien describir un concepto mediante una propiedad que contiene un valor básico.

Estos elementos se representan algebraicamente mediante una serie de *sorts* y las operaciones de definición de términos para dichos *sorts*, constituyendo la teoría MOMENT-REP. Esta teoría indica cómo describir un metamodelo que pueda ser pasado como parámetro al módulo MOMENT-OP que contiene las operaciones genéricas de manipulación de modelos.

A pesar que esta teoría ha sido implementada algebraicamente usando el modelo computacional lambda-cálculo mediante el lenguaje funcional F# [9], se indican los constructores de los principales *sorts* de la presentación algebraica en notación Maude por ser más conocida su sintaxis en el contexto de las especificaciones algebraicas:

Un esquema es definido mediante el constructor

op schema : LayerId SchemaId SchemaId NodeSet -> Schema

En el cual se indica el nivel en el que se está definiendo el esquema (si es el nivel M2 se trata de un metamodelo, si es el nivel M1 se trata de un modelo y si es el M0 se trata de la población de un modelo), el identificador del esquema padre del nivel superior (por ejemplo para un modelo UML del nivel M1 sería su metamodelo del nivel M2), su identificador y el conjunto de nodos que definen el esquema, que pueden ser conceptos o propiedades.

Un concepto se define mediante el término:

op concept___ : LayerId SchemaId ConceptId ConceptId -> Concept [ctor]

En el que se indica el nivel y el esquema en el que se encuentra el concepto, el identificador del concepto padre (definido en el esquema padre del esquema que lo contiene), y su identificador.

Una propiedad se define mediante el término:

op conceptProperty_____ : LayerId SchemaId PropertyId PropertyId Cardinality Cardinality NodeId ConceptId -> Property [ctor]

En el que se indican el nivel y esquema en el que se define la propiedad, el identificador de la propiedad padre, su identificador, las cardinalidades mínima y máxima que afectan a las instancias de los conceptos relacionados mediante la propiedad, el identificador del concepto que es descrito mediante la propiedad (sujeto) y el concepto que la describe (objeto). Una propiedad tiene dos variantes:

1. Propiedad de tipado: permite asociar un tipo de datos a un determinado concepto. En ella el concepto objeto es sustituido por un tipo de datos.
2. Propiedad de valor: permite indicar el valor asociado a un concepto mediante una instancia de una propiedad de tipado. En ella el concepto objeto se sustituye por un valor adecuado según el tipo de datos indicado en la propiedad padre.

Los niveles que constituyen la plataforma MOMENT se representan como términos del sort Layer que agrupan una serie de esquemas. Siguiendo el enfoque de metamodelado del estándar MOF, se dispone de los siguientes niveles (como se ilustra en la Fig. 3):

1. Nivel M3: contiene los elementos (conceptos y propiedades) necesarios para poder describir cualquier

metamodelo en el siguiente nivel. Es el nivel más abstracto de la plataforma.

2. Nivel M2: sus esquemas constituyen metamodelos, donde sus respectivos conceptos y propiedades indican como definir un modelo en el siguiente nivel.
3. Nivel M1: sus esquemas representan modelos de un determinado metamodelo del nivel M2. Los conceptos y propiedades que forman un modelo permiten definir información en el siguiente nivel.
4. Nivel M0: sus esquemas representan instanciaciones de modelos del nivel M1 que contienen datos concretos.

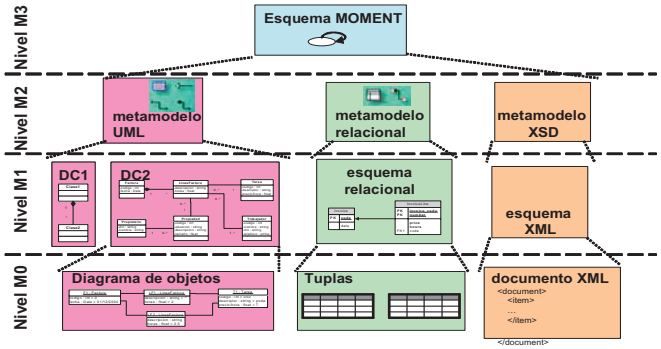


Fig. 3. Jerarquía de niveles de modelado de la plataforma MOMENT

Una vez definidos los metamodelos y los modelos mediante términos de la teoría MOMENT-REP (implementada en F#) se traducen automáticamente a términos en notación Maude con el fin de ser tratados mediante los operadores genéricos de la herramienta. Para facilitar el uso de esta teoría para representar modelos algebraicamente presentamos una arquitectura que permite realizar esta operación a través de una metáfora gráfica cercana al dominio del usuario.

IV. Elementos principales del entorno visual de herramientas CASE de modelado

El área de trabajo de las herramientas CASE de modelado visual tienen cuatro elementos en común, que son descritos a continuación:

1. Figuras: Símbolos que pueden ser prediseñados o pueden ser de creación propia, ajustándose a las necesidades del modelo que el usuario necesita representar. Dichos símbolos pueden contener información de valor añadido mediante propiedades.
2. Hoja de dibujo: Zona de trabajo donde se depositan (pegan) las figuras, constituyendo un diagrama o modelo gráfico concreto.
3. Primitivas gráficas: Distintos tipos de figuras que se pueden definir en la hoja de dibujo de la herramienta CASE. Una primitiva gráfica permite definir una figura en la hoja de dibujo. Poseen unos atributos o propiedades por defecto que definen las características de sus figuras.
4. Vistas gráficas: Plantillas que agrupan a las primitivas gráficas según el modelo o tipo de diagrama al que pertenecen, encapsulando la ontología gráfica que permite definir un modelo.

MS Visio 2003 es una herramienta visual de modelado fácilmente personalizable y extensible mediante la tecnología .NET. En el vocabulario propio de la herramienta, los elementos que hemos identificado anteriormente de forma genérica reciben los siguientes nombres: las figuras son denominadas shapes; la hoja de dibujo shapesheet; una primitiva gráfica recibe el nombre de master; y la vista que agrupa una serie de primitivas gráficas recibe el nombre de stencil.

La personalización del entorno visual de Visio se realiza mediante add-ons, es decir, conjuntos de stencils que proporcionan la información que constituye un metamodelo completo. La extensión de la herramienta se consigue mediante add-ins, que son módulos que permiten añadir funcionalidad a la herramienta. Dada la facilidad de extensión aportada por Visio 2003, ha sido la herramienta escogida para construir la interfaz gráfica de usuario de MOMENT.

V. Arquitectura de la solución

Partiendo de la plataforma formal de gestión de modelos MOMENT, se ha desarrollado un módulo, llamado *MOMENT Integrator*, que permite asociar una metáfora gráfica a un metamodelo formal mediante la interfaz de herramientas CASE con entornos visuales que proporcionen cierto grado de extensibilidad, como MS Visio 2003.

A. Arquitectura

En la Fig. 4, se muestra la arquitectura que permite integrar la plataforma MOMENT en una herramienta CASE con un entorno gráfico de forma genérica. Aunque se ha contextualizado su implementación en el entorno Visio 2003, se está pensando en desarrollar la misma arquitectura para integrar MOMENT en la herramienta de desarrollo Eclipse aprovechando su entorno gráfico Graphical Editor Framework [10]. La arquitectura utilizada es la tradicional separación en tres capas: una interfaz que permite representar gráficamente metamodelos y modelos, la funcionalidad que permite asociar la metáfora gráfica a un metamodelo especificado algebraicamente y la capa de persistencia en la que se almacena toda la información utilizada.

En la interfaz gráfica, la herramienta CASE ofrece la funcionalidad necesaria para definir los gráficos que se van a utilizar en un metamodelo, y permite acceder a la funcionalidad del módulo *MOMENT Integrator*.

En la capa de funcionalidad, el módulo *MOMENT Integrator* permite definir asociaciones entre los elementos gráficos que se han definido mediante el editor gráfico de la herramienta CASE y los términos algebraicos mediante los cuales MOMENT representa un metamodelo. Como veremos en la siguiente sección, dichas asociaciones se almacenan en la propia plataforma como instancias de clases UML a través de la librería MomentUMLSupport.

En la capa de persistencia se pueden distinguir dos unidades principales de almacenamiento: la de los elementos gráficos de la herramienta CASE y la de la plataforma MOMENT.

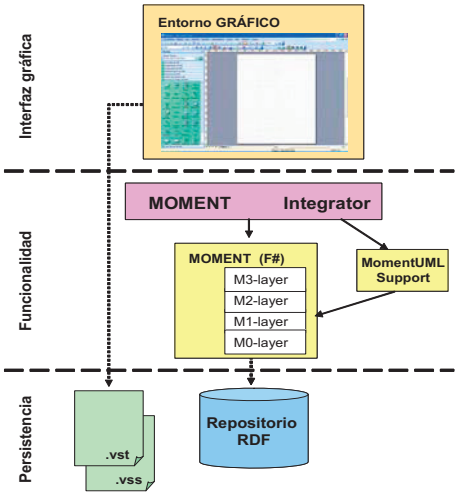


Fig. 4. Arquitectura de la integración de MOMENT en una herramienta CASE.

Las herramientas CASE suelen almacenar la información gráfica mediante una serie de ficheros. En el caso específico de la herramienta Visio, los modelos gráficos se almacenan mediante dos tipos de ficheros. Los ficheros con extensión *.vss* almacenan los documentos que se crean, o sea el área dibujada con figuras (*shapesheet*), junto con las plantillas o templates a partir de las cuales se ha creado el diagrama o modelo. Los ficheros con extensión *.vst* almacenan las plantillas con *masters* (*stencils*). Por defecto, Visio ya incorpora un gran número de plantillas con todo tipo de formas para todo tipo de modelos, pero un usuario puede crearse y guardar sus propias plantillas con una colección de formas de otras plantillas o con formas creadas por el mismo.

Por otro lado, MOMENT almacena toda la información en un repositorio RDF puesto que los conceptos y propiedades manejados en la plataforma son un reflejo de los recursos y propiedades RDF [11]. El repositorio utilizado es Redland [12] y permite almacenar la información sobre esquemas de todos los niveles de la plataforma y asociaciones entre elementos gráficos del entorno visual y términos de MOMENT, explotando las facilidades que nos ofrece el entorno gráfico de la herramienta CASE.

B. Diseño de la capa de funcionalidad

Mediante el módulo *MOMENT Integrator*, los elementos gráficos de la herramienta CASE se vinculan con los elementos de la plataforma MOMENT. Cómo nuestro objetivo consiste en la definición de la metáfora gráfica asociada a un metamodelo y en su utilización para la definición de modelos, nos centramos en el uso de los esquemas de los niveles M2 y M1 de la plataforma, que son los metamodelos y los modelos, respectivamente. De manera que, un esquema del nivel M2 se asocia con una vista gráfica (*stencil* en Visio), mientras que un esquema del nivel M1 se asocia con una hoja de dibujo (*shapesheet* en Visio). Por otro lado, para representar gráficamente los conceptos y las propiedades de los esquemas que constituyen un metamodelo en el nivel M2 de la plataforma, hacemos uso de las primitivas gráficas que componen la vista gráfica elegida (*masters* en el contexto de Visio). En el caso de la representación gráfica de elementos que constituyen un modelo en el nivel M1, se utilizan las figuras asociadas (*shapes* en Visio) a las primitivas gráficas de la vista gráfica.

La asociación de las especificaciones algebraicas de MOMENT a la representación gráfica que proporciona el entorno visual se ha modelado utilizando notación UML tal y como indica la Fig. 5. En este modelo la clase *SchemaWrapper* contiene la información necesaria para relacionar un esquema con su representación visual, mientras que la clase *NodeWrapper* contiene la información necesaria para relacionar un concepto o una propiedad con una imagen determinada. Las especializaciones de ambas clases permiten determinar si un esquema es un metamodelo (*GraphicViewWrapper*) o un modelo (*GraphicModelWrapper*), o bien si un concepto o una propiedad es un elemento de un metamodelo (*GraphicPrimitiveWrapper*) o de un modelo (*PictureWrapper*).

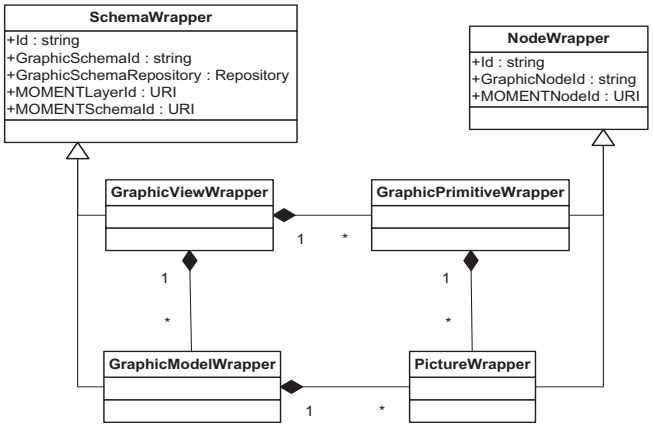


Fig. 5. Diagrama de clases UML que modela las asociaciones de elementos gráficos con términos algebraicos de la plataforma MOMENT.

Para asociar un metamodelo especificado en el nivel M2 de la plataforma MOMENT con su representación gráfica en el entorno de modelado nos centramos en las siguientes clases:

– *GraphicViewWrapper*: es la clase compuesta y encapsula información sobre una vista gráfica determinada y sobre el esquema correspondiente del nivel M2 de la plataforma. Es en esta clase, por tanto, donde se asocia el esquema del metamodelo a la plantilla gráfica. Una instancia de la clase *GraphicViewWrapper* está formado por un conjunto de instancias de la clase *GraphicPrimitiveWrapper* que definen las primitivas gráficas del metamodelo.
La información que contiene esta clase es el identificador o nombre del contenedor, el de la vista gráfica (*stencil* en el contexto de Visio) al que se le asocia el metamodelo junto con la ruta completa donde se encuentra el fichero en el que se almacena la vista gráfica, y finalmente los atributos que identifican la definición del metamodelo en la plataforma, es decir, los identificadores del nivel y del esquema. El tipo

de datos *Repository* permite abstraer el mecanismo de almacenamiento de una herramienta CASE específica en el módulo coordinador *MOMENT Integrator*.

– *GraphicPrimitiveWrapper*: es la clase componente que encapsula la información sobre una primitiva gráfica específica (*master* en el contexto de Visio) y el concepto correspondiente de un metamodelo de la plataforma MOMENT. Este concepto puede disponer de una serie de propiedades, definidas como términos en MOMENT.
  Los atributos de esta clase son un identificador o nombre para el contenedor, el identificador de la primitiva gráfica que pertenece a la vista gráfica especificada y el identificador del nodo MOMENT (concepto o propiedad) que se asocia.

Cuando se utiliza la representación visual del metamodelo para definir un modelo en el nivel M1 de la plataforma, la relación entre los elementos de la especificación algebraica del modelo y su representación visual también debe ser plasmada y almacenada. Con dicho objetivo se hace uso de las siguientes clases:

– *GraphicModelWrapper*: encapsula información sobre una hoja de dibujo y sobre el esquema que especifica el modelo contenido en la hoja de dibujo en el nivel M1 de la plataforma. Una instancia de la clase *GraphicModelWrapper* está formado por un conjunto de instancias de la clase *PictureWrapper* que definen las figuras que constituyen la representación visual del modelo. Además una instancia de esta clase puede acceder a la información que describe el modelo, tanto algebraicamente como visualmente.
  La información que contiene esta clase es el identificador o nombre del contenedor, el de la hoja de dibujo (*shapesheet* en el contexto de Visio) al que se le asocia el modelo junto con la ruta completa donde se encuentra el fichero en el que se almacena mediante el tipo de datos *Repository*, y finalmente los atributos que identifican la definición del modelo en la plataforma, es decir, los identificadores del nivel y del esquema.
– *PictureWrapper*: es la clase componente que encapsula la información sobre una primitiva gráfica específica (*shape* en el contexto de Visio) y el concepto o propiedad correspondiente de un modelo de la plataforma MOMENT. Este concepto puede disponer de una serie de propiedades, definidas como términos algebraicos en MOMENT. Una instancia de esta clase dispone de la información que describe una primitiva gráfica en la representación visual de un metamodelo.
  Los atributos de esta clase son un identificador o nombre para el contenedor, el identificador de la figura que pertenece a la hoja de dibujo indicada y el identificador del nodo MOMENT (concepto o propiedad) que se asocia.

C. MOMENT como repositorio de artefactos software UML

La asociación entre elementos gráficos de la herramienta CASE con elementos de la plataforma definida en el diagrama de clases UML de la Fig. 5, se almacena utilizando la propia plataforma MOMENT como repositorio orientado a objetos, puesto que permite definir modelos UML y sus extensiones (instancias de las clases que constituyen el modelo). Con el fin de presentar como se almacenan artefactos software UML en la plataforma se presenta como se ha especificado algebraicamente parte del metamodelo UML y cómo se define un modelo a partir de dicha especificación.

1) Especificación de parte del metamodelo UML en el nivel M2 de la plataforma MOMENT

Para llegar a tal fin, se aprovecha la parte del metamodelo UML, especificado como un esquema en el nivel M2 de la plataforma, que permite la definición de clases y de asociaciones. Se ha especificado la parte del diagrama de clases definida en la especificación del metamodelo UML (versión 1.5) [1] que aparece en la Fig. 6.

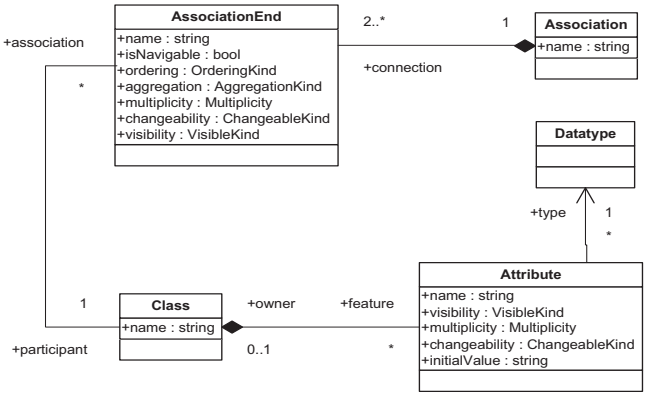


Fig. 6. Identificación de elementos en el metamodelo UML

El metamodelo se especifica mediante un término *Schema* que tiene como tipo a la ontología abstracta de MOMENT, es decir, al esquema *MOMENTSchema* del nivel M3 de la plataforma:

(schema 'M2-layer' 'MOMENTSchema' 'UML' set_nodes)

Donde *set_nodes* es la lista de términos *Concept* o *Property* que especifican el contenido del metamodelo. Las clases de la parte del metamodelo UML considerado se representan como instancias del concepto *MOMENTConcept* del esquema *MOMENTSchema* del nivel M3 mediante términos *Concept*. De manera que el término que representa el concepto clase es el siguiente *(concept 'M2-layer' 'UML' 'MOMENTConcept' 'Class')*. De igual manera se procede con el concepto *Attribute* mediante el término *(concept 'M2-layer' 'UML' 'MOMENTConcept' 'Attribute')*. La asociación existente entre ambos conceptos se especifica mediante una propiedad expresada mediante los términos *(conceptProperty 'M2-layer' 'UML' 'MOMENTProperty' 'class/Feature' (card 0) many 'Class' 'Attribute')* y *(conceptProperty 'M2-layer' 'UML' 'MOMENTProperty' 'attribute/owner' (card 1) (card 1) 'Attribute' 'Class')*. Estas propiedades indican que una instancia del concepto *Class* se puede relacionar con muchas instancias del concepto *Attribute* en un esquema del nivel M1, mientras que una instancia del concepto *Attribute* sólo puede acceder a una instancia del concepto *Class*. De esta manera se procede con las demás clases y asociaciones del metamodelo UML.

Para especificar los atributos de una clase del metamodelo se define un concepto. Por ejemplo, el atributo *changeability* de la clase *Attribute* se representa mediante una propiedad de tipado especificada como *(datatypeProperty 'M2-layer' 'UML' 'MOMENTProperty' 'attribute/Changeability' card 1 card 1 'Attribute' string)*.

2) Especificación del diagrama de clases que representa la relación entre la representación algebraica de un artefacto software y su representación visual en el nivel M1 de la plataforma MOMENT.

A partir del término algebraico que representa la parte del metamodelo UML considerada, se ha definido el modelo UML de la Fig. 5 como un esquema del nivel M1 de la plataforma.

El modelo se representa como un término *Schema* en el nivel M1 de la plataforma se representa mediante el término *(schema 'M1-layer' 'UML' 'GraphicMetaphorLinker' set_nodes)*, indicando que es instancia del esquema que representa el metamodelo UML en el nivel M2. En este término *set_nodes* es el conjunto de conceptos y propiedades que constituyen el modelo.

La clase *GraphicViewWrapper* se especifica mediante el término *(concept 'M1-layer' 'GraphicMetaphorLinker' 'Class' 'GraphicViewWrapper')*, indicando que es instancia del concepto *Class* del metamodelo UML. Para especificar su atributo *Id* se instancia el concepto *Attribute* mediante el término *(concept 'M1-layer' 'GraphicMetaphorLinker' 'Attribute' 'GraphicViewWrapper_Id')*. Este atributo se asigna a la clase *GraphicViewWrapper* mediante el término *(conceptProperty 'M1-layer' 'GraphicMetaphorLinker' 'class/Attribute' 'graphicViewWrapper/ GraphicViewWrapper_Id' (card 1) (card 1) 'GraphicViewWrapper' 'GraphicViewWrapper_Id')* y *(conceptProperty 'M1-layer' 'GraphicMetaphorLinker' 'attribute/Class' 'graphicViewWrapper_Id/ GraphicViewWrapper' (card 1) (card 1) 'GraphicViewWrapper_Id' 'GraphicViewWrapper')*, que representan instancias de las propiedades *class/Attribute* y *attribute/Class*, respectivamente. Para definir el tipo del atributo se especifica una propiedad de tipado como instancia de la propiedad *attribute/Type* del metamodelo UML mediante el término *(datatypeProperty 'M1-layer' 'GraphicMetaphorLinker' 'attribute/Type' 'GraphicViewWrapper_Id' (card 1) (card 1) 'GraphicViewWrapper_Id' string)*. Por otra parte para dar valor a la propiedad *changeability* del atributo *GraphicViewWrapper_Id* se especifica una propiedad de valor mediante el término *(valueProperty 'M1-layer' 'GraphicMetaphorLinker' 'attribute/Changeability' 'GraphicViewWrapper_Id/Changeability' (card 1) (card 1) 'GraphicViewWrapper_Id' false)*, indicando que el valor del atributo *GraphicViewWrapper_Id* no es variable.

3) Especificación de las instancias de las clases de un modelo UML en el nivel M0 de la plataforma MOMENT

Para definir instancias de las clases de un modelo UML se procede de la misma manera que se ha indicado para definir un modelo del nivel M1 a partir de un metamodelo del nivel M2.

De esta manera, MOMENT ya dispone de la información necesaria para almacenar instancias de las clases del diagrama de clases definido para relacionar la representación algebraica de un modelo con su representación visual.

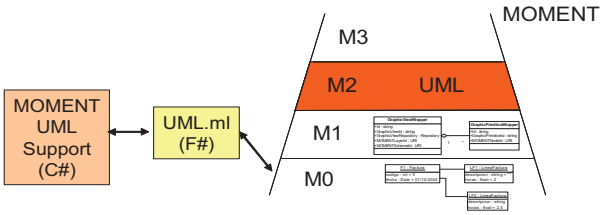


Fig. 7. Estructura de la librería MomentUMLSupport

Para facilitar el uso de MOMENT como repositorio orientado a objetos, se ha desarrollado una API que se comporta como la interfaz directa entre la plataforma de gestión de modelos y el módulo *MOMENT Integrator*. Esta librería, llamada *MomentUMLSupport*, ha sido desarrollada en C# y utiliza las funciones de un módulo F#, llamado *UML.ml*, que utiliza la funcionalidad de MOMENT para insertar instancias de las clases de un modelo del nivel M1, en una instanciación de modelo del nivel M0. La Fig. 7 muestra la estructura de la librería *MomentUMLSupport*.

El API de la librería *MomentUMLSupport* permite manipular instancias de clases de cualquier modelo UML, involucrando la instanciación, consulta o destrucción de objetos en el nivel M0 de la plataforma. De manera que cuando se instancia una clase mediante esta librería, se está almacenando la información en la plataforma.

D. Definición de la metáfora gráfica de un metamodelo

Para construir un metamodelo mediante una herramienta CASE asociamos un metamodelo MOMENT (un esquema del nivel M2) a una vista gráfica. Una vez definido el significado de esa vista, se especifica cada uno de sus primitivas gráficas con conceptos y propiedades de un esquema del nivel M2 de la plataforma, completando la metáfora gráfica asociada al metamodelo. A continuación se detallan estos dos pasos en el contexto de la implementación que se ha realizado para la herramienta MS Visio 2003:

1. En primer lugar, para definir el *stencil* se accede a la interfaz de definición que se muestra en la Fig. 8, donde se selecciona el *stencil* que se desea asociar y el esquema del nivel M2 de la plataforma que define el metamodelo.
2. Una vez asociada una vista gráfica a un metamodelo MOMENT mediante una instancia *GraphicViewWrapper*, ya sea por haber seguido los pasos anteriores o por haber cargado la plataforma MOMENT de un repositorio Redland, se definen gráficamente los conceptos y propiedades que posee el metamodelo en MOMENT.
   Para ello se accede a la interfaz de la Fig. 9, donde se selecciona un metamodelo definido gráficamente. Acto seguido, se selecciona una primitiva gráfica de la vista seleccionada y un nodo (concepto o propiedad) del metamodelo MOMENT especificado.

Después de haber definido una primitiva gráfica, se pueden consultar las propiedades ocultas que describen al nodo MOMENT, es decir, aquéllas que no están definidas gráficamente. La información de esas propiedades aparece en la lista inferior, y además se puede navegar recursivamente por las propiedades mediante el árbol situado en la parte izquierda inferior de la ventana.

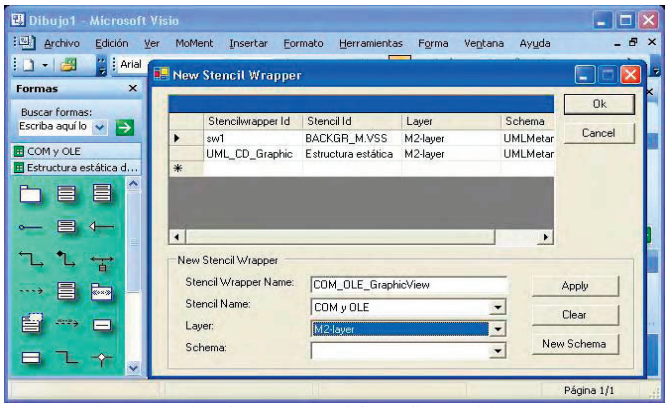


Fig. 8. Creación de un metamodelo mediante VisioMoment

Una vez se ha definido un metamodelo en la herramienta CASE, se puede definir un modelo de la forma convencional mediante la técnica *drag-and-drop* característica de este tipo de entornos de modelado, soltando primitivas gráficas de la vista gráfica sobre la hoja de dibujo, creando una nueva figura. El módulo *MOMENT Integrator* también enriquece esta funcionalidad definiendo los términos del esquema correspondiente del nivel M1 de forma automática y transparente al usuario. De manera que cuando un nodo del metamodelo tiene asignado un conjunto de propiedades ocultas, éstas se instancian tomando como valor el valor por defecto especificado para la propiedad o uno asignado de forma automática por el módulo *MOMENT Integrator*. De manera que no sólo definimos un modelo gráfico, sino que además utilizamos la información semántica asociada a un metamodelo, tanto para los conceptos como para las propiedades.

Además, cuando disponemos de un metamodelo en MOMENT que ya ha sido asociado a una metáfora gráfica, se puede cargar un modelo de ese metamodelo, directamente del nivel M1 de MOMENT. Automáticamente se asocian las figuras, que son instancias de las primitivas gráficas de la vista asociada al metamodelo, a los términos concepto y propiedad del modelo formal, dando lugar a un diagrama que representa gráficamente el modelo cargado mediante la metáfora del metamodelo.

VI. Trabajos Relacionados

La definición de metáforas gráficas para técnicas de modelado no es una idea nueva en Ingeniería del Software como se indica en [13], pero pocas son las herramientas que permiten especificaciones formales, por no decir ninguna de las herramientas CASE más conocidas. En esta sección tratamos las herramientas CASE más conocidas en dos ámbitos de modelado: el independiente de dominio y el específico a dominio.

La técnica de modelado más representativa de las independientes de dominio es, sin duda alguna, UML. UML aporta una notación gráfica para diseñar productos software siguiendo el paradigma orientado a objetos, pero no proporciona una especificación formal y sólida que garantice la producción de software de calidad y consistente.

La falta de una definición formal de los elementos del estándar UML provoca que el propio estándar vaya evolucionando sin garantía alguna sobre compatibilidad entre las diferentes versiones. De esta manera, una herramienta CASE que soporte UML, perfectamente puede quedar desfasada tras la publicación de una nueva versión del estándar. La existencia de una definición formal permitiría utilizar herramientas para demostrar que la nueva versión es compatible con la anterior o incluso facilitar la migración automática hacia la nueva versión.

No obstante, hoy en día, existen herramientas CASE que permiten generar un producto software total o parcial a partir de un modelo UML, como Rational Rose [14], MS Visio [15], ArgoUML [16]. Estas herramientas únicamente suelen generar

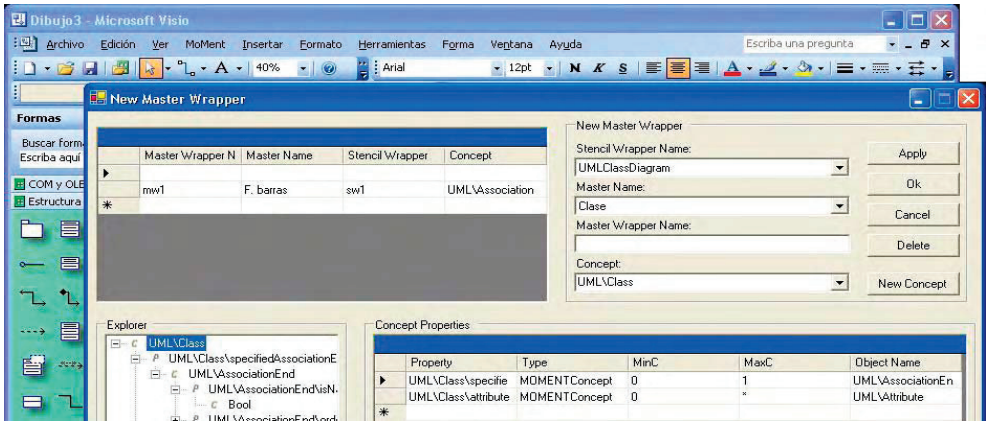


Fig. 9. Creación de los conceptos y propiedades de un metamodelo

el esqueleto de la aplicación software final y las herramientas que proporcionan productos finales suelen ser difícilmente extensibles, lo que no permite seguir la contiua evolución de UML, y se suelen centrar en la generación de un tipo específico de aplicaciones. La funcionalidad de algunas de estas herramientas ha sido embebida en entornos de desarrollo (como Rational Rose XDE Developer [17] para Visual Studio .NET y EMF (Eclipse Modeling Framework) [18] para Eclipse) explotando los modelos UML para proporcionar facilidades durante la implementación del producto software. En [19] se realiza un amplio estudio de las propuestas de formalización de UML, pero ninguna de ellas ha llegado a sensibilizar a OMG para enriquecer el estándar UML.

Por otra parte, las técnicas de modelado específicas a dominio se están consolidando como una fuerte alternativa a las independientes de dominio. Esto es debido a que permiten mejorar el proceso de producción de software especializando a los usuarios en el espacio del problema (su área de trabajo) y no en el espacio de la solución (implementación tecnológica). La computación integrada en modelos (MIC – Model Integrated Computing) [20] es una aproximación de desarrollo de sistemas y de software que permite el uso de modelos específicos a dominio para representar aspectos relevantes de un sistema. El ciclo de desarrollo MIC consiste en identificar los conceptos de un dominio, sus atributos y relaciones obteniendo un metamodelo concreto. Este metamodelo se traduce a un entorno de diseño específico del dominio (DSDE – Domain Specific Design Environment) que permite definir modelos en un dominio. Además los DSDE suelen obtener código ejecutable, realizar análisis o animar los modelos. Algunas de las herramientas que destacan en este campo son GME [21], Atom3 [22] y MetaEdit+ [23]. Todas estas herramientas constituyen un marco de metamodelado basadas en un lenguaje abstracto para definir la sintaxis, semántica y visualización de lenguajes específicos a dominio.

En cambio, muchas de ellas utilizan lenguajes abstractos propietarios o UML (como GME) para definir metamodelos en lugar de utilizar un formalismo adecuado. De esta manera, los intérpretes de modelos obtenidos a partir de un metamodelo concreto suelen tener una complejidad considerable. Además, los transformadores de modelos se basan en las capacidades de estos lenguajes para realizar dichas traducciones de modelos a metamodelos diferentes. Por otra parte, ninguna de ellas consta de un mecanismo para definir metáforas gráficas como el que proporciona el entorno gráfico de MS Visio.

Mediante el *add-in* que hemos desarrollado para dicha herramienta CASE, no somos sólo capaces de definir metamodelos específicos a dominio de una forma sencilla y consiguiendo una buena representación gráfica de los conceptos relevantes de un dominio, sino que también ofrecemos soporte formal para notaciones gráficas utilizadas masivamente para el modelado independiente de dominio, como UML. Esta aportación se consigue mediante las especificaciones algebraicas que representan artefactos software en MOMENT.

VII. Conclusiones y trabajo futuro

El modelado constituye un punto clave en el proceso de desarrollo del software, contando con diferentes aproximaciones. El modelado independiente de dominio que permite diseñar soluciones basándose en un determinado paradigma, como UML se centra en la orientación a objetos. Por otra parte, el modelado específico a dominio permite diseñar un producto software utilizando los conceptos del dominio del problema. Ambos tipos de técnicas de modelado suelen proporcionar una representación gráfica y una serie de facilidades para generar el producto software, total o parcialmente, y para transformar modelos que representan una determinada implementación tecnológica.

La teoría de grafos es el formalismo más utilizado para representar modelos y realizar transformaciones entre ellos pero su aceptación viene limitada por la complejidad de los algoritmos que proporcionan una solución a este tipo de problemas. En cambio, el enfoque algebraico de la plataforma de gestión de modelos MOMENT, permite abordar el problema de una forma genérica utilizando sistemas de reescritura de términos para realizar dichas transformaciones de una forma sencilla puesto que la implementación consiste únicamente en definir un conjunto de reglas de reescritura.

En este artículo presentamos un mecanismo de representación gráfica para los metamodelos especificados algebraicamente en MOMENT. De esta manera, permitimos definir metáforas gráficas asociándolas a metamodelos formales utilizando una herramienta CASE de gran difusión y fácilmente extensible. Este hecho permite acercar la gestión formal de modelos al usuario final a través de una interfaz gráfica fácilmente personalizable.

Para llegar a tal fin, hemos descrito la plataforma MOMENT y los principales elementos del entorno gráfico de una herramienta CASE. A continuación hemos descrito la arquitectura de un módulo que permite extender dichas herramientas, las asociaciones entre sus elementos gráficos y los elementos de la plataforma MOMENT, contextualizando la implementación en la herramienta Visio 2003. Finalmente, detallamos la interfaz que permite especificar dichas asociaciones para un metamodelo específico.

En los trabajos relacionados hemos citado herramientas que soportan el modelado independiente de dominio y específico a dominio, con mecanismos de manipulación de modelos mucho más complejos que los que ofrece MOMENT. Además con la aproximación seguida, MOMENT se beneficia del entorno gráfico de herramientas CASE, evitando el coste de desarrollo que sería necesario invertir para obtener un editor gráfico de las mismas características. Siguiendo este enfoque, se está estudiando la posibilidad de desarrollar un plug-in para la herramienta Eclipse que se beneficie de su módulo Graphical Editor Framework [10] y que utilice MAUDE [7] como sistema de reescritura de términos para manipular artefactos software en MOMENT.

VIII. REFERENCIAS

[1] Object Management Group, *Unified Modeling Language (UML) 1.5 draft*, Marzo 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01

[2] A. Boronat, J. Á. Carsí e I. Ramos, "An Algebraic Baseline for Automatic Transformations in MDA". Workshop Software Evolution Through Transformations: Model-based vs. Implementation-level Solutions (SETra'04), 2nd International Conference on Graph Transformation (ICGT2004), ENTCS, Roma (Italia). Octubre 2004.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. Quesada, *Maude: "*Specification and programming in rewriting logic". Theoretical Computer Science, 2001.

[4] G. Wideman, "Microsoft Visio 2003 Developer's Survival Pack", 2004.

[5] P. A. Bernstein, A. Y. Levy, y R. A. Pottinger, "A Vision for Management of Complex Models", SIGMOD Record (ACM Special Interest Group on Management of Data), 2000.

[6] OMG, The Model-Driven Architecture, Guide Version 1.0.1, OMG Document: omg/2003-06-01. Available from www.omg.org

[7] MAUDE. Official web site: http://maude.cs.uiuc.edu/

[8] Object Management Group, *Meta-Object Facility (MOF) version 1.4*, April 2003. http://www.omg.org/technology/documents/formal/mof.htm

[9] F# web site: http://research.microsoft.com/projects/ilx/fsharp.aspx

[10] Graphical Editing Framework. Official web site: http://www.eclipse.org/gef

[11] W3C, Resource Description Framework (RDF), http://www.w3.org/RDF/

[12] D. Beckett, "The Design and Implementation of the Redland RDF Application Framework". Tenth International World Wide Web Conference. Mayo 2-5, 2001, Hong Kong. http://www10.org/cdrom/papers/490/

[13] G. Bruno, "Model Based Software Engineering*"*, Chapman & Hall, 1995.

[14] Rational Rose Developer. Official web site: http://www-306.ibm.com/software/rational/

[15] MS Office Visio 2003. Official web site: www.microsoft.com/office/visio/

[16] ArgoUML. Official web site: http://argouml.tigris.org/

[17] Rational Rose XDE Developer. Official web site: http://www-306.ibm.com/software/awdtools/developer/rosexde/

[18] Eclipse Modeling Framework. Official web site: http://www.eclipse.org/emf/

[19] J. L. Fernández, "Una Propuesta de Formalización de la Arquitectura en Cuatro Capas de UML*"*. Tesis Doctoral, Departamento de Informática y Sistemas, Univ. de Murcia, 2001.

[20] J. Sztipanovits y G. Karsai, *Model-Integrated Computing*, Computer, Apr. 1997, pp. 90-92.

[21] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason , G. Nordstrom, J. Sprinkle y P. Volgyesi, "The Generic Modeling Environment". In Proc. Workshop on Intelligent Signal Processing. 2001.

[22] J. Lara, H. Vangheluwe, M. Alfonseca, "Using Meta-Modelling and Graph Grammars to create Modelling Environments". Workshop on Graph Transformations and Visual Modellig Techniques GT-VMT at ICGT'2002. Barcelona, Octubre 2002. ENTCS Vol.72, (3).

[23] S. Kelly, K. Lyytinen y M. Rossi, "Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment". In Proceedings of CAiSE'96, LNCS 1080, Springer-Verlag, Heraklion, Creta, Grecia, Mayo 1996, pp. 1-21.

IX. BIOGRAFÍAS



Boronat Moll, Artur es Ingeniero en Informática por la Universidad Politécnica de Valencia (España) en 2002. Ha obtenido el Diploma de Estudios Avanzados en el programa de doctorado Programación Declarativa e Ingeniería de la Programación en la UPV en 2004. Becario FPI del Departamento de Sistemas Informáticos y Computación de la UPV. Sus áreas de interés son la evolución del software, la gestión de modelos, la transformación automática de modelos y el desarrollo de software guiado por modelos.



Pedrós Salvá, Julian está realizando el proyecto final de carrera de Ingeniería en Informática en la Universidad Politécnica de Valencia (España) en el Departamento de Sistemas Informáticos y Computación de la UPV. Sus áreas de interés son la gestión de modelos y el desarrollo de software guiado por modelos.



Carsí Cubel, José Ángel es Licenciado en Informática por la Universidad Politécnica de Valencia (España) en 1996. Doctor en Informática por la UPV en 1999. Profesor Titular de Universidad del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia. Sus áreas de interés son la evolución del software, la gestión de modelos, los modelos de componentes, y los lenguajes de descripción de arquitecturas.



Ramos Salavert, Isidro es Licenciado en Ciencias Físicas por la Universidad de Valencia (España) en 1966, Licenciado en Informática por la Universidad Politécnica de Madrid (España) en 1972. Doctor en Ciencias Físicas por la Universidad Complutense de Madrid (España) en 1971. En la actualidad es profesor Catedrático de Universidad en la Universidad Politécnica de Valencia. Sus areas de interés son el desarrollo de software basado en componentes y orientado a aspectos, las arquitecturas software y la gestión de modelos.

Computación y Sistemas Vol. 9 Núm. 1, pp. 001-016
© 2005, CIC-IPN, ISSN 1405-5546, Impreso en México

Análisis Teórico-Experimental de Criterios de Comparación de Esquemas Conceptuales Orientados a Objeto

Theoretic-Experimental Analysis of Comparison Criteria for Object-Oriented Conceptual Schemas

Josep Silva, José A. Carsí e Isidro Ramos
Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Camino de Vera s/n, E-46071 Valencia, España
{jsilva, pcarsi, iramos}@dsic.upv.es

Artículo recibido en enero 10, 2002; aceptado en abril 05, 2005

Resumen

En algunos sistemas se produce la necesidad de comparar dos esquemas conceptuales orientados a objeto; el criterio de comparación utilizado, determinará en gran medida el resultado de la comparación. En este trabajo se realiza una clasificación de criterios de comparación de esquemas conceptuales orientados a objeto, a partir de su división en grupos dependientes de la información utilizada para la comparación (metainformación, información sobre su población, información exacta o estadística, etc.). Se enumeran las características de cada criterio, y se hace hincapié en sus puntos fuertes y débiles citando ejemplos. Finalmente, se presentarán estadísticas de comparación de esquemas conceptuales industriales. Las conclusiones finales, contrastarán dichas estadísticas con el análisis teórico previo. El presente trabajo forma parte del desarrollo de una herramienta para la migración automática de bases de datos, que está siendo desarrollada en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia.
Palabras Clave: Comparación de esquemas, Orientación a Objetos, Criterios de comparación, Definición de métricas, Evolución, Migración de datos, Estadísticas de comparación.

Abstract

Some systems have the need to compare object-oriented conceptual schemas; the comparison criteria used have a big influence on the result produced. In this work, a classification of object-oriented conceptual schemas comparison criteria is presented, based on the kind of the information used to compare (meta-information, population information, exact or statistical information, etc.). For each criterion, its main characteristics and strong and weak points are detailed and validated with some results from industrial conceptual schema comparisons. Final conclusions contrast the statistical results with the theoretical analysis previously done. The present work is part of a data migration tool under development by the Department of Computer Science and Information Systems of the Technical University of Valencia – Spain.
Keywords: Schema Comparison, Object-Orientation, Comparison Criteria, Metrics, Evolution, Data Migration.

1 Introducción

La comparación de esquemas conceptuales tiene un especial interés en la migración automática de bases de datos. Idealmente, la evolución del software se debe realizar a nivel de esquema conceptual, y a partir del mismo evolucionan automáticamente las aplicaciones y las bases de datos que forman el sistema de información. En ocasiones ocurre que las bases de datos del sistema original contienen información, y dicha información debe ser trasvasada al nuevo sistema, este hecho es conocido como migración de datos. El principal inconveniente en la migración de datos es que las bases de datos entre las que se quiere realizar la migración suelen ser muy distintas, y en ocasiones incompatibles.

En una migración de datos, es fundamental conocer exactamente el grado de evolución sufrido por el esquema y la base de datos asociada que va a ser migrada. Se hace indispensable conocer qué modificaciones han sufrido los elementos del esquema origen, así como qué inserciones y borrados se han producido durante el proceso de evolución. También es una tarea importante determinar qué información se debe usar para identificar el origen y el destino de la migración. Por ejemplo, si en la BD del sistema origen se dispone de la tabla α, y en el destino también se dispone de la tabla α, ¿qué nos asegura que la información de α en el origen debe ser trasvasada a la tabla α del destino? Puede haber ocurrido que la tabla α del origen se llame β en el destino, y esa información es determinante para la migración. En este contexto, adquiere especial relevancia un algoritmo capaz de automatizar el proceso de comparación de esquemas conceptuales (Silva et al., 2001).

Silva, J., Carsí, J. A., Ramos, I. Análisis teórico experimental de criterios de comparación de esquemas conceptuales orientados a objeto. Computación y Sistemas, Vol. IX no.1. 2005.

El objetivo principal de los algoritmos de comparación de esquemas conceptuales es establecer una correspondencia1 para cada elemento de los esquemas. Intuitivamente una correspondencia es una relación entre dos elementos, uno perteneciente a cada esquema, que indica que un elemento es una evolución del otro y las instancias del primero deben ser migradas al segundo. Todos estos algoritmos tienen en común que gran parte de su complejidad y de su tasa de acierto depende del criterio de comparación empleado. Se llama criterio de comparación a la información que se utiliza para comparar un conjunto de elementos dados. En un esquema conceptual la información disponible para comparar es muy rica (identificadores, nombres, relaciones con otros elementos, número de atributos en el caso de las clases, etc.) y dicha información puede determinar en gran medida el éxito de la comparación.

En este trabajo se presenta una relación de criterios de comparación de esquemas conceptuales, junto a su clasificación. Además, se presentan los resultados de un experimento de comparación de esquemas conceptuales industriales, que tiene por objetivo evaluar la calidad de los criterios de comparación presentados.

En el apartado 2, se presenta la estrecha relación existente entre la definición de criterios de comparación, y la definición de métricas. Posteriormente, en el apartado 3, se presentarán tres clasificaciones no disjuntas para criterios de comparación según distintas propiedades de los mismos. En el apartado 4, veremos una caracterización completa de criterios de comparación en función de la información utilizada para realizar la comparación. El apartado 5 presenta los resultados obtenidos en un experimento que consistía en realizar diversas comparaciones reales con algunos de los tipos de criterios presentados en los anteriores apartados. Finalmente, en el apartado 6 se extraerán las conclusiones fruto del estudio realizado.

2 Estado del Arte

Comparar siempre implica medir. Existen trabajos relativos al diseño de métricas para esquemas conceptuales. En (Dolado, 2000) puede encontrarse una recopilación de métricas destinadas a medir cada uno de los componentes de un esquema conceptual orientado a objetos. La aplicación de una métrica tiene como principal objetivo extraer una medida de alguna propiedad; pero si al aplicar una misma métrica a dos elementos distintos obtenemos medidas diferentes, nos daremos cuenta de que esos elementos son distintos. Realmente, cuando se comparan dos elementos, lo que se está haciendo es comparar dos medidas realizadas sobre alguna o algunas propiedades de esos elementos. De todo lo anterior podemos concluir que el diseño de métricas está íntimamente relacionado con el diseño de criterios de comparación.

Podemos aprovechar los estudios realizados sobre métricas para extraer, a partir de las mismas, criterios de comparación. Por ejemplo, en (Calero, 2001) se propone medir el tamaño de los servicios de una clase a partir de la complejidad ciclomática de McCabe (1976). Dicha métrica puede extrapolarse y ser redefinida como criterio de comparación de clases. A pesar de que en la mayoría de los casos se podrá obtener un criterio de comparación a partir de una métrica, no debemos olvidar que el propósito de las métricas y los criterios es esencialmente distinto; y no siempre existe un criterio adecuado que pueda derivarse a partir de una métrica. Si bien es cierto que puede ocurrir que alguna métrica no sea adecuada para derivar un criterio a partir de la misma; también lo es que todos los criterios tienen una métrica asociada; es decir, no todas las métricas pueden utilizarse para derivar un criterio de comparación de esquemas conceptuales, pero todos los criterios de comparación utilizan alguna métrica para comparar.

Se ha considerado conveniente no realizar en este punto una descripción de las métricas propuestas en la literatura; y realizarlo en un punto posterior (apartado 4) para, de este modo, poder contrastar cada criterio de comparación con su métrica asociada.

En la actualidad, el problema de la obtención de diferencias entre esquemas, es tratado por diferentes herramientas comerciales utilizando la traza de las operaciones de evolución. Este es el caso de VDIFF de Rational Rose. El principal problema que plantean estas herramientas es que los EC deben ser dependientes. Además, el hecho de que deba existir una traza de evolución entre los esquemas obliga a que los esquemas sean del mismo formato o pertenezcan a la misma aplicación. Estos problemas desaparecen con un proceso de comparación.

En (Silva et al., 2001) se presenta un algoritmo específico de comparación de esquemas conceptuales; en ese artículo se pone de manifiesto el hecho de que los algoritmos de comparación no pueden garantizar una tasa de aciertos del 100 %. Además, dicha tasa de aciertos es función directa del criterio de comparación utilizado. En el artículo se apela a la necesidad de un estudio que clasifique criterios de comparación con el fin de conocer sus propiedades y su adecuación a los distintos contextos que puede presentar una migración de datos. Si bien es cierto que existe una gran cantidad de artículos que dedican gran parte de sus esfuerzos a la definición de métricas para esquemas conceptuales; no se han encontrado en la bibliografía trabajos relativos a la clasificación de criterios de comparación. Los trabajos actuales más cercanos a la defini-

[1] El término correspondencia es la traducción al castellano de 'Mapping', su definición formal puede encontrarse en (Silva et al., 2001).

ción de criterios para comparar esquemas conceptuales, tienen por objetivo comparar los esquemas con el fin de estimar la calidad de la especificación.

En los siguientes apartados se aborda el problema de la comparación desde una perspectiva de evolución, tratando de definir criterios que identifiquen la evolución de los elementos de esquemas conceptuales orientados a objeto.

La comparación de esquemas conceptuales es una necesidad independiente del lenguaje de modelado subyacente que se utiliza. Por ello, la caracterización de criterios que se va a presentar es genérica y, por lo tanto, válida para cualquier modelo orientado a objetos que se utilice. No obstante, la mayoría de los lenguajes de modelado presentan características propias que permiten enriquecer con más información algunos criterios de comparación. A continuación, se va a presentar el lenguaje de modelado que se ha utilizado para elaborar el experimento, y al cual están referidos la mayoría de los ejemplos.

'Open and Active Specification for Information Systems', OASIS (Letelier et al., 1998), es un lenguaje de especificación formal para el modelado conceptual de sistemas de información usando el paradigma orientado a objetos. En OASIS todos los objetos son especificados a partir de su plantilla de objeto. Una plantilla de objeto está totalmente definida mediante la tripleta (I, τ, ρ), donde I es una función de identificación utilizada para identificar unívocamente al objeto, τ indica el tipo del objeto, y ρ especifica la parte pública de τ. El tipo de un objeto se define a partir de la tupla (A, X, ϕ, π) donde A es el conjunto de atributos, X el conjunto de eventos, ϕ es el conjunto de formulas de la lógica dinámica que especifican el comportamiento de los objetos, y π es un conjunto de términos del álgebra de procesos. ρ es el par (Ap, Xp) : $Ap \subseteq A \ \wedge \ Xp \subseteq X$. Toda la información de un objeto OASIS susceptible de ser utilizada para comparar está incluida en los elementos de la plantilla de objeto.

En OASIS está contemplada la herencia simple y la múltiple. El hecho que dos objetos están relacionados se modela mediante la agregación que pueden ser inclusiva o relacional, disjunta o no-disjunta, flexible o estricta, estática o dinámica, univaluada o multivaluada. Desde el punto de vista de OASIS, las asociaciones son un caso especial de agregación dinámica y multivaluada.

Desde la versión 3.0, OASIS contempla en su especificación el metanivel de un esquema conceptual formado por metaclases. Una metaclase es una clase cuyas instancias, también llamadas metaobjetos, son objetos especiales que a su vez son clases de objetos. Una petición de servicio en un metaobjeto que modifique su estado, está modificando la plantilla de la clase que define, modificando, de esta manera el esquema conceptual. El hecho de considerar las clases del sistema como objetos, permite aprovechar información adicional de las mismas durante un proceso de comparación.

3 Clasificación de Criterios de Comparación

La clasificación de criterios de comparación puede enfocarse desde distintos puntos de vista ortogonales entre sí: dependencia o independencia de los esquemas; exactos o estadísticos; basados en información del esquema o en su población.

3.1 Criterios Basados en la Dependencia o Independencia de los Esquemas

Esta clasificación divide los distintos criterios en dos grupos atendiendo a si tienen en cuenta o no la dependencia o independencia de ambos esquemas. Dos esquemas son independientes si ninguno de los dos es una evolución directa del otro, o si lo es, no conserva sus propiedades de edición (como fecha de creación, identificadores, etc.). Tal es el caso de la migración de datos en los 'Legacy Systems'. En contraposición a los esquemas independientes están los esquemas dependientes o evolutivos, donde uno de los dos es una evolución directa del anterior fruto de una secuencia de modificaciones; en algunos sistemas dicha secuencia de modificaciones se realiza a través de llamadas a servicios de evolución. En (Carsí, 1999) se propone un marco en el que hacer evolucionar el software basado en la introducción de metaclases en el modelo de objetos OASIS (Letelier et al., 1998). Dichas metaclases definen como servicios propios más de 50 servicios que permiten hacer evolucionar los esquemas conceptuales de sistemas de información.

En adelante, indicaremos que un criterio está basado en la independencia de los esquemas con una 'I', y en la dependencia con una 'D'.

3.2 Criterios Exactos y Estadísticos

Algunos criterios pueden utilizar más de una propiedad para comparar; en este tipo de criterios se puede obtener un rango de valores como resultado de aplicar la función criterio. Llamaremos criterios exactos a aquellos que aceptan una pareja de elementos como iguales sí y solo sí, el valor es igual a uno dado; llamaremos criterios estadísticos a aquellos que aceptan un rango de valores como buenos. Veámoslo con un ejemplo:

Supongamos que tenemos que comparar dos clases usando un conjunto de propiedades δ1, δ2,..., δn. Y que esas propiedades están ponderadas mediante ρ1, ρ2, ... , ρn, respectivamente (ρi indica el peso asignado a cada propiedad). El valor de la función de comparación será:

$$V = (\mu_1 * \rho_1) + (\mu_2 * \rho_2) + ... + (\mu_n * \rho_n)$$

donde μi es un valor lógico (0 ó 1) que indica si la comparación se cumple (1), o no (0); y ρi puede adoptar cualquier valor.

Un criterio exacto sólo daría por exitosa la comparación, si la función tiene exactamente un cierto valor (dicho valor suele ser el máximo posible), mientras que un criterio estadístico aceptaría un margen de error (una aproximación). En resumen:

Exactos: comparan por la igualdad de una o más propiedades.

Estadísticos: comprueban si estadísticamente puede tratarse del mismo elemento por su similitud. En adelante, haremos referencia a los criterios exactos con una 'X' y a los estadísticos con una 'E'.

3.3 Criterios Poblacionales o No Poblacionales

Cuando se comparan esquemas conceptuales en un contexto de migración de datos, los esquemas suelen estar vinculados a una población de instancias que van a ser migradas. Dicha población contiene una valiosa información que puede ser explotada durante la comparación. En adelante, haremos referencia con una 'P' a los criterios que utilizan información relativa a la población de los esquemas, y con una 'N' a los que no la utilizan.

4 Caracterización de Criterios Comparativos

En este apartado se va a enumerar una serie de criterios de comparación junto a su descripción y su clasificación. Se ha tratado de que el estudio sea bastante exhaustivo en cuanto a lo que a migración de datos se refiere. La muestra de criterios presentada contiene una representación de todos los tipos de clasificaciones.

4.1 Criterios

Inicialmente, se van a presentar los diferentes criterios que se pueden aplicar de manera individual a la hora de comparar dos esquemas conceptuales.

4.1.1 Comparación por Nombre (I, X, N)

La comparación de elementos por el nombre es la forma más intuitiva y fácil de comparar: Dos elementos (por ejemplo clases) son el mismo elemento si tienen el mismo nombre. Este es el criterio que utilizan los algoritmos de comparación de árboles de Selkow (Selkow, 1977) y Kuo-Chung Tai (Kuo-Chung, 1979) porque es idóneo para comparar árboles etiquetados. En un árbol etiquetado, la etiqueta es la única información que tienen asociada los nodos, y por lo tanto si cambia la etiqueta cambia todo el nodo; sin embargo, en un esquema conceptual el nombre no es más que un subconjunto de información, y puede ocurrir que dicho nombre cambie sin que eso signifique el cambio total del elemento. Este criterio fallará siempre que haya cambios de nombre en los elementos de los esquemas.

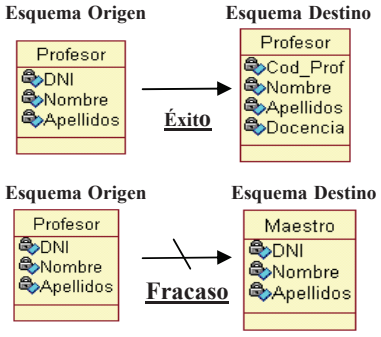


Fig. 1. Éxito y fracaso en la comparación de clases por nombre

4.1.2 Comparación por Identificador (D, X, N)

Cuando hablamos de identificador no se trata del identificador de las instancias de los elementos de los esquemas (Oid), sino que se hace referencia al identificador interno que tiene asociado cada elemento de los esquemas conceptuales. No obstante, en este artículo se hará referencia a este tipo de criterios como comparación por Oid, debido a que dicho identificador sí se corresponde con el Oid de los metaobjetos en el esquema conceptual. En una herramienta CASE, generalmente, este identificador será un valor interno de la herramienta, pero en un contexto no soportado por tecnología software puede ocurrir que este valor no exista.

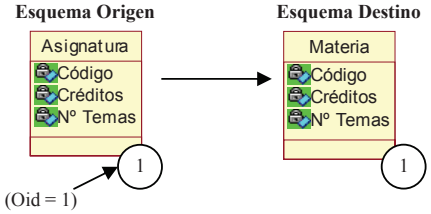


Fig. 2. Los identificadores se mantienen invariantes a la evolución

Comparar utilizando identificadores puede ser peligroso porque está (o puede estar) vinculado al editor de los esquemas. Por ejemplo, si la clase 'Asignatura' evoluciona a 'Materia' mediante un cambio de nombre, este criterio funciona perfectamente, puesto que el identificador es invariante a la evolución (ver figura 2). Sin embargo, debido al vínculo existente con el editor, cuando una misma clase que evoluciona sin ningún cambio aparente, es borrada y vuelta a crear, el editor le asignará un nuevo identificador, y hará que el criterio de comparación por identificador falle (ver figura 3).

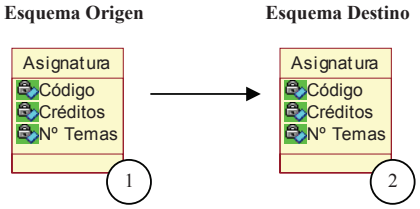


Fig. 3. Fracaso en la comparación por Identificador

4.1.3 Comparación por Metainformación (D, X, N)

Cuando los esquemas conceptuales han sido desarrollados por una herramienta CASE, a menudo ocurre que el editor almacena metainformación sobre los esquemas. Dicha metainformación puede variar en riqueza dependiendo de la herramienta, incluso puede no existir, con lo cual, no se podrían aplicar este tipo de criterios. Como ejemplos de metainformación, el editor puede almacenar la fecha de creación de cada elemento, comentarios asociados a los elementos, o incluso información dejada por el analista con el fin de guiar la comparación.

Toda esta información puede ser explotada a la hora de comparar. A partir de esta información, se podría averiguar el orden de creación de los elementos y servir de esta manera como criterio de comparación; o, como ya se ha mencionado, la metainformación podría estar destinada específicamente a guiar la comparación, asociando marcas a los elementos que el comparador podrá posteriormente utilizar.

La metainformación tiene la ventaja de que el diseñador de los esquemas puede aprovecharla para modelar de una determinada manera, guiando de esta forma la futura comparación (ver figura 4); por ejemplo, puede crear en el mismo orden los elementos del esquema, y posteriormente utilizar un criterio que compare el orden de creación. En cualquier caso, no olvidemos que en muchas ocasiones, no se dispone de dos esquemas evolucionados uno del anterior, o ni siquiera han sido hechos por el mismo editor.

Las posibilidades de la metainformación son muy grandes, pero están fuertemente ligadas a las posibilidades del editor de esquemas subyacente. Si el editor no ofrece ninguna metainformación, este criterio no podrá ser aplicado.

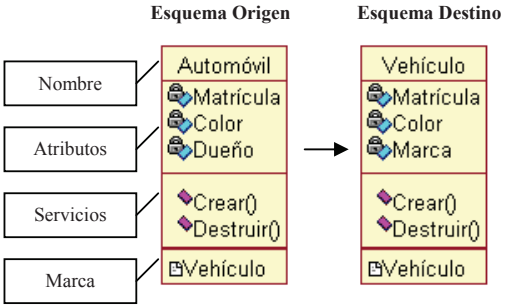


Fig. 4. La metainformación puede ser usada para dirigir la comparación

En la figura 4 se observa un ejemplo en el que el analista dejó una marca en la clase Vehículo en el momento de su modelado para que el algoritmo pueda identificar fácilmente los elementos correspondientes. En el ejemplo se observa que la marca 'Vehículo' se mantiene tras la evolución.

4.1.3.1 Comparación Utilizando Servicios de Evolución (D, _[2], N)

Realmente no se trata de un criterio de comparación, puesto que no se sigue un proceso de comparación. Algunas herramientas CASE, son capaces de guardar la traza de llamadas a servicios que hacen evolucionar un esquema.

Estos servicios son llamados servicios de evolución, y dicha traza determina totalmente la evolución (Carsí, 1999). A partir de la traza de servicios de evolución, se puede averiguar mediante procesos heurísticos de deducción qué elementos proceden de cuales. Por ello, en caso de disponer de la traza de servicios de evolución la tasa de aciertos del comparador sería del 100%. Es obvio que para disponer de dicha traza, el esquema conceptual final ha de ser una evolución directa del inicial.

4.1.3.2 Comparación por Orden Relativo de los Elementos (D, X, N)

Si representamos un esquema conceptual como un árbol etiquetado, este criterio trata de comparar el orden relativo de los elementos dentro del árbol. Para el caso de atributos consiste en comparar el orden relativo de los mismos en el subárbol que representa una clase.

Se trata de comparar cierta metainformación que muchos (prácticamente todos) editores guardan, y que consiste en el orden en que fueron creados los atributos de las clases. Se parte de la idea de que en una evolución ese orden es conservado en su mayoría.

En este ejemplo vemos que el atributo 'C.lectivos' de la clase 'Materia' está exactamente en la misma posición que el atributo 'Créditos', lo cual le otorga una probabilidad alta de ser el mismo atributo. De hecho, si fuera un atributo nuevo estaría a la derecha del atributo 'Nº Temas' (obviamente, es dependiente del algoritmo de inserción que siga el editor).

En la mayoría de los casos el orden de creación de los elementos puede inferirse a partir del fichero donde la herramienta CASE guarda el modelo, y el citado algoritmo de inserción.

[2] Por su propia naturaleza, no es aplicable la clasificación de criterios exactos y estadísticos.

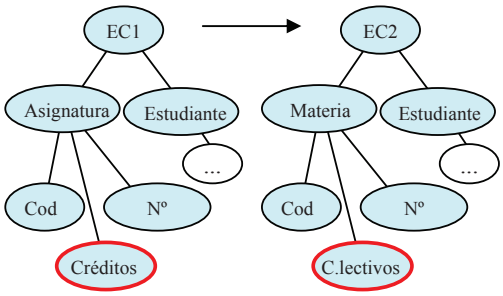


Fig. 5. El orden relativo de los elementos de conserva en la evolución

4.1.4 Comparación por Propiedades de Clase (I, E, N)

Existe mucha información asociada a las clases que puede ser explotada para comparar. Es el caso de los servicios. El número de servicios de una clase, así como su tamaño (Li y Henry (1993) proponen una métrica basada en la complejidad ciclomática de McCabe) pueden utilizarse para comparar. Asociado a este criterio, surge la idea de que todos los servicios de las clases no deberían ser tratados por igual, sino que deberían ponderarse asociando un peso mayor a los servicios compartidos. Surgen de esta manera diversos criterios asociados a los servicios de las clases. Finalmente, podemos utilizar el tamaño de las clases para comparar, considerando el tamaño de una clase como la suma del tamaño de sus atributos y sus métodos (este criterio está asociado a la métrica Sc de Li y Henry (1993), y a la versión ponderada de Thomas y Jacobson (1989)).

4.1.4.1 Comparación por Similitud de Atributos (I, E, N)

Este criterio está basado en la idea de que si dos clases evolucionan, es posible que cambien algunos atributos, pero es poco probable que cambien todos o la mayor parte de ellos. Se fundamenta en la idea de que en contextos de migración de datos, los cambios aplicados en un esquema conceptual suelen ser cambios evolucionarios y no revolucionarios (Staudt, 2000). Por su propia naturaleza, sólo puede utilizarse para comparar clases, y suele ser bastante acertado utilizarlo como segundo criterio de comparación3. Normalmente, se acepta un porcentaje de similitud como suficiente para dar la comparación por exitosa; por ejemplo, si dos clases tienen más del 75% de los atributos iguales se asume que dichas clases son la misma clase. Como puede suponerse, puede ocurrir que haya empates, lo que es un problema si no se utilizan criterios de desempate.

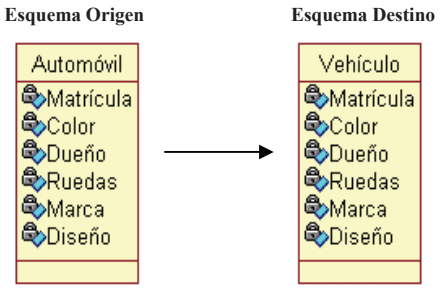


Fig. 6. Gran parte de los atributos suelen conservarse en una evolución

Existe otra variante que no mide la similitud de atributos, sino el número de atributos de la clase; este criterio puede ser útil cuando sabemos que la evolución se ha producido por renombrado de las clases; en estos casos el número de atributos permanece constante y puede ser un criterio a tener en cuenta. Debido a que también presenta el problema de los empates, este criterio puede ser muy útil combinado con otros criterios. Asociadas a este criterio está la métrica NOA (Lorenz y Kidd 1994) y la métrica WAC (Sharble y Cohen 1993).

3 Como veremos en el apartado dedicado a la combinación de criterios, lo usual es combinar varios criterios.

4.1.4.2 Comparación por Similitud de Tipos (I, X, N)

En el caso de los atributos, existe una propiedad muy propensa a ser utilizada para comparar; dicha propiedad es el tipo del atributo. Aunque es cierto que el tipo de un atributo puede cambiar en una evolución, el rango de tipos al que puede cambiar está limitado. Es posible que un atributo 'fecha' que era de tipo 'Date' evolucione a tipo 'Long' porque así lo indiquen los requerimientos; Sin embargo es poco probable que evolucione a tipo 'Char'. Aprovechando esta idea, un algoritmo de comparación puede utilizar tablas de evolución de tipos en las cuales se especifiquen las probabilidades de posibles cambios de tipo que se pueden dar en una evolución. De esta manera el algoritmo podría calcular, por ejemplo, la probabilidad de que un atributo de tipo 'Integer' evolucionara a 'long' y así descartar posibles correspondencias.

En el caso de que se comparen estrictamente los tipos de los atributos este criterio está clasificado como exacto. En el caso de que se admita una probabilidad de cambio de tipo este criterio está clasificado como estadístico.

4.1.4.3 Comparación por Función de Identificación (I, X, N)

Las funciones de identificación se forman a partir de uno o más atributos y permiten identificar a los objetos por parte de los usuarios en el espacio del problema. En algunos sistemas (Letelier et al., 1998) (Grau, 1998) las funciones de identificación son conocidas como alias. Los criterios de comparación que utilizan el alias de la clase para comparar son un caso particular de la comparación por similitud de atributos (donde los atributos a comparar son los que forman el alias de la clase). Pero es importante reseñar una diferencia importante: Este criterio es de tipo exacto, ya que todos los atributos que forman el alias deben coincidir en la comparación, mientras que la comparación por similitud de atributos es un criterio clasificado como estadístico.

4.1.5 Comparación por Propiedades de Relación (I, E, N)

Existen tres tipos de relaciones en un esquema conceptual, cada una de ellas dispone de ciertas propiedades que se pueden utilizar en el proceso de comparación. Las relaciones de agregación y asociación pueden ser comparadas utilizando las cardinalidades máximas y mínimas de la relación. La especialización dispone del tipo de especialización (temporal / permanente, por-condición / por-evento). En los tres casos, existe la posibilidad de utilizar los roles de las clases participantes, además del nombre y el identificador de la relación.

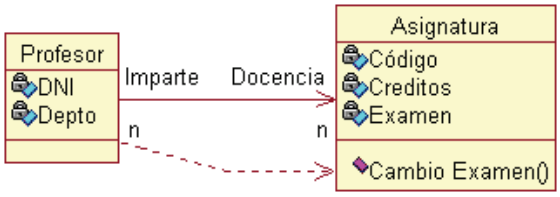


Fig. 7. Relaciones de asociación y agente entre clases

Dependiendo del modelo de objetos concreto que sirve de soporte a los esquemas conceptuales existirán variaciones, o características especiales para comparar. En OASIS existe la posibilidad de definir una agregación como inclusiva o relacional, disjunta o no-disjunta, flexible o estricta, estática o dinámica, inclusiva o multivaluada. En OmTroll (Grau, 1998) no está permitida la herencia múltiple ni la especialización dinámica, pero se puede especificar la interacción existente entre los elementos del esquema a partir del 'Communication Diagram', donde aparecen propiedades inherentes al lenguaje Troll. En la figura 7 se observan dos relaciones entre las clases 'Profesor' y 'Asignatura'. La primera relación es una relación de asociación caracterizada por sus roles y su cardinalidad; la segunda relación es inherente al lenguaje Troll, y representa una relación de agente entre ambas clases que indica que los objetos de la clase 'Profesor' pueden invocar el servicio 'Cambio Examen' de la clase 'Asignatura'.

4.1.6 Comparación por Similitud de Relaciones (I, E, N)

Del mismo modo que la comparación de clases puede estar guiada por la similitud de sus atributos, también lo puede estar por la similitud de sus relaciones con otras clases. Este criterio es totalmente resistente a los cambios sufridos en la plantilla

de una clase, siempre y cuando se mantengan sus relaciones de agregación, asociación y especialización con otras clases. Igual que ocurría en el caso de la similitud de atributos, se acepta un porcentaje de similitud como suficiente para dar la comparación por exitosa; y del mismo modo, este criterio también plantea el problema de los empates.

Debido a la independencia que mantiene este criterio respecto de la plantilla de las clases que se comparan, puede tener éxito en situaciones en las que los demás criterios suelen fallar; en la figura 8, puede observarse que a pesar de que 'Asignatura' ya no conserva ni su nombre, ni su Oid, ni ninguno de sus atributos, ni (podemos suponer) ninguna otra propiedad; si que conserva su relación con la clase 'Matemáticas', y esto hace que la comparación tenga éxito por este criterio. Este criterio también admite otra variante que no mide la similitud de relaciones, sino el número de relaciones que mantiene la clase. Asociada a este último criterio está la métrica CBO (Chidamber y Kemerer 1994).

Es evidente que para que puedan establecerse correspondencias entre clases utilizando este criterio, previamente habrán de establecerse todas las correspondencias entre las relaciones del esquema. Este es un claro ejemplo en el que se ve la interdependencia existente entre el algoritmo de comparación y el criterio utilizado para comparar.

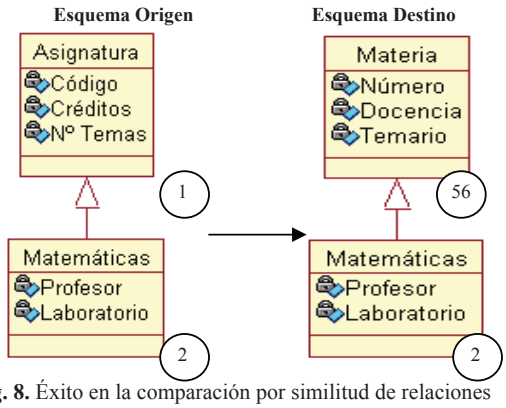


Fig. 8. Éxito en la comparación por similitud de relaciones

4.1.7 Comparación por Similitud de Clases (I, E, N)

La filosofía de este criterio es similar a la del anterior; no se fundamenta en las propiedades de los elementos que compara, sino en las de los que están relacionados con él. Se utiliza para comparar relaciones comparando las clases implicadas en la relación. En el caso de las relaciones de agregación y asociación se mirarán las propiedades de las clases componente y compuesta, y en el caso de las relaciones de especialización y generalización las de las clases padre e hija. En ambos casos se podrán aplicar las ideas comentadas en los criterios anteriores.

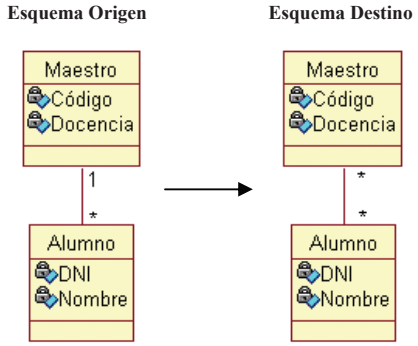


Fig. 9. Éxito en la comparación por similitud de clases

Este tipo de criterios es muy propenso a ser utilizado con la técnica de los anclajes que será explicada en la última sección de este apartado. Si el algoritmo de comparación ya ha identificado como correspondientes a dos pares de clases (existen dos anclajes), es muy probable que una relación entre ellas sea identificada como correspondiente.

4.1.8 Comparación por Población (I, E, P)

En algunas ocasiones, se da la circunstancia de que la migración entre las bases de datos asociadas a los esquemas conceptuales se realiza paulatinamente. En estas situaciones, puede ocurrir que ambos esquemas estén poblados4 cuando se produzca la comparación. En los casos en los que la población ya migrada no es borrada de las bases de datos iniciales, existe información duplicada en las bases de datos. Los criterios de comparación por población explotan dicha información para comparar.
Este criterio puede ser muy interesante cuando la población migrada no ha sufrido cambios durante la migración; en estos casos, por mucho que pueda cambiar el esquema, los datos asociados permanecen invariantes; este hecho supone una garantía en la comparación. Los algoritmos de comparación, solo pueden comparar aquella información almacenada que sea del mismo tipo. En la figura 10 se observa que la clase vehículo ha cambiado de nombre en la evolución; sin embargo, sus instancias han permanecido inmutables y permiten a los algoritmos identificar correctamente la evolución.

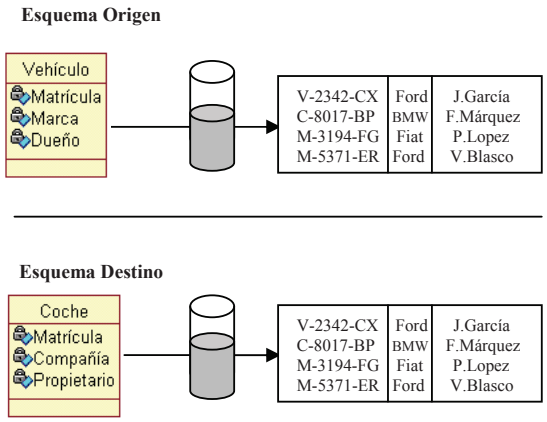


Fig. 10. La información almacenada en la base de datos puede ser utilizada para comparar

4.2 Combinación de Criterios

En los anteriores apartados se ha puesto de manifiesto que cada tipo de criterio presenta una mejor adaptación en determinados contextos. En la práctica, lo deseable es combinar varios criterios con el objetivo de mejorar la comparación cubriendo de esta manera las deficiencias de cada criterio con el otro al cual se combina.

La combinación de dos criterios puede realizarse de dos formas:

- *Combinación secuencial:* Se aplica el primer criterio (dominante5) a cada elemento, y el segundo criterio a cada elemento que no haya sido emparejado por el primer criterio.
- *Combinación paralela:* Se aplican los dos criterios a la vez a cada elemento, y se da una de las tres situaciones siguientes:
  - Los dos aciertan: Se acepta la pareja.
  - Solo uno acierta: Se acepta o rechaza la pareja.
  - Los dos fallan: Se rechaza la pareja.

Si en la situación en la que sólo un criterio acierta, se rechaza la pareja, la combinación será llamada estricta; si la pareja es aceptada, la combinación será llamada no-estricta.

4 Que un esquema esté poblado quiere decir que existen instancias almacenadas en la base de datos asociada al esquema.
5 El otro criterio será llamado criterio débil.

Sea λ el conjunto de correspondencias encontradas por un criterio δ, sea α y β los subconjuntos de λ tales que α contiene las correspondencias correctas de λ y β las incorrectas (λ = α + β). Cualquier criterio combinación secuencial δ' con δ como criterio dominante cumple lo siguiente:

a) $|\lambda'| \geq |\lambda|$
b) $|\alpha'| \geq |\alpha|$
c) $|\beta'| \geq |\beta|$
d) $\forall a \in \lambda \rightarrow a \in \lambda'$
e) $\forall a \in \alpha \rightarrow a \in \alpha'$
f) $\forall a \in \beta \rightarrow a \in \beta'$

Cualquier criterio combinación paralela estricta δ'' con δ como criterio participante cumple las siguientes propiedades:

a) $|\lambda''| \leq |\lambda|$
b) $|\alpha''| \leq |\alpha|$
c) $|\beta''| \leq |\beta|$
d) $\forall a \in \lambda'' \rightarrow a \in \lambda$
e) $\forall a \in \alpha'' \rightarrow a \in \alpha$
f) $\forall a \in \beta'' \rightarrow a \in \beta$

Para la combinación paralela no-estricta:

a) $|\lambda''| \geq |\lambda|$
b) $|\alpha''| \leq |\alpha|$ No tiene porqué cumplirse.
c) $|\beta''| \leq |\beta|$ No tiene porqué cumplirse.
d) $\forall a \in \lambda'' \rightarrow a \in \lambda$ No tiene porqué cumplirse.
e) $\forall a \in \alpha'' \rightarrow a \in \alpha$ No tiene porqué cumplirse.
f) $\forall a \in \beta'' \rightarrow a \in \beta$ No tiene porqué cumplirse.

Aunque a priori podría parecer que la combinación secuencial es similar a la combinación paralela en la que se acepta un acierto de cualquier criterio, esto no es así. En la combinación secuencial se prima mucho más el criterio dominante, de tal forma que todos los elementos que hayan sido emparejados por el criterio débil no podrán ser emparejados por el dominante.

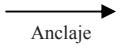
En cambio, en la combinación paralela puede ocurrir que un elemento que haya sido emparejado por el criterio débil sí pueda ser emparejado por el dominante. Es importante saber combinar los criterios, y tenerlos en cuenta a la hora de modelar, puesto que pueden mejorar sustancialmente la comparación.

4.3 Utilización de Anclajes

La comparación de esquemas conceptuales es un proceso iterativo en el que se va obteniendo información a medida que se avanza en el proceso. En este contexto, el término anclaje, hace referencia a una correspondencia previamente obtenida durante el proceso de comparación. De manera formal, un anclaje en el instante ti es una correspondencia obtenida por el algoritmo de comparación en un instante tj donde tj < ti. Los anclajes representan información que puede ser tenida en cuenta por el criterio (y el algoritmo) de comparación.

Los anclajes están sujetos a la implementación particular del algoritmo de comparación; y su validez es dependiente del esfuerzo computacional y de la cantidad de información que el algoritmo emplea para calcularlos. Por ejemplo, un algoritmo de comparación podría realizar una primera comparación entre los elementos de los esquemas con el fin de establecer un cierto número de anclajes. Utilizando dichos anclajes realizar otra comparación (con algún criterio basado en anclajes) de la que se obtendrían a su vez nuevos anclajes, y así sucesivamente hasta establecer todas las correspondencias.

Esquema Origen **Esquema Destino**



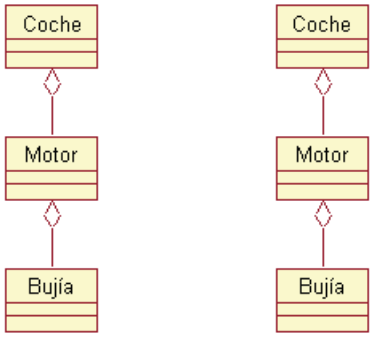


Fig. 11. Los anclajes establecen puntos de fijos para futuras comparaciones

Como puede apreciarse, el uso de anclajes supone un proceso incremental, en el que cada iteración obtiene información que podrá ser utilizada en la siguiente. De esta manera, los criterios utilizados en las primeras iteraciones deben ser criterios muy exigentes para garantizar que los anclajes sobre los que se soportarán el resto de comparaciones sean correctos.

Es importante hacer notar que un criterio de comparación puede tener una complejidad elevada si utiliza anclajes. Aunque este tipo de criterios presenta mayores costes y su complejidad ciclomática es de un orden superior (debido a su interdependencia con el propio algoritmo de comparación), este tipo de criterios será en la mayoría de los casos el criterio de comparación apropiado, puesto que por termino medio, presenta una mejoría sustancial en sus tasas de evaluación (Silva et al., 2001).

5 Análisis Estadístico de Comparación de Criterios

Una vez concluido el análisis teórico, disponemos de toda una estructura conceptual que es necesario validar de manera empírica. La experimentación que se ha realizado en este sentido ha seguido la línea que se conoce como diseño intra-sujetos. Este tipo de experimentos están caracterizados por el hecho de que una vez se ha seleccionado la muestra que va a participar en el experimento, a todos los sujetos se les aplican todas las condiciones experimentales.

A pesar de que sí se han tenido en cuenta posibles efectos distorsionantes como pudiera ser el aprendizaje, la fatiga, o la práctica; no ha existido una aleatorización rigurosa a la hora de seleccionar la muestra con la que se pretendía realizar la experimentación. Este hecho se debe a que los esquemas conceptuales no han sido seleccionados aleatoriamente, sino que han sido escogidos por sus distintas propiedades y características. Se ha seleccionado una muestra de quince esquemas conceptuales orientados a objetos, y todos ellos tienen entre sí características distintas (algunos han sufrido muchos renombrados, modificaciones masivas, simplificaciones... ; se han utilizado esquemas dependientes e independientes; se ha forzado el caso de reconstrucción de clases para la no coincidencia de Oids; etc.). Se dispone así de estadísticas de comparación, fruto de la experimentación, que ponen de manifiesto cómo se comportan los distintos criterios ante una gran diversidad de circunstancias. Este fue el objetivo principal de la experimentación y ha sido ampliamente satisfecho como se puede ver en la tabla 1.

La cual está dividida en dos bloques independientes; el primer bloque muestra las tasas de acierto obtenidas por siete criterios distintos habiendo sido utilizados frente a una muestra de quince esquemas. La fila 'Total' muestra la tasa media de acierto para cada criterio. El segundo bloque representa la muestra utilizada compuesta por quince esquemas conceptuales orientados a objeto. En esta tabla, la columna 'Evolución' indica la dependencia o independencia entre el esquema origen y el destino; el resto de campos son autoexplicativos.

La principal conclusión que se extrae de las estadísticas es que los criterios que se especializan en comparar por información identificativa obtienen mejores tasas de éxito; es el caso, por ejemplo, del alias. En todas las pruebas, la comparación por alias ha superado a la comparación por similitud de atributos. Teniendo en cuenta que el alias es un subconjunto de los atributos, queda de manifiesto que existen atributos que deberían pesar más que los demás en una ponderación. Del estudio se deduce que deben otorgarse los pesos más elevados a aquellos atributos que participen en alguna función de identificación.

Tabla 1. Estadísticas de comparación obtenidas en el experimento

Esquema	Nombre	Oid	Relaciones	Atributos	Alias	Nombre+Oid	Oid+Nombre
1	66,67%	100%	37,50%	66,67%	66,67%	66,67%	100%
2	100%	100%	82,45%	100%	100%	100%	100%
3	91,67%	0%	100%	83,33%	100%	91,67%	91,67%
4	92,86%	0%	85,71%	78,57%	85,71%	92,86%	92,86%
5	88,89%	83,33%	77,78%	83,33%	83,33%	94,44%	100%
6	100%	100%	75%	100%	100%	100%	100%
7	100%	100%	100%	100%	100%	100%	100%
8	77,78%	22,22%	66,67%	77,78%	88,89%	77,78%	77,78%
9	88,23%	94,12%	82,35%	76,47%	88,23%	88,23%	100%
10	100%	100%	100%	100%	100%	100%	100%
11	63,15%	0%	52,63%	47,37%	68,42%	63,15%	63,15%
12	83,33%	100%	83,33%	91,67%	91,67%	100%	100%
13	100%	88,89%	77,78%	88,89%	100%	100%	100%
14	100%	100%	75%	100%	100%	100%	100%
15	81,81%	81,81%	90,90%	72,72%	72,72%	90,90%	81,81%
Total	88,96%	71,36%	79,14%	83,34%	89,71%	91,05%	93,82%

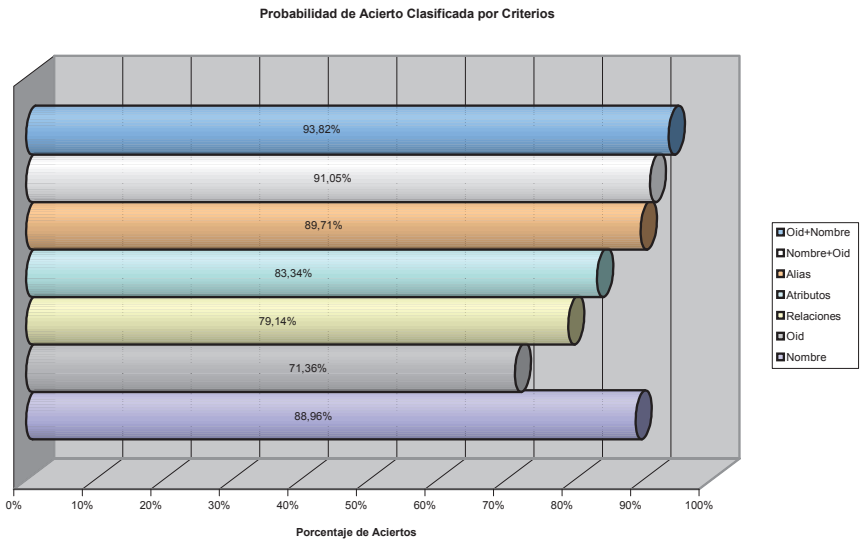
Esquema	Nombre	Evolución	Inserciones	Borrados	Modificaciones	Nº Clases
1	Gestión de facturas	Si	2	0	2	8
2	EvEs20a	Si	3	0	0	11
3	Discografía	No	4	1	5	12
4	Zoológico	No	5	2	3	14
5	Club Tenis	Si	7	3	6	18
6	Gestoría Pardo	Si	4	0	0	11
7	Comparador	Si	7	0	12	18
8	Hipermercado	No	4	2	4	9
9	Viajes Márquez S.A.	Si	6	8	12	17
10	Fotógrafos	Si	1	5	3	10
11	Venta Software	No	8	6	10	19
12	Consultoría	Si	5	3	6	12
13	Seguridad Cajeros	Si	3	2	4	9
14	Videoclub	Si	2	2	3	8
15	Competición Judo	Si	5	4	6	11

En los casos en los que los esquemas sean dependientes, los criterios que utilizan el Oid son muy aconsejables. En los casos en que sean independientes no tendrá sentido aplicar ni combinar ningún criterio clasificado como dependiente. En las estadísticas queda confirmada la idea de que la combinación de criterios mejora en la mayoría de los casos el éxito en la comparación (ver gráfica 1). Por otra parte queda de manifiesto que la combinación de criterios no cumple la propiedad conmutativa puesto que la combinación de Oid y Nombre no obtiene las mismas tasas de acierto que la combinación de Nombre y Oid. En esta combinación particular es preferible utilizar como dominante el criterio Oid frente al de Nombre.

En la tabla 2 se muestran las desviaciones típicas del conjunto de pruebas de cada criterio. Se observa que a pesar de que los esquemas conceptuales presentaban condiciones de comparación muy diversas, la desviación típica es pequeña; lo cual indica que en la mayoría de los casos los criterios tienen una tasa de aciertos similar y un comportamiento estable.

Tabla 2. Desviación típica obtenida en las pruebas de cada criterio

Nombre	OID	Relaciones	Atributos	Alias	Nombre+Oid	Oid+Nombre
0,123	0,418	0,172	0,152	0,122	0,123	0,111



Gráfica 1. Porcentaje medio de acierto obtenido en 7 criterios de comparación

Los valores de la desviación típica de la tabla 2 dan una idea de la variabilidad de comportamiento de cada criterio ante diversas situaciones experimentales. En el caso de la comparación por Oid la desviación típica es elevada; esto se debe a que se ha utilizado este criterio de tipo independiente para comparar esquemas dependientes lo cual ha hecho que se obtengan tasas de acierto muy bajas y aumente la desviación típica. Muy lejos de este valor, los criterios de comparación por Oid son extraordinariamente estables, y presentan varianzas muy pequeñas. En el caso de nuestro experimento, el valor de la desviación típica para el criterio de Oid calculado solamente a partir de las comparaciones de esquemas dependientes es de 0,072.

La experimentación que ha sido presentada, forma parte del desarrollo de una herramienta para la migración automática de datos que está siendo desarrollada en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia. Dicha herramienta es capaz de generar de manera totalmente automatizada un plan de migración de datos entre bases de datos de sistemas orientados a objetos.

La primera fase de la herramienta es la comparación automática de los esquemas a partir de una combinación de criterios de comparación, llegándose incluso a realizar combinaciones secuenciales de hasta cuatro criterios. Dicha comparación identifica los orígenes y los destinos de los datos en la migración, y a partir del mismo es generado un plan de migración por defecto. Cada comparación automática puede ser supervisada por un analista que determinará cuando ha existido un error en la comparación y podrá solucionarlo realizando nuevas comparaciones solamente sobre aquellos elementos que él determine. La experiencia obtenida hasta el momento concluye que los errores son muy escasos, obteniéndose tasas de acierto que oscilan sobre el 98%.

6 Conclusiones

El éxito en un proceso de comparación de esquemas conceptuales está totalmente ligado al criterio de comparación empleado. Podemos clasificar los criterios en función de la información que utilizan para llevar a cabo la comparación; existen tres clasificaciones primarias no disjuntas entre sí:

* Criterios que utilizan la dependencia o independencia de los esquemas.
* Criterios que requieren exactitud o aproximación en la comparación.
* Criterios que utilizan o no, la población de los esquemas a comparar.

No todos los criterios obtienen las mismas tasas de acierto, ni existe un criterio que sea siempre mejor que los demás. Cada criterio de comparación tiene unas características propias que le hacen ser más apropiado en unas situaciones, y me-

nos apropiado en otras. Deberá ser el analista quien decida qué criterio es apropiado, en función del tipo de evolución que haya entre los esquemas.

En general, lo más acertado es utilizar una combinación de criterios que se complementen entre sí. Dicha combinación puede realizarse de una manera paralela o secuencial. También es aconsejable el uso de anclajes, puesto que suponen una mejora en las tasas de acierto que justifica su incremento en el coste computacional.

Se ha establecido una tipología de criterios de comparación clasificada en función de la información utilizada para comparar. Para cada tipo de criterio, se han determinado sus características principales detalladas con ejemplos.

Finalmente, las estadísticas extraídas de comparaciones de esquemas conceptuales industriales han permitido extraer conclusiones fruto de pruebas reales. Se ha podido estudiar qué criterios mantienen un comportamiento medio más adecuado ante diversos tipos de esquemas conceptuales. Además, a partir de la varianza se ha podido estudiar cómo y cuanto varía dicho comportamiento. El análisis experimental ha contrastado el estudio teórico previo, permitiendo cuantificar las conclusiones establecidas.

Referencias

1. **Aho A.V., Hirschberg D.S.** y **Ullman J.D.**, "Bounds on the complexity of the longest common subsecuence problem", Journal of the ACM, 23(1):1-12, January 1976.
2. **Anaya R.,** "Desarrollo y gestión de componentes reutilizables en el marco de Oasis", Tesis Doctoral, Universidad Politécnica de Valencia, Noviembre 1999.
3. **Assenova P.** y **Johannesson P.**, "Improving Quality in Conceptual Modelling by the Use of Schema Transformations", in: B. Thalheim (Ed.): Conceptual Modeling: proceedings ER '96, Springer, Berlin et al., S. 277-291, 1996.
4. **Calero C.,** "Definición de un conjunto de métricas para la mantenibilidad de bases de datos relacionales, activas y objeto-relacionales", Tesis Doctoral, Universidad de Castilla-La Mancha, 2001.
5. **Carsí J.A.,** "OASIS como marco conceptual para la evolución del software", Tesis Doctoral, Universidad Politécnica de Valencia, Octubre 1999.
6. **Dolado J.J.** y **Fernández L.,** "Medición para la gestión en la ingeniería del software", Libro Editorial RA-MA, ISBN 84-7897-403-2, 2000.
7. **Grau A.,** "Computer-Aided Validation of formal conceptual models", Tesis Doctoral, Technical University of Braunschweig, 1998.
8. **Herden O.**, "Measuring Quality of Database Schemas by Reviewing - Concept, Criteria and Tool", Oldenburg Research and Development Institute for Computer Science Tools and Systems, Escherweg 2, 26121 Oldenburg, Germany
9. **Kuo-Chung Tai,** "The tree-to-tree correction problem", Department of Computer Science, North Carolina State University, Raleigh, Journal of the ACM, vol. 26 no 3, pp. 422-433, 1979.
10. **Letelier P., Ramos I., Sánchez P.** y **Pastor O.,** "Oasis 3.0: Un enfoque formal para el modelado conceptual orientado a objeto", Editorial SPUPV- 98.4011, Departamento Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 1998.
11. **Moreno M., García F. y Polo M.,** "Medición de la calidad del software en el ámbito de la especificación de requisitos", Universidad de Salamanca, Departamento de Informática y Automática, Taller JISBD, 2000.
12. **Pérez J., Carsí J.A., Ramos I., Anaya V.** y **Silva J.,** "Generación Automática de un plan de migración entre poblaciones de esquemas conceptuales orientados a objetos", Jornadas de Trabajo de DOLMEN, Sevilla 2001.
13. **Ramos I.** y **Pastor O.,** "OO-Method: Una metodología OO para la producción automática de software.", Technical Reports II/8/95, Departamento Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Valencia, España. Junio 1995
14. **Sankoff D.**, "Matching sequences under deletion/insertion constraints", Proc. Nat. Acad. Sci., USA 69, pp. 4—6, 1972.
15. **Selkow S.,** "The tree-to-tree editing problem", Department of Computer Science, University of Tennessee, Knoxville, *Information Processing Letters*, 6(6):184-186, December 1977.
16. **Silva J.,** "Comparación de esquemas conceptuales orientados a objeto y generación automática de planes de migración", Proyecto Final de Carrera, Facultad de Informática, Universidad Politécnica de Valencia, 2001.
17. **Staudt B.,** "A model for compound type changes encountered in schema evolution", University of Massachusetts, Amherst, ACM Transactions on Database Systems, Vol. 25, No. 1, March 2000, Pages 83–127.
18. **Varas M.** y **Pradenas J.,** "Hacia la definición de métricas de calidad para esquemas conceptuales de bases de datos", Revista Electrónica del DIICC, edición nº 6, año 3, 2000.
19. **Wagner R.A.** y **Fisher M.J.**, "The string to string correction problem", *Journal of the ACM*, 21(1):168-173, January 1974.
20. **Wong C.K.** y **Chandra A.K.**, "Bounds for the string editing problem", Journal of the ACM, 23(1):13-16, January 1976.



Josep Silva. *Se graduó como Ingeniero Superior en Informática en la Universidad Politécnica de Valencia, España. Terminó sus estudios de Especialista en Programación Declarativa e Ingeniería de la Programación en la misma universidad obteniendo finalmente el Diploma de Estudios Avanzados. Actualmente trabaja como investigador para el grupo de Investigación ELP de la UPV en España. Sus campos de interés incluyen métodologías formales de análisis y diseño, incluyendo transformaciones de programas como la fragmentación estática y dinámica, depuración y optimización de código en lenguajes funcionales y lógico-funcionales, y la construcción de plataformas de desarrollo para lenguajes multiparadigma.*



José A. Carsí Cubel. *Es Licenciado en Informática en 1993 y Doctor en Informática en 1999 por la Universidad Politécnica de Valencia. Actualmente es profesor Titular de Universidad en la Universidad Politécnica de Valencia siendo miembro del grupo de investigación "Programación Lógica e Ingeniería del Software". Áreas de interés: evolución del software, gestión de modelos, lenguajes de descripción de arquitecturas y modelado orientado a objetos.*



Isidro Ramos Salavert. *Es Licenciado en Ciencia Físicas por la Universidad de Valencia (España) desde 1966, Licenciado en Informática por la Universidad Politécnica de Madrid (España) desde 1972, Doctor en Ciencias Físicas por la Universidad Complutense de Madrid (España) desde 1971. Áreas de interés: modelado conceptual de sistemas de información, lenguajes de descripción de arquitecturas y gestión de modelos. Actualmente es Catedrático de Universidad y director del grupo de Investigación "Programación Lógica e Ingeniería del Software" de la Universidad Politécnica de Valencia.*

Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine

Artur Boronat, José Á. Carsí, Isidro Ramos
Department of Information Systems and Computation
Polytechnic University of Valencia
Camí de Vera s/n
46022 Valencia-Spain
{aboronat, pcarsi, iramos}@dsic.upv.es

Abstract

Legacy systems are information systems that have been developed by means of methods, tools and database management systems that have become obsolete, but they are still being used due to their reliability. As time goes on, the maintenance of this software becomes more complex, expensive and painful. The OMG's MDA initiative provides some guides to develop software that raise models and transformations between them as first class citizens. This point of view provides a precise way to develop a new application from the legacy one. The MOMENT Framework supports automatic formal model transformations in MDA. This model transformation approach is based on the algebraic specification of models and benefits from mature term rewriting system technology to perform model transformation using rewriting logic. In this paper, we present how we apply this formal transformation mechanism to recover a legacy relational database, obtaining a UML-based application. This approach keeps the legacy knowledge by pumping the legacy data to the new database generated from the UML model. Our approach enhances the integration between formal environments and industrial technologies such as .NET technology, and exploits the best features of both.

1. Introduction

Legacy systems can be defined informally as "software we do not know what to do with, but it is still performing a useful job" [1]. They are information systems that have been developed by means of methods, tools and database management systems that have become obsolete, but they are still being used due to their reliability. They are characterized by the following features:

- Software architecture based on obsolete technology that has probably been patched in order to adapt to new changes in requirements. This fact complicates the maintenance of the application.
- Poor, complex documentation that prevents effective maintenance or software updating, making it necessary to check the source code to understand the functionality of the system.
- Cumulative experience working with the system that has filled its database with information that is significant for the company.

As in all complex systems with a medium life cycle, the requirements for this kind of applications go on changing at the same rate as technology does. There are two main approaches to performing changes in these systems. On the one hand, there is the patching of the legacy system that has obsolete technology code. The disadvantages to this approach are that the technology does not consider new features to improve either code reuse, quality or documentation generation, and that the staff that will develop the new part of the system needs to be trained. On the other hand, the whole system can be developed with a new technology taking advantage of all its features. Both approaches imply a high cost, but we prefer the second option because the first one only temporarily delays the translation into a new technology, making maintenance harder and harder each time the system is changed.

The OMG's Model Driven Architecture (MDA) initiative [2] is set in this context and provides several proposals to define models, through the standard Meta-Object Facility (MOF) [3]. It also offers proposals to perform model transformations by means of the Query/View/Transformations (QVT) language, which is still in its early stages [4]. While a lot of attention has been given to the transformation of platform-

Boronat, A., Carsí, J. A., Ramos, I. Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. Proc. of the 9th European Conference on Software Maintenance and Reengineering. 2005.

independent models into platform-specific models, the scope of MDA goes beyond this in an attempt to model all the features of a software product throughout its life cycle. Nonetheless, a precise technique to provide formal support for the entire process of model transformation has not yet been developed.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. MOMENT is based on an algebra that is made up of *sorts* and operators that permit the representation of any model as a term that can be automatically manipulated by means of operators. The MOMENT Framework benefits from the best features of current visual CASE tools and the main advantages of formal environments such as term rewriting systems, combining the best of both industry and research.

This paper presents the application of the MOMENT framework to a reengineering problem where a UML model is obtained from a legacy relational database by means of the MOMENT model transformation, which relies on a term rewriting system.

The paper is structured as follows: Section 2 indicates an example to illustrate the transformation process through the paper; Section 3 provides an overview of the MOMENT Framework; Section 4 presents the model transformation mechanism of the MOMENT Framework, focusing on the use of a Term Rewriting System (TRS) as a formal environment to perform automatic model transformations; Section 5 presents other approaches to provide automatic support for model transformations. Finally, Section 6 summarizes the work and indicates the future directions of our research tasks.

2. Example of a Common Legacy System Recovering Process

Consider a car maintenance company that has worked for a long time for a large car dealership. The maintenance company works with an old C application where the information is stored in a simple relational database that does not even consider integrity constraints. The car dealership has recently acquired the car maintenance company and its president has decided to migrate the old application to a new OO technology in order to improve maintenance and efficiency. Therefore, the target application will be developed by means of an OO programming language although the database layer remains a relational database. This time, new integrity constraints are provided in the new relational database in order to improve maintainability.

Consider the part of the legacy system that stores information about invoices in the example. Each invoice contains data about the task performed in a specific period of time and at a specific price.

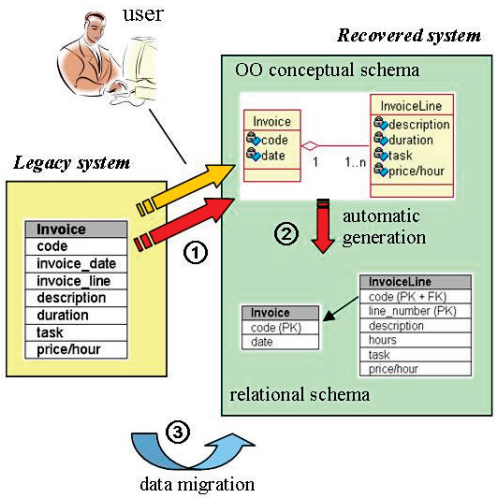


Figure 1. Legacy Database Recovering Example

To recover the legacy system, a designer has to build a semantically equivalent OO conceptual schema that captures the semantics that is disorganized in the legacy system. This task is usually done manually and involves high development costs. What is worse is that the human factor does not guarantee an error-free process to obtain a correct OO conceptual schema. Step (1) constitutes a manual, reverse-engineering process where the designer detects that the legacy table can be broken down into two classes: one containing the information about a performed task during a period of time and another one representing the collection of performed tasks for a specific customer, i.e., the *InvoiceLine* class and the *Invoice* class. In this step, works like those by [5], [6] and [7] can be applied to obtain OO models from relational schemas.

Once the OO model is complete, the relational database has to be generated (2). Here the designer can use many CASE tools such as Rational Rose, Together or System Architect in order to generate the new relational schema automatically. Despite obtaining a relational schema, these tools do not take into account legacy data.

The cumulative experience of the maintenance company is collected in its database and, it is expected to be preserved in the new database (3). Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done

by means of SQL statements or user defined scripts which can be executed on the database. Although ETL tools provide friendly interfaces to migrate data between databases, DB administrators must write migration code manually, and this is very costly in terms of people and time.

In next sections, we explain the MOMENT Framework and the way in which its application provides a more efficient solution to this reengineering problem.

3. The MOMENT Framework

The MOMENT (MOment manageMENT) Framework is a modular architecture divided into the three traditional layers: interface, functionality and persistence. In each one of them, the environment benefits from mature tools, such as graphical CASE tools at the interface layer, term rewriting systems at the functionality layer, and RDF repositories at the persistence layer. Hence, the MOMENT Framework aims at using the best features of each environment, bringing industrial modeling tools closer to more formal systems. Figure. 2 shows an overview of the MOMENT Framework.

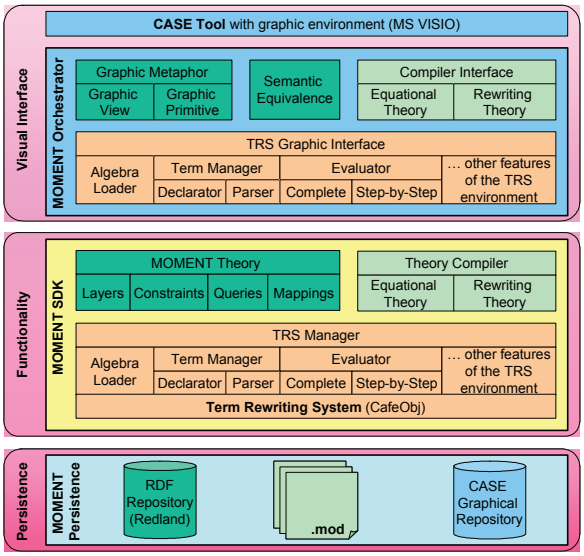


Figure 2. The MOMENT Framework

The functionality layer permits the representation of models and the performance of transformations over them. The core of the functionality layer is a module called *MOMENT Theory*, which allows model representation and manipulation by means of an algebraic approach. We use the expressiveness of the algebra that the platform is based on to define and represent a model as an algebraic term. This algebra represents models by means of terms of a *sort* called *Schema*. These terms are made up of by concepts and properties. The concepts are the main entities of a model, and the properties either describe them with values or establish relationships between them. The properties contain information about cardinality, indicating how many concepts can be related to the owner of the property.

The MOMENT platform uses several metadata layers to describe any kind of information including new metadata types. This architecture is based on both the classical four-layer metamodeling architecture (following standards such as ISO [8] and CDIF [9]) and on the more modern four-layer framework proposed in the MOF specification [3]. In our work, we divide the platform into four abstract layers:

- The M0-layer collects the examples of all the models, i.e., it holds the information that is described by a data model of the M1-layer.
- The M1-layer contains the metadata that describes data in the M0-layer and aggregates it by means of models. This layer provides services to collect examples of a reality in the lowest layer.
- The M2-layer contains the descriptions (meta-metadata) that define the structure and semantics of the metadata located at the M1-layer. This layer groups meta-metadata as metamodels. A metamodel is an "abstract language" that describes different kinds of data. The M2-layer provides services to manage models in the next lower layer.
- The M3-layer is the platform core, containing services to specify any metamodel with the same common representation mechanism. It is the most abstract layer in the platform. It contains the description of the structure and the semantics of the meta-metadata, which is located at the M2-layer. This layer provides the "abstract language" to define different kinds of metadata.

The *MOMENT Theory* module also provides a mechanism to define transformations between metamodels. The *TRS Manager* module wraps a TRS, which carries out the model transformation by applying a set of rewriting rules automatically. We have used the CafeOBJ environment as TRS [10]. The *Theory Compiler* module permits the compilation of the algebraic specification of a metamodel into a theory based on equational logic. It also compiles the defined mappings between the elements of the metamodels into a theory based on rewriting logic in order to perform the model transformation on the wrapped TRS. Some of these modules have been developed using the functional language F# [11], which provides convenient features to work with algebraic

specifications and with imperative programming environments such as .NET technology. A combination of functional languages and algebraic specification languages has permitted us to reach our goals. On the one hand, the MOMENT algebra is implemented in F#, which provides efficient structures for navigation and specification manipulation. On the other hand, a TRS provides a suitable environment to support automatic model transformation.

4. Legacy Systems Recovery Process in MOMENT

MDA raises the level of abstraction in the software development process by treating models as primary artifacts. Models are defined using modeling languages, but when those languages are intended to be used for anything more sophisticated than drawing pictures, both their syntax and their semantics must be specified. In this case, the use of formal languages usually involves dealing with their complex syntax, making them unpopular in industry. In this sense, the MOMENT Framework is user-friendly and permits the use of formal techniques from well-known CASE tools to both define models by means of algebraic specifications and to perform model transformations using rewriting logic [12].

Rebuilding a legacy system into a semantically equivalent one with a new technology can be treated as a model transformation problem in the MDA context. Our tool provides a generic model transformation mechanism that has been applied to recover legacy databases. The reengineering process is composed of two steps:

a) A data reverse engineering process that extracts an abstract description from the legacy system database in order to know its structure and its behavior. Changes can be applied to it in order to adapt the systems to new requirements or to new technologies, such as the change between the structured paradigm and the OO paradigm. Our tool recovers a legacy database obtaining the static component of an equivalent OO conceptual schema using formal methods.
b) A forward engineering process that generates the software application (its structure in our case) based on a specific technology from the abstract description extracted from the legacy system. We use the Rose Data Modeler add-in [13] of the Rational Rose tool case to generate a new relational schema from the OO conceptual schema.

Our tool also allows for data migration from the legacy database to the recently generated one, keeping the knowledge stored in the old database. The data reverse engineering process and the data migration process followed by our approach will reduce the time invested and the number of people involved in the data evolution process. This optimization is reached by the automatic tasks that are performed by the tool in three phases. Despite the fact that these tasks are performed automatically, the results can be freely modified by the analyst. In this case, the process is semi-automatic. The tool performs the following three phases in order to reach this goal:

a) A UML conceptual schema is obtained by applying a data reverse engineering process in order to recover a relational legacy database. Both relational and UML conceptual schemas are represented as terms of an algebra and the correspondences between terms are specified using term rewriting rules.
b) The rewriting rules applied in the first phase and the patterns used by the Rose Data Modeler add-in to generate the new relational schema are used to describe a data migration plan which is specified using a declarative language.
c) The data migration plan is compiled into DTS[1] packages whose execution automatically migrates data from the legacy database to the new one.

In the following sections, we present the three phases in more detail, illustrating their application by means of the example of the motivating scenario.

4.1. Reengineering Process

The input of this phase is a relational schema of a legacy database and is the input to the main tool. This phase generates two XML documents as output: one representing the generated UML conceptual schema in XMI format ([19]) and another one that contains the rewriting rules applied to obtain the final UML conceptual schema. In this phase the process that produces these outputs follows three steps:

a) The reading of the relational schema of the legacy database. The access to the relational schema is performed by means of an API, named RSAO in Figure. 3, in order to access heterogeneous databases. The information that describes the legacy relational schema is used to instantiate elements of the relational schema at the M1-layer as a model. This phase also considers features of the old DBMS or other repository forms which do not allow for the

[1] Data Transformation Services (DTS) is a technology of SQL Server that allows the transfer of data among heterogeneous databases.

definition of constraints (either integrity or reference constraints). These constraints, implemented by code in the best case scenario, are not explicitly defined in the legacy database structure. Thus, user interaction may be necessary to provide additional information to obtain a complete relational conceptual schema. This extra information is added to the relational model obtained from the relational database by means of a graphical interface that hides the algebraic formalisms to the user.

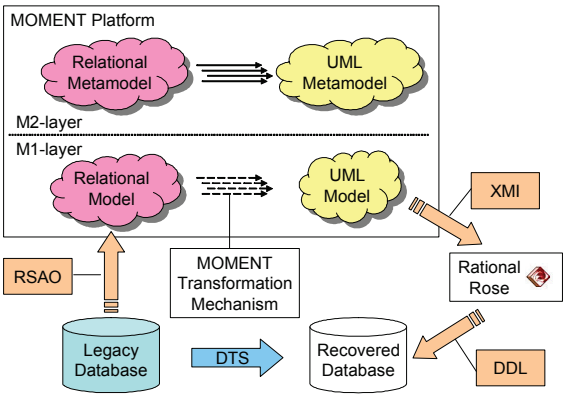


Figure 3. MOMENT solution to the legacy system recovering problem

b) Translation of the relational model into an UML term by means of the model transformation mechanism of the MOMENT Framework. Here the user may decide to apply other rewriting rule than the default rules chosen by the TRS in order to generate a more semantically accurate UML term. The applied rewriting rules in the translation process are written down to a XML document that will be used in the second phase.

c) The UML model obtained from the MOMENT transformation is stored at the M1-layer of the platform. It can be exported in an XMI document, which can be imported in turn by most of the CASE tools that manage UML diagrams, such as Rational Rose. These CASE tools provide support to manipulate a UML model and many other services such as generate its code or its database. Although the generation of the relational schema can be treated as a model transformation issue in the MOMENT platform, we also allow the communication with other CASE tools by means of XMI documents to benefit from their modeling features.

In this section, we present the way in which the MOMENT model transformation mechanism is applied to generate a UML model from a relational schema, providing formal support for generic model transformation in the MDA context. First, we explain a general overview of the transformation mechanism, and later, we focus on the most formal phases of the process.

4.1.1. Overview of the MOMENT model transformation process

Transforming any model using the MOMENT Framework constitutes a process that is detailed in Figure. 4. To obtain the corresponding UML model from the relational schema of the motivating example, we perform the following steps:

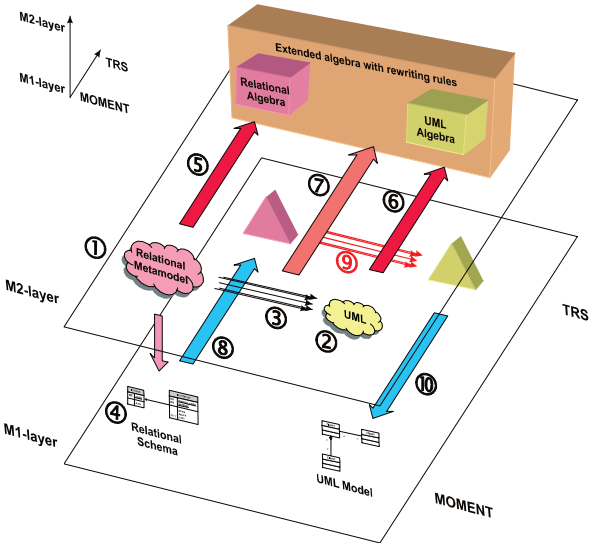


Figure 4. Overview of the MOMENT model transformation mechanism

– (1) and (2): We specify both relational and UML metamodels, respectively, at the M2-layer of the MOMENT platform using the operations of the MOMENT algebra. Each one of the metamodels is a schema made up of concepts, which describe the main entities of the ontology, and by properties, which describine the concepts by specifying values and establishing relationships between them. These algebraic specifications are performed through visual wizards that are embedded in a specific CASE tool to disguise the equational logic formalism.

– (3): Mappings are specified between the concepts of both metamodels at the M2-layer by means of a script language, indicating semantic relationships. There are two kinds of equivalence mappings that can be expressed in this language:

a) Simple mappings, which define a simple correspondence between two concepts that belong to different metamodels; for instance, between a table and a class, or between a column of a table and an attribute of a class.
b) Complex mappings, which define correspondences between elements of a source metamodel and a target metamodel. These mappings relate two structures of concepts that represent a similar semantic meaning. For instance, to define an equivalence relationship between a foreign key of the relational metamodel and an association of the UML metamodel, we have to relate the foreign key, the unique constraint and the not null value constraint concepts to the association concept. This is because all three of these concepts of the relational metamodel provide the necessary knowledge to define an association between two classes in the UML metamodel, such as the cardinalities of the association.

– (4): The original relational schema is specified by means of concepts and properties in a schema of the M1-layer of the MOMENT platform. Both concepts and properties are instances of the elements of the relational metamodel defined in step 1.
– (5) and (6): Both relational and UML metamodels, respectively, are compiled into algebraic theories by means of the *Theory Compiler* module of the Framework. The compilation uses the concepts to define the sorts of the new theory and the properties to define constructors and query operators. The generated theories are interpreted by the CafeOBJ TRS, providing the respective algebras to define models in the TRS as algebraic terms.
– (7): The semantic mapping that is specified between the concepts of both metamodels at the M2-layer is also compiled into another theory that extends the theories described above with a set of rewriting rules. This theory indicates how to transform a model of the source metamodel (relational metamodel) into a new model of the target metamodel (UML) in an automatic way.
– (8): The original relational schema, which is defined in step (4) at the M1-layer of the MOMENT platform, is compiled into a term of the relational algebra in the CafeOBJ TRS by means of the *Term Manager* module of the Framework.
– (9): The TRS evaluates the term that represents the initial relational schema in the algebra obtained in step (7). The user can manage this process through the *Evaluator* module of the Framework. The evaluation process can be carried in a step-by-step mode or in only one step with the full-evaluation mode, benefiting from the evaluation features of the TRS. The TRS reduces the initial term by applying the rewriting rules obtained in step (7), generating a term of the target algebra.
– (10): This is the last step of the model transformation process. It parses the obtained term in step (9), defining a model in the M1-layer as an instance of the target metamodel defined at the M2-layer. There, it is disguised with the visual metaphor associated to the target metamodel in the graphical CASE tool.

Previously to the transformation process, the source and target metamodels (step (1) and (2)), and the semantic mappings between the elements of both metamodels (step (3)) must be defined on the platform. In the model transformation process, the user only interacts with the MOMENT platform when defining the initial model (step (4)). The other steps are automatically carried out by the Framework, although the user can participate in the evaluation process by specifying the rewriting rules to be applied by the TRS at each step of the term reduction.

In the following sections, we explain phases (5), (6), (7), (8) and (9) in more detail, indicating how the TRS is able to perform model transformations providing formal support to the objectives of MDA.

4.1.2. Compilation of equational logic based theories

The relational and UML metamodels defined at the M2-layer of the MOMENT platform are compiled into theories based on equational logic in steps (5) and (6), respectively. The compilation of MOMENT metamodels into equational theories uses the concepts of the metamodel to obtain the sorts of the theory; for instance, the sorts Table, Field, ForeignKey for the relational metamodel, as well as the identifiers for these sorts, i.e., the sorts TableId, FieldId and ForeignKeyId. The properties of a MOMENT metamodel provide information about the structure of the term of a sort by means of the cardinalities. Thus, when a concept *A* is related to a concept *B* by means of a property that has cardinality 1..1, the constructor of the sort *A* looks like this: *op a _ : B → A*. Nevertheless, if the minimum cardinality is zero or the maximum cardinality is *many*, then the constructor for a term of the sort *A* looks like this: *op a _ : ListB → A*, where *ListB* is a *sort* that permits the definition of lists, whose items are terms of *sort B*. As CafeOBJ belongs to the OBJ language family, it permits equational specification through several equational theories, such as associativity, commutativity, identity, idempotence and combinations

between all these. This feature is reflected at the execution level by term rewriting by means of such equational theories.

Figure. 5 shows the constructors of the compiled theory for the relational metamodel; and Figure. 6 shows the constructors for the UML metamodel. We obviate the definition of sorts and other constructors in the theory, as well as the definition of query operators, focusing on the elements of the metamodels that permit us to illustrate the example. We must point out that the constructors obtained for the UML theory permit us to define terms that represent UML-compliant models.

```
-- FIELD:  id, type, is nnv, is pk, tableid, dbid
op field _ _ _ _ _ _ : FieldId Datatype Bool Bool TableId
DatabaseId ->
Field {constr}
-- FOREIGNKEY:  id, field list, related table, is unique, table that
-- contains the fk, database
op foreignKey _ _ _ _ _ _ : ForeignKeyId ListField TableId Bool
TableId DatabaseId -> ForeignKey {constr}
-- TABLE
op table _ _ _ _ : TableId ListField ListForeignKey DatabaseId ->
Table {constr}
-- DATABASE
op database _ _ : DatabaseId ListTable -> Database {constr}
```

Figure 5. Part of the relational theory

```
-- ATTRIBUTE:  id, type, required, constant, identifier, ClassId,
-- schemaid
op attribute _ _ _ _ _ _ _ : AttributeId Datatype Bool Bool Bool
ClassId
SchemaId -> Attribute {constr}
-- ASSOCIATION
op association _ _ _ _ : AssociationId AssociationEndId
AssociationEndId SchemaId -> Association {constr}
-- ASSOCIATIONEND: id, id of the class, id of the association,
-- isNavigable, ordering, aggregation, min card., max card,
-- changeability, visibility, id of the schema
op associationEnd _ _ _ _ _ _ _ _ _ _ _ : AssociationEndId
ClassId AssociationId Bool OrderingKind AggregationKind
Cardinality Cardinality ChangeableKind VisibleKind SchemaId ->
AssociationEnd {constr}
-- CLASS
op class _ _ _ : ClassId ListAttribute SchemaId -> Class {constr}
-- OOSCHEMA
op ooSchema _ _ : SchemaId ListClass -> OOSchema {constr}
```

Figure 6. Part of the UML theory

4.1.3. Compilation of rewriting logic-based theories

To transform the relational schema of the example, semantic mappings have been defined between the concepts of both source and target metamodels in step (3). These mappings are compiled into an algebra that extends both relational and UML algebras (steps (5) and (6)) with a set of rewriting rules describing the guidelines for the model transformation. These rules are automatically applied by the TRS rewriting the initial term into a term of the target algebra. The new algebra constitutes the context where semantical relationships between the source and target ontologies are defined. To allow the transformation process, the new algebra must relate the *sorts* of the initial algebra (relational metamodel) to the *sorts* of the target algebra (UML metamodel). Relationships between the sorts of both algebras result in a subsort order that involves all the sorts. For instance, in the example, the sort Table is a subsort of the sort Class, indicating that a class can take the place of a table that was there before. Subsort relationships affect all the sorts of both algebras, even identifiers and lists, because they are the related concepts in the MOMENT algebra.

The properties that relate concepts in the MOMENT algebra define a canonical order among the sorts of the compiled algebras. This order is taken into account to generate the rewriting rules. We present the rewriting rules that are applied to the relational schema of the example to obtain a semantically equivalent UML model in CafeOBJ syntax:

a) Field

A field of a table becomes an attribute of a class in the term that represents a UML model. The rule reuses the features of the field (datatype, whether it is null or not and whether it is primary key) to generate an attribute. Field features also indicate the attribute datatype, whether it is required or not and whether it is the identifier of the class to which it belongs:

op field _ _ _ _ _ _ : FieldId Datatype Bool Bool TableId DatabaseId -> Attribute

eq field FI D NNV PK TI DBI = attribute FI D NNV false PK TI DBI .

b) Foreign Key

A foreign key can define an association between two classes in the UML context. The following rule is applied when the foreign key is unique and not null, obtaining an association 1..1 - 0..* between the classes generated from the related tables.

op foreignKey _ _ _ _ _ _ : ForeignKeyId ListAttribute TableId Bool TableId DatabaseId -> ListClass

ceq foreignKey FkI LA RTI U TI DBI = (association FkI TI RTI DBI) (associationEnd TI TI FkI true unordered aggregate card 0 many frozen public DBI) (associationEnd RTI RTI FkI true unordered none card 1 card 1 frozen public DBI) if U and isRequired (LA) .

c) Table

A table becomes a class. The rewriting rules must take into account the fact that a table is made up of fields and foreign keys, so that a field will become an attribute of the new class and a foreign key will

become a set of elements of the UML model, i.e., an association and two association ends, according to the UML metamodel.

op table _ _ _ _ : TableId ListAttribute ListClass DatabaseId -> ListClass

eq table TI LA nilForeignKey DBI = (class TI LA DBI) .

eq table TI LA LC DBI = (class TI LA DBI) LC .

d) Database

Finally, a database is rewritten into a term of the sort OOSchema, representing the target UML model, by means of the following rule:

op database _ _ : DatabaseId ListClass -> OOSchema

eq database DBI LC = ooSchema DBI LC .

Relational Schema (a)

```
database 'InvoiceRS'
((table 'Invoice'
  ((field 'code' integer true true 'Invoice' 'InvoiceRS')
    (field 'date' datetime false false 'Invoice' 'InvoiceRS'))
  nilForeignKey
  'InvoiceRS')
(table 'InvoiceLine'
  ((field 'code' integer true true 'InvoiceLine' 'InvoiceRS')
    (field 'number' integer true true 'InvoiceLine' 'InvoiceRS')
    (field 'description' string false false 'InvoiceLine' 'InvoiceRS')
    (field 'date' datetime true false 'InvoiceLine' 'InvoiceRS')
    (field 'hours' decimal true false 'InvoiceLine' 'InvoiceRS'))
  ((foreignKey 'FK-InvoiceLine-Invoice'
      ((field 'code' integer true true 'InvoiceLine' 'InvoiceRS'))
      'Invoice' true 'InvoiceLine' 'InvoiceRS'))
  'InvoiceRS'))
```

```
OOSchema 'InvoiceRS'
((class 'Invoice'
  ((attribute 'code' integer true false true 'Invoice' 'InvoiceRS')
    (attribute 'date' datetime false false false 'Invoice' 'InvoiceRS'))
  'InvoiceRS')
(class 'InvoiceLine'
  ((attribute 'code' integer true false true 'InvoiceLine' 'InvoiceRS')
    (attribute 'number' integer true false true 'InvoiceLine' 'InvoiceRS')
    (attribute 'description' string false false false 'InvoiceLine' 'InvoiceRS')
    (attribute 'date' datetime true false false 'InvoiceLine' 'InvoiceRS')
    (attribute 'hours' decimal true false false 'InvoiceLine' 'InvoiceRS'))
  'InvoiceRS')
(association 'FK-InvoiceLine-Invoice' 'a' 'b' 'InvoiceRS')
(associationEnd 'a' 'InvoiceLine' 'FK-InvoiceLine-Invoice' 'InvoiceRS')
(associationEnd 'a' 'Invoice' 'FK-InvoiceLine-Invoice' 'InvoiceRS'))
```

UML Model (b)

Figure 7. Original term representing the source relational schema (7.a), and the generated term representing the target UML model (7.b).

4.1.4. Term rewriting process

Step (8) compiles the relational schema defined in the M1-layer of the MOMENT platform, obtaining the algebraic term in Figure. 7.a. The TRS applies the rewriting rules specified above to this initial term, obtaining a term of the target algebra (UML), shown in Figure. 7.b. This term is parsed and is defined as a UML model in the M1-layer of the MOMENT platform. There, it is automatically related to graphical pictures in a specific CASE tool.

During the term rewriting process, some additional information could be required in order to perform a correct transformation, as the metamodels do not have the same expressive power. For instance when a transformation case has not been taken into account or when several rewriting rules can be applied to the source model. In these cases a visual wizard helps the user to chose one option or even to add a new transformation rule, providing a visual interface for the CafeOBJ interpreter.

To benefit from the MOMENT features, we have integrated the functionality of the MOMENT Framework into a visual modeling environment [14]. In this way, we can relate algebraic specifications to visual notations so that the user can use these graphics to build a model. The CASE tool we have chosen is MS Visio [15]. We have developed an add-in that permits the definition of metamodels with concepts and properties. Figure. 8 shows the interface that permits the definition of the graphical symbol of a class in the MOMENT platform.

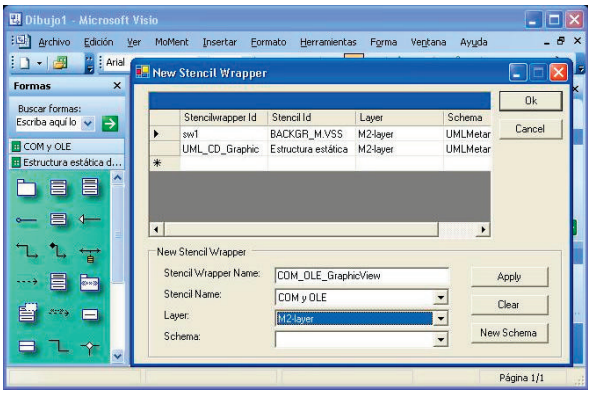


Figure 8. Visual interface to define a graphical primitive algebraically

In visual CASE tools, models are usually defined by dropping graphical primitives on a sheet where the model is defined. By means of the developed add-in, dropping a primitive on the sheet does not only add a figure to the model but it also defines it algebraically, specifying the model so that it can be manipulated afterwards.

5. Relational Migration Plan Generator

This phase generates a migration plan that specifies what information must be copied from the legacy database to the database of the new OO application. Its

inputs are two XML documents that contain the mappings:

- between elements of the legacy relational schema and elements of the recently generated OO conceptual schema.
- between elements of this OO conceptual schema and elements of the new relational schema generated by the Rose Data Modeler add-in.

The migration plan generator applies a set of patterns to the input correspondences and produces a migration plan that is specified using a relational declarative language indicating what information has to be copied from the legacy database to the new one. The use of a declarative language provides independence from the specific DBMSs used for supporting the databases. Additionally, this phase checks the constraints of the target database in order to avoid constraint violation.

5.1. Relational Migration Plan

A relational migration plan specifies the actions that must be performed in order to copy data from the legacy database to the new database, generated from the recovered OO conceptual schema. The migration plan consists of a set of migration modules. There exists one migration module for each specific table of the target database. Therefore, a migration module assigns a view over the legacy database to a target table indicating where to find the source data.

To specify the data copy process, a migration module contains a set of mappings between columns of the source view and the target table. Those mappings constitute the migration expressions that can be used in a migration plan and they are specified by means of the relational declarative language.

The automatic generation of the migration plan considers its structure and its contents. Thus, two kinds of patterns are used: migration patterns and migration expression patterns. Migration patterns generate the structure of the migration plan following the ordering of the tables of the new database. Migration expression patterns generate their content by means of migration expressions that represent mappings between columns of the relational tables.

The migration plan generator gets the applied rewriting rules of the reengineering process from the two input XML documents, one from the data reverse engineering phase and another one from the Rose Data Modeler. These rules provide enough information to determine how many migration modules are needed and which tables of the legacy database form the source view for each module. Thus, the generator constructs the migration modules by applying the migration patterns. Then, it applies the migration expression patterns in order to link attributes of the source view with attributes of the target table in each migration module, reflecting the generation process followed to obtain the target database. Once the migration plan is finished, the generator writes it in an XML document, ready to be compiled into a specific technology in order to perform the data migration.

6. Migration Plan compiler

This phase performs the compilation of the migration plan to a specific technology and its execution in order to carry out the physical data migration. We use the Data Transformation Services (DTS) of Microsoft SQL Server. This tool allows data migration between heterogeneous relational DBMS by applying data transformation services in order to fulfill target database requirements.

The compiler receives an XML document that describes the migration plan from the second phase of RELS and obtains a set of DTS packages that are able to perform the specified migration plan between the legacy database and the new one.

7. Related works

The MétaGen project [16] has dealt with model engineering since 1991, aiming at a fully automatic generation of a conventional application from a description given by its intended user. Such a description is performed by means of PIR3, a variant of what is known in the Database community as Entity-Relationship Model. In [17], Revault et al. compared three metamodeling formalisms to share the experience they acquired during the MétaGen project. In that paper, they presented a way to transform MOF-based metamodels into PIR3-based metamodels so that the metamodels could benefit from the MétaGen tools. This proposal constrains the expressivity of the source metamodels because the Object-Oriented paradigm is richer than the Entity-Relationship paradigm. Our model management approach supports the definition of several metamodeling languages such as MOF or PIR3, considering them as models at the same abstraction level and using generic operators. This makes automated transformations between models of both metamodels possible, without loss of expressivity.

XML [18], the standard for data communication between applications is also used to represent models and metamodels by means of the XMI specification [19]. MOF defines a meta-metamodel, while XMI

indicates the physical representation for the metamodels and the models that can be defined with it. In the model transformation field, there is a XML specification that allows the transformation of a XML document into another one, called XSLT [20]. It could be used to transform models that are represented in the XML format. Comparing XSLT to term rewriting systems, there are some differences that should be pointed out:

- Although XSLT is said to be a declarative language, control instructions, such as jumps and loops, can be used to guide the transformation process. In contrast, a term rewriting system takes over the transformation rule evaluation process.
- Writing an XSLT program is a long and painful process which implies poor readability and high maintenance cost for associated programs. Also, writing an XSLT program requires good skills in the MOF and XMI specification, because when using XPATH and XSLT, the developer must take into account the structure of models which depends on metamodels. These metamodels are in turn widely influenced by MOF and XMI. By expressing metamodels as algebras, we can deal with a more specific syntax that reflects their semantics using the algebras as domain specific languages. Therefore, writing models (and consequently transformation rules) becomes easier and more comprehensible.
- Transformation rules in XSLT are applied without taking into account the target XML schema (metamodel when transforming models), implying a posterior checking to determine whether the obtained document is a valid XML document that conforms to the target schema. Rewriting rules in an algebra take into account the source and the target algebras so that a posterior checking is no longer needed.
- Executing an XSLT program is not user-friendly for model transformation because there are no error messages to advise the user about an incorrect transformation.

The MTRANS Framework [21] provides an abstract language to define transformation rules that are compiled to XSLT. Even though this language is more compact and easier to understand than XSLT, it still keeps instructions to manage the transformation rules evaluation. Nevertheless, transformations using XML technology imply the use of standard specifications that are industrially supported while term rewriting systems usually remain within the field of research.

8. Conclusions

Nowadays, software applications have become complex combinations of technology, which have to be well understood in order to manage them. The development of software artifacts involves models that can be mixed with others to obtain an entire system from partial views or that can be interconnected with others in order to guarantee both interoperability in a distributed environment and their implementations.

MDA raises the level of abstraction in the software development process by treating models as primary artifacts. MDA potentially covers the modeling of all aspects of a system throughout its life cycle, making software development processes easier and more automated.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. The MOMENT Framework benefits from the best features of current visual CASE tools and from the main advantages of formal environments such as term rewriting systems, combining both industrial and research features.

In this paper, we have presented the generic model transformation mechanism provided by the MOMENT Framework, focusing on the use of the CafeOBJ Term Rewriting System to perform automatic translations of models. This mechanism has been applied to legacy relational database recovering in the MDA context, obtaining a UML model from a relational schema. The functionality of TRSs permits us to deal with model transformation from a more abstract point of view, since the application of rewriting rules can be performed in a transparent way. This fact allows us to focus all the efforts on the specification of models without having to take the evaluation logic into account. Our approach constitutes an algebraic baseline to cope with the future model transformation language QVT, providing a user-friendly environment to manipulate models from a visual CASE tool [14]. In [22], we present the fundamental mainstay on which we have built our MOMENT platform taking into account our previous experience in the industrial project RELS, a tool for the recovery of legacy systems.

Currently, we are working with transformations between relational schemas and UML models. In the near future, we will also take into account software architecture specifications by means of PRISMA ADL [23], studying semantic interoperability between software architectures and other types of software artifacts represented through UML models.

9. Acknowledgments

This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

10. References

[1] Ward, M.P., and Bennett, K.H., "Formal Methods to Aid the Evolution of Software"; *Journal of Software Maintenance: Research and Practice*, Vol 7, no 3, May-June 1995, pp 203-219.

[2] OMG, "The Model-Driven Architecture, Guide Version 1.0.1", OMG Document: omg/2003-06-01. Available from www.omg.org

[3] OMG, "Meta Object Facility 1.4", OMG Document: formal/02-04-03. Available from www.omg.org

[4] OMG, "MOF 2.0 Query/Views/Transformations RFP", OMG Document ad/2002-04-10. Available from www.omg.org

[5] Hainaut, J. L., Henrard, J., Roland, D., Englebert, V., and Hick, J.M., "Structure Elicitation in Database Reverse Engineering", WCRE'96 3 (1996), 131-140.

[6] Premerlani, W.J., and Blaha, M.: "An approach for reverse engineering of relational databases"; *Communications of the ACM* 37,5 (1994), 42-49.

[7] Ramanathan, S., and Hodges, J.: "Reverse Engineering Relational Schemas to Object-Oriented Schemas"; Technical Report MSU-960701, Mississippi State University, Mississippi, 1996.

[8] ISO/IEC 10746-1, 2, 3, 4 | ITU-T Recommendation X.901, X.902, X.903, X.904, "Open Distributed Processing – Reference Model". OMG, 1995-96.

[9] CDIF Technical Committee, "CDIF Framework for Modeling and Extensibility", Electronic Industries Association, EIA/IS-107, January 1994. See http://www.cdif.org/

[10] Razvan Diaconescu, and Kokichi Futatsugi, "An overview of CafeOBJ", *Electronic Notes in Theoretical Computer Science* vol. 15: (2000)

[11] Microsoft Research (Don Syme), "The F# official web site": http://research.microsoft.com/projects/ilx/fsharp.aspx

[12] N. Martí-Oliet and J. Meseguer, Rewriting logic: roadmap and bibliography, *Theoretical Computer Science*, vol. 285, issue 2 (August 2002), pp. 121-154.

[13] Boggs, W., Boggs, M.: "Mastering UML with Rational Rose 2002"; Sybex. January 2002.

[14] A. Boronat, J. Á. Carsí, I. Ramos, "An Architecture for the Definition of Graphic Metaphors for Metamodels", Software Engineering and Databases Workshop (Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2004). Málaga, 10-12 November 2004 (in Spanish).

[15] Graham Wideman, "Microsoft Visio 2003 Developer's Survival Pack", Trafford, 2004.

[16] N. Revault, H.A. Sahraoui, G. Blain and J.F. Perrot, A Metamodeling technique: "The MÉTAGEN system", TOOLS 16: TOOLS Europe'95, Prentice Hall, pp. 127-139. Versailles, France. Mar. 1995.

[17] N. Revault, X. Blanc and J. F. Perrot, "On Meta-Modeling Formalisms and Rule-Based Model Transforms", Comm. at workshop, In Iwme'00 workshop at Ecoop'00, Jean Bézivin and Johannes Ernst (ed), Sophia Antipolis and Cannes, France, June, 2000.

[18] Elliotte Rusty Harold, "XML Bible", IDG Books Worldwide, 1999.

[19] OMG, XML Metadata Interchange (XMI) Specification, OMG Document formal/02-01-01.

[20] W3C, XSL Transformations (XSLT) v1.0. W3C Recommendation, http://www.w3.org/TR/xslt, Nov. 1999.

[21] M. Peltier, J. Bézivin, and G. Guillaume, "MTRANS: A general framework, based on XSLT, for model transformations". In WTUML'01, Proceedsings of the Workshop on Transformations in UML, Genova, Italy, Apr. 2001.

[22] A. Boronat, J. Pérez, J. Á. Carsí, and I. Ramos, "Two experiencies in software dynamics". *Journal of Universal Science Computer*. Special issue on Breakthroughs and Challenges in Software Engineering. April 2004.

[23] J. Pérez, I. Ramos , J. Jaén, P. Letelier, E. Navarro, "*PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*", 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 © IEEE Computer Society Press pp. 59-66.

Automatic Support for Traceability in a Generic Model Management Framework♦

Artur Boronat, José Á. Carsí, Isidro Ramos

Department of Information Systems and Computation
Polytechnic University of Valencia
Camí de Vera s/n
46022 Valencia-Spain
{aboronat | pcarsi | iramos}@dsic.upv.es

Abstract. In a MDA process, software artifacts are refined from the problem space (requirements) to the solution space (application). A model refinement involves the application of operators that perform tasks over models such as integrations and transformations, among others. We are working on a model management framework, called MOMENT (MOdel manageMENT), where model operators are defined independently of any metamodel in order to increase their reusability. This approach also increases the level of abstraction of solutions of this kind by working on models as first-class citizens, instead of working on the internal representation of a model at a programming level. In this context, traceability constitutes the mechanism to follow the transformations carried out over a model through several refinement steps. In this paper, we focus on the generic traceability support that the MOMENT framework provides. These capabilities allow the definition of generic complex operators that permit solving specific problems such as change propagation.

Keywords: Model-Driven Architecture, Model Management, traceability, software maintenance.

1. Introduction

Traceability is an important issue in environments where there is a process chain. In these cases, information about each step in the chain may be stored for further processing. For example, in the automotive industry, traceability makes recalls possible; in the food industry, it contributes to food safety; in the Software Engineering field, it provides support for requirements validation and improves the quality of the software development process.

In any scenario of the Software Engineering field, there is a manipulation of a software artifact. The capability of describing and querying the manipulation that has been performed on a specific artifact might be relevant to correlated tasks. However, traceability still remains in the background when software engineering problems are

♦ This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

Boronat, A., Carsí, J. A., Ramos, I. Automatic Support for Traceability in a Generic Model Management Framework. Proc. of the European Conference on Model Driven Architecture - Foundations and Applications. 2005.

solved. It is often misunderstood and burdensome due to the lack of tools that provide full automatic support for it [1, 2].

In the Model-Driven Architecture initiative [3] (MDA), a software artifact is viewed as a model. Typical tasks, such as code production, integration of applications, interoperability between applications, are performed on models directly. This allows the user to work at a conceptual level, and makes the identification of the elements needed to automate these tasks easier. These tasks are pervasive in many scenarios and are usually solved in an ad-hoc manner.

Following this model-driven approach, a new discipline, called Model Management, was proposed in [4]. This discipline considers models as first-class citizens and provides a set of generic operators to deal with them: *Merge*, *Diff*, *ModelGen*, etc. These operators provide a reusable solution to the tasks described above so that the user deals directly with models, rather than working on the internal representation of a model at a programming level. Several approaches to this discipline [5, 6] specify operators that are based on mappings to deal with models. A mapping is a relationship between an element of a domain model and an element of a range model that indicates that they represent the same element in different models. This means that mappings between two models must be explicitly defined in order to apply an operator to them.

Using our experience in applying the algebraic specification formalism to solve actual software engineering problems [7, 8], we are working on a model management framework called MOMENT (MOdel manageMENT). In our approach, we represent the relationships between two models in an implicit manner by means of an equivalence morphism that is defined between two metamodels. Our approach describes equivalence relationships between two models from a more abstract and reusable point of view. However, explicit mappings between two models are also beneficial when there is no definition of the equivalence morphism between two metamodels. We refer to mappings of this kind as traceability links.

In this paper, we focus on the automatic traceability support that is provided by our framework from a generic point of view. We define what a traceability model is in this setting and how they can be used to provide traceability support in many different scenarios: requirements, workflows, ontologies, etc. We also show how traceability support contributes to automate solutions such as the software maintenance.

The structure of the paper is as follows: Section 2 reviews the traceability management studies that have been performed in the Requirements Engineering field; Section 3 provides an example to illustrate the use of traceability; Section 4 provides an overview of our model management approach; Section 5 details the generic traceability support that is provided in our framework; Section 6 solves the problem of the case study with our model management operators; Section 7 presents some related work; finally, Section 8 summarizes the main features of our approach.

2 The Traceability Problem in Requirements Engineering

In Requirements Engineering, the IEEE Guide to Software Requirement Specifications [9] indicates that a software requirements specification is traceable if

the origin of each requirement is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

Based on this definition, Gotel and Finkelstein [10] described requirements traceability as the ability to describe and follow the life of a requirement, both forwards and backwards, through all the refinement steps in a software development process. Queries of this kind allow the user to know what refinements have been applied to a requirement (in a forward direction), and permit the identification of a requirement from a more specific software artifact such as code (in a backward direction). Therefore, traceability can be used for requirements validation and for providing support for software maintenance. Traceability also provides economic leverage as it details how the system has been developed and avoids the redundant development of certain parts.

To achieve traceability in a software development process, several tasks should be taken into account [2]:

1. Trace definition, to indicate the kinds of objects in our system that can be traced and what information is going to be defined in a trace.
2. Trace production, to indicate what activities, actions, decisions and events happening during software development generate traces for further use.
3. Trace extraction, to indicate how the traces produced in the previous task can be queried in order to achieve certain goals, such as requirements validation or software maintenance.
4. Trace verification, to maintain the integrity of the set of objects and traces.

There are many tools that provide requirements traceability management [11]. To provide efficient traceability management, these tools must resolve certain problems that are present in the industrial setting:

- The lack of a common guideline that describes how to define a traceability model through a well-defined metamodel and how to use it; for example, the way the UML standard provides support to object-oriented modeling. This is due to the variable nature of the traceability capture and use [2], which varies from one organization to another, from one project to another and even from one stakeholder to another.
- The *establish and end-user conflict* [10], where trace providers and users have different goals and priorities.
- The use of heterogeneous tools to define and manipulate the software artifacts involved in a software development process. Thus, the interoperability issue arises as an important feature to be taken into account in a traceability management tool.

These tools are also implemented in an ad-hoc way for the requirements traceability problem without taking into account that the same functionality can be used in other contexts.

3 A Software Maintenance Case Study: Change Propagation

In this case study, we use the change propagate scenario that was introduced in [5]. We illustrate it by means of a specific example shown in Fig. 1. We have defined the information structure of an application in a XML schema (XSD). To build a new

application that stores the information in a relational database, we reuse the metainformation that describes the XML schema. By applying a transformation[1] mechanism (step 1), we obtain the new relational database (RDB). The transformation mechanism also generates a set of links between the new generated RDB relational schema and the source XML schema in order to provide traceability support ($map_{XSD2RDB}$).

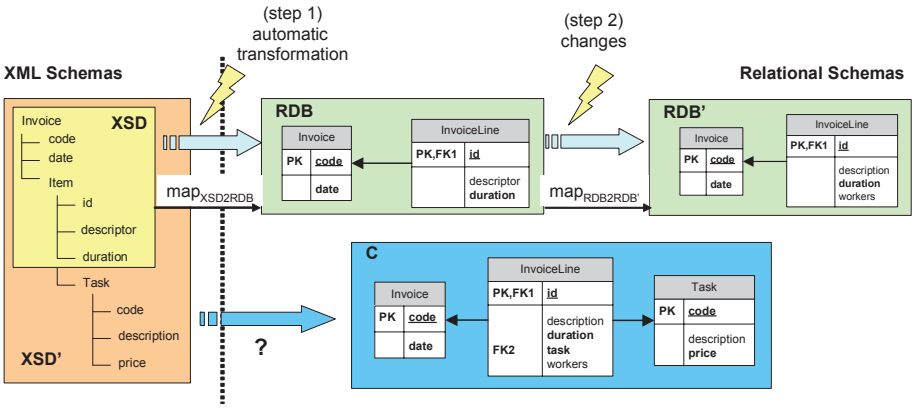


Fig. 1. An example of change propagation.

After obtaining a semantically equivalent relational database from the original XML schema, we continue with the development of the new system. This may involve changes in the application and in the database (step 2), obtaining the relational schema (RDB'). These changes are traced and stored by the tool that manages the model manipulation or by the user directly ($map_{RDB2RDB'}$).

Once the new system is developed, changes may occur in the requirements of the system, requiring modifications. It is easier to extend the XML schema than to modify the RDB database. At this point, the application of the transformation mechanism used in step 1 will discard the changes applied from RDB to RDB'.

A solution to this change propagation example can be performed by using model management operators, as shown in Section 6. In our approach, traceability links are used to automate the propagation of changes that were applied to the RDB relational schema, for the new system C.

4 The MOMENT Framework: a MOdel manageMENT Environment into Eclipse

The MDA initiative of the OMG consists of a family of standards that indicate how to define and how to use models to develop software applications in a MDE setting. Application integration and interoperability are two goals of this initiative, as indicated in the request for proposals for the new standard

[1] We use the term 'model transformation' for the mechanism that translates a model between two different metamodels.

Query/Views/Transformations [12]. Nevertheless, to achieve interoperability between applications, bridges built between them are still ad-hoc.

We are working on the application of the model management trend in the context of MDA. We have developed a framework, called MOMENT (MOdel manageMENT), which is embedded in the Eclipse platform. It provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF) [13]. EMF provides a close implementation to the MDA guidelines. This framework enables the automatic importation of software artifacts from heterogeneous datasources: UML models (by means of visual modeling environments), relational schemas of any relational database management system (through the Rational Rose tool) and XML schemas. Moreover, third-party researchers and developers are bringing new tools to work on ontologies through EMF [14, 15] and graphical Domain Specific Languages [16, 17]. Therefore, EMF has become an industrial framework for MDA.

4.1 Bridging the EMF and the Maude Technical Spaces

The concept of technical spaces (TS) was introduced by Kurtev et al. in the discussion on the problem of bridging different technologies [18]. A technical space is a working context with a set of concepts, a body of knowledge, tools, required skills, and possibilities [19]. For example, we use the EMF and Maude technical spaces for our framework. The EMF is characterized by its interoperability with industrial tools for solving actual Software Engineering problems. Maude constitutes the formal backbone for our model management approach.

The algebra of operators, which was proposed by Bernstein [20] to deal with models and mappings between models as first-class citizens, has been adapted and directly specified as a generic algebra by using the algebraic specification formalism Maude [21] in the MOMENT framework. This algebraic specification language belongs to the OBJ family, and its equational deduction mechanism animates the specification of an operator over a piece of data, providing the operational semantics for our model management operators. We have developed a plug-in that embeds the Maude environment into the Eclipse framework so that we can use it for our purposes.

In [7, 8], we envisioned the advantages of applying this formalism to solve actual problems in MDA such as model transformation. To fulfill this goal, we have defined two bridges between both technical spaces, at the M2-layer and at the M1-layer (using the Meta-Object Facility [22] terminology). Both of them permit the integration of MOMENT with EMF.

We have defined a projection mechanism at the M2-layer that obtains the algebraic specification[2] that corresponds to a specific metamodel automatically, by applying generative programming techniques. The inverse projection mechanism that obtains an EMF metamodel from an algebraic specification is not interesting in our tool,

[2] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel, they only permit the representation of information for model management issues.

because the algebraic specification must conform to several features in order to be used by our operators, and they should be automatically achieved. We also think that visual modeling environments are more suitable to define such metamodels.

At the model level, we have developed a bidirectional projection mechanism that permits us to project an EMF model as a term of an algebra and to project an EMF model from a term. In this case, the bidirectionality is needed to apply an operator to an input model, since the input model must be serialized as a term and the output term must be deserialized into an EMF model in order to be persisted.

4.2 Operators

In MOMENT, operators are defined in a parameterized module called MOMENT-OP. In this way, operators are defined generically. To apply these operators to specific models, this module must be instantiated by passing a metamodel as actual parameter. This task is automatically performed by the MOMENT tool.

To understand the solution that is given for the change propagation example in Section 6, we informally present some of the model management operators that we use in our approach by indicating their inputs, outputs and semantics:

1. *Cross* and *Merge*
   These operators correspond to well-defined set operations: intersection and disjoint union, respectively. Both operators receive two models (*A* and *B*) as input and produce a third model (*C*). The *Cross* operator returns a model *C* that contains elements that participate in both the *A* and *B* input models; while the *Merge* operator returns a model *C* that contains elements that belong to either the input model *A* or the input model *B*, deleting duplicated elements. Both operators also return two models of links (map_{AC} and map_{BC}) that relate the elements of each input model to the elements of the output model.
   Example: $<C, map_{AC}, map_{BC}> = Cross(A,B)$.
2. *Diff*
   This operator performs the difference between two input models (*A* and *B*). The difference between the two models (*C*) is the set of objects in model *A* that does not correspond to any element in model *B*.
3. *ModelGen*
   ModelGen performs the translation of a model *A*, which conforms to a source metamodel *MMA*, into a target metamodel *MMB*, obtaining model *B*. This transformation implies dealing with two metamodels. This is perfectly feasible in our approach due to the modularity and reusability that algebraic specifications provide. This operator also produces a model of links (map_{AB}) relating the elements of the input model to the elements of the generated model.
   Example: $<B, map_{AB}> = ModelGen_{MMA2MMB}(A)$.

5 Traceability Support in Model Management

All the definitions of Requirements Traceability stated in Section 2 have one feature in common: a trace provides information about a task that has been performed on a

source software artifact in a software development process, and relates it to the resulting software artifact. Traceability support must provide both the mechanism that is needed to define traceability links and the query functionality that permits link navigation. In this section, we define traceability in MDA through a model management lens, and we present a set of operators that provide traceability support in the MOMENT framework.

5.1 Generic Traceability Management

A MDA process consists of a sequence of operations performed over a set of models. These models conform to a metamodel and represent specific software artifacts. Operations such as model integration or model transformation can be directly supported by simple model management operators (Section 4.2). Other operations, such as the change propagation mechanism of the case study, can be specified as a complex operator made up of other operators.

Each simple operator carries out a manipulation over a set of input models. To fulfill this, the operator invokes a function that is defined at the metamodel level. The semantics of this function is defined axiomatically in equational logic, and each one of its axioms is called a manipulation rule. To register the task performed over a model, each operator automatically produces a set of links between the elements of a source model and the elements of the resulting model. Such links are stored as models and are used to provide support for traceability.

Following the model management approach, we define Generic Traceability Management as two main issues:

1. The definition of a traceability metamodel to indicate the information needed to link the elements of two different models that can belong to different metamodels in a specific context. The detail of the metamodel depends on the common understanding of the traceability management in a specific society. For example, a generic traceability metamodel may be described for the Requirements Engineering field, although it seems more feasible to define a traceability metamodel for each organization or even each project.
2. A mechanism to extract information from a traceability model independently of the metamodel used. This mechanism is made up of two kinds of operators:
   - Query operators that provide forward and backward navigation through a traceability model.
   - Traceability management operators to manipulate the traceability models in order to automate the reasoning over traceability links. For instance, the *Compose* operator permits chaining traceability links in order to make implicit traceability links explicit; and the *Match* operator permits the inference of traceability models between two models. Furthermore, a traceability model can also be manipulated by model management operators.

5.1.1 Definition of the Traceability Metamodel

To define the traceability metamodel, the user can use the UML notation. This work is done by the user for a specific working context. For the case study, we have specified a traceability metamodel which basically provides the constructs needed to

relate elements of a domain model to elements of a target model, independently of their metamodels. In Fig. 3, we show the part of the MOMENT framework metamodel that concerns the traceability support.

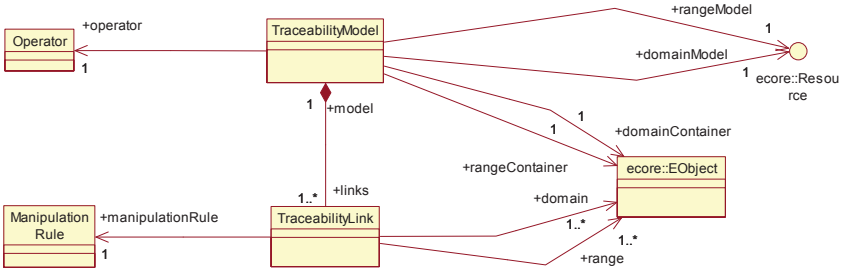


Fig. 2. The default traceability metamodel of the MOMENT framework.

In our specific metamodel, the *TraceabilityModel* class is the root element of the package. It allows us to define traceability models. An instance of the *TraceabilityModel* class contains: information about the storage of both the domain and the range model (represented by the interface *ecore::Resource*); which element of each model is the root (by means of the *domainContainer* and *rangeContainer* roles); the operator that has been applied to the domain model; and the links that constitute the traceability model.

The *TraceabilityLink* class indicates how to define a relationship between a set of elements of the domain model and a set of elements of the range model (by means of the *domain* and *range* roles). Each link is associated to the step of the model manipulation task that has produced it (through the *manipulationRule* role). In the metamodel of the figure, *ecore::Resource* and *ecore::EObject* refer to the interface *org.eclipse.emf.ecore.resource.Resource* and to the class *org.eclipse.emf.ecore.EObject*, respectively. The former permits access to a model stored physically. The latter permits access to any element of an EMF model so that any model defined by means of the EMF can be dealt with. The *Operator* class represents an operator that is defined algebraically in MOMENT. The *ManipulationRule* class defines the information needed to specify an axiom for the manipulation function used by the simple operators.

By applying the projection mechanism defined between the EMF TS and Maude TS, we obtain the algebraic specification of the traceability metamodel. This means that we can specify traceability models as sets of elements so that MOMENT operators can be used to manipulate them (*Merge*, *Cross*, *ModelGen*, …).

5.2 Traceability Operators

Once we have shown the definition of a specific traceability metamodel, we explain the generic traceability operators that are provided by MOMENT. The operators that provide support for traceability are defined generically in a parameterized algebraic specification, called *MOMENT-TRAC(Y :: BASICTMM)*.

Fig. 4 shows the elements involved in the parameter passing mechanism diagram. *BASICTMM* (BASIC Traceability MetaModel) is the algebraic specification of the

formal parameter, called theory in Maude. This theory declares some operators that guarantee the independence between the semantics of the generic traceability operators and the semantics of a specific metamodel. For example, an operator of this kind is *GetDomain*. It obtains the domain element of a traceability link independently of the syntactical representation of the link. Thus, the formal parameter behaves as an interface through which the generic operators can access the elements of a model that conforms to a specific traceability metamodel.

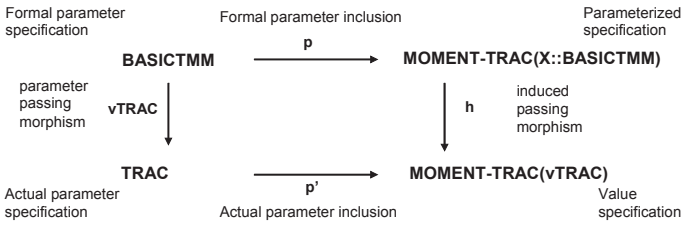


Fig. 3. The parameter passing diagram for the *MOMENT-TRAC(Y :: BASICTMM)* parameterized module.

TRAC is the algebraic specification obtained by the projection mechanism from a specific traceability metamodel. The *TRAC* specification constitutes the actual parameter for the *MOMENT-TRAC(Y :: BASICTMM)* module and defines the semantics of the operators that are only declared in the *BASICTMM* theory. The *vTRAC* view is the morphism that relates the elements of the *BASICTMM* formal parameter to the elements of the *TRAC* actual parameter.

The *MOMENT-TRAC(Y::BASICTMM)* parameterized algebraic specification contains the definition of the traceability operators that are independent of the specific TRAC traceability metamodel. The *MOMENT-TRAC(vTRAC)* value specification results from the instantiation[3] of the parameterized module with the specific *TRAC* traceability metamodel.

In this figure, p and p' are inclusion morphisms that indicate that the formal parameter specification is included in the parameterized specification, and that the actual parameter specification is included in the value specification, respectively. The h morphism is the induced passing morphism that relates the elements of the parameterized module to the elements of the *MOMENT(vTRAC)* value specification, by using the *vTRAC* parameter passing morphism.

The traceability operators defined in the *MOMENT-TRAC(X::BASICTMM)* parameterized module are classified in two groups: operators that provide support for navigability and operators that perform tasks on traceability models. In this paper, we focus on the first group of operators.

We define the operators that provide navigability through a traceability model with the following elements: two input models (A and B); a traceability model (map_{AB}) that relates the elements of the two input models and that has been automatically produced by an operator or manually produced by a user; a model (A') that is a sub-

[3] In the context of algebraic specifications, the instantiation of a parameterized module refers to the fact of passing an actual parameter to the parameterized module, obtaining the final value specification.

model of A (i.e. A' only contains elements that also belong to A); and a model (B') that is a sub-model of B. The traceability operators that are considered here are:

1. *Domain* and *Range*

   These operators provide the backward and forward navigation through a traceability model, respectively. Both operators obtain a model as output, which is not a traceability model.

   The operator *Domain* takes three models as input: a traceability model (map_{AB}), a domain model (*A'*), and a range model (*B'*). The operator navigates the traceability links of the traceability model that have elements of *B'* as target elements, and returns a sub-model of *A* (*A'*), as shown in Fig. 5.a.

   The operator *Range* also receives three inputs: a traceability model (map_{AB}), a domain model (*A'*), and a range model (*B*). This operator performs the opposite task to the previous one: it navigates the traceability links that have elements of *A'* as domain elements and returns a sub-model of the range model *B* (*B'*), as shown in Fig. 5.b.

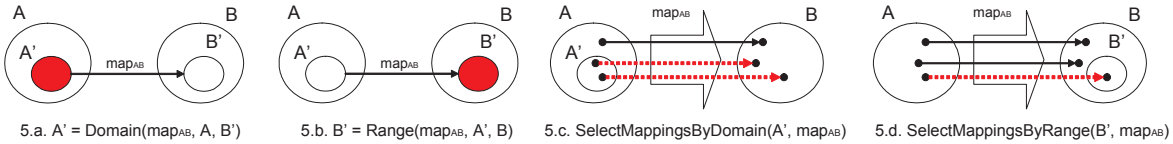


Fig. 4. Generic operators for traceability navigation.

2. *SelectMappingsByDomain* and *SelectMappingsByRange*

   These operators produce a traceability model as output and permit selection of parts of a traceability model.

   The operator *SelectMappingsByDomain* receives two input models: a domain model (*A'*) and a traceability model (map_{AB}). The operator extracts the traceability links of the map_{AB} traceability model that have elements of the *A'* model as domain elements and returns this sub-model. The traceability links that are added to the output traceability model are highlighted by a dotted line in Fig. 5.c.

   The operator *SelectMappingsByRange* receives two input models: a range model (*B'*) and a traceability model (map_{AB}). In this case, the operator extracts the traceability links of the map_{AB} traceability model that have elements of the *B'* model as range elements, and returns this sub-model, as shown in Fig. 5.d.

5.3 Process

Taking into account the process described in [2] to define a traceability model, we indicate how its tasks are performed in our tool:

1. Trace definition. Users can define their own metamodel to fit into a specific working context. Moreover the default traceability metamodel of the MOMENT framework can be used in its place. As seen above, the traceability metamodel can be defined using well-known graphical notations, such as UML, through EMF-compliant tools.
2. Trace production. By default, the traceability model is automatically generated by an operator when it performs a manipulation over a set of input models.

Nevertheless, traceability links can be defined manually by means of an editor generated from the traceability metamodel, following the EMF software development culture. Moreover, a traceability model can be inferred automatically between two models by using heuristics [23] or historical knowledge [24].

3. Trace extraction. The analysis of the knowledge provided by a set of traceability models can be useful to perform other tasks. The traceability operators are used to deal with this information in our framework. Such operators constitute an automatic reusable solution that provides support for traceability in many scenarios in the MDE field. Therefore, our framework provides automatic support for this step although the user has to reason about the extracted knowledge. In future works, heuristics may be applied in this step to achieve richer information.
4. Trace verification. The consistency of traceability models can be kept automatically when either their domain or their range model is modified, by means of the application of traceability operators. Consider that we have a domain model *A*, a range model *B* and a traceability model map_{AB} that has been defined between them. Three kinds of model modifications are available: addition of elements, modification of existing elements and deletion of elements. In the case of adding elements to a model, the traceability model remain consistent on the grounds that there is no connection between the new elements and the elements of the other model, unless this connection is defined explicitly afterwards. In the case of modifying and deleting elements of a model, links can be broken when the domain or the range elements are deleted. This problem is easily solved by using traceability operators. If we delete elements in model *A*, obtaining model *A'*, we can apply the *SelectMappingsByDomain* operator to obtain the new consistent traceability model $map_{AB'}$: $map_{AB'} = SelectMappingsByDomain(A', map_{AB})$.

6 Application to the Case Study

The problem explained in the case study can be simplified as shown in Fig. 6, where the $map_{XSD2RDB'}$ traceability model can be easily obtained from the $map_{XSD2RDB}$ and $map_{RDB2RDB'}$ traceability models by means of the *Compose* operator. Therefore, the problem can be enunciated as follows:

We have the following models: an original XML schema (XSD); a XML schema (XSD'), which has been evolved from XSD; a relational database RDB', which has been generated from the XML schema XSD and modified afterwards; and a traceability model between XSD and RDB' ($map_{XSD2RDB'}$). The goal is to obtain a relational database from the XML schema XSD' that preserves the changes applied to RDB'.

This problem can be solved by the following complex operator:

operator PropagateChanges(XSD, XSD', RDB', $map_{XSD2RDB'}$) =
$<Unmodified, map_{XSD2Unmodified}, map_{XSD'2Unmodified}> = Cross(XSD, XSD')$ (1)
$RDB'' = Range(map_{XSD2RDB'}, Unmodified, RDB')$ (2)
$<newXSD> = Diff(XSD', Unmodified)$ (3)
$<newRDB, map_{newXSD2newRDB}> = ModelGen_{XSD2RDB}(newXSD)$ (4)
$<C, map_{RDB''2C}, map_{newRDB2C}> = Merge(RDB'', newRDB)$ (5)
return (C)

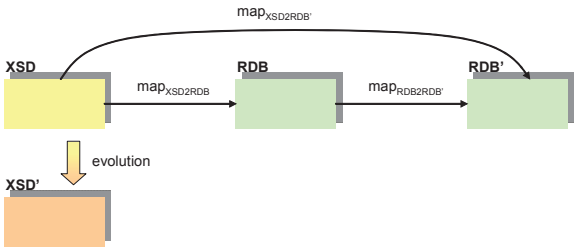


Fig. 5. Schematization of the case study problem.

This operator is made up of simple operators of the MOMENT algebra and the steps followed in the script are represented in Fig. 7. These steps are the following:

1. *Unmodified* is the part of the *XSD* model that remains unmodified in the *XSD'* model.
2. *RDB''* is the sub-model of *RDB'* that corresponds to the unmodified part of XSD'.
3. *newXSD* is the part of *XSD'* that has been added to the XSD model.
4. *newRDB* is the relational schema obtained from the translation of *newXSD* into the relational metamodel.
5. C is the final model obtained from the integration of the relational databases that we have obtained in steps 2 and 4.

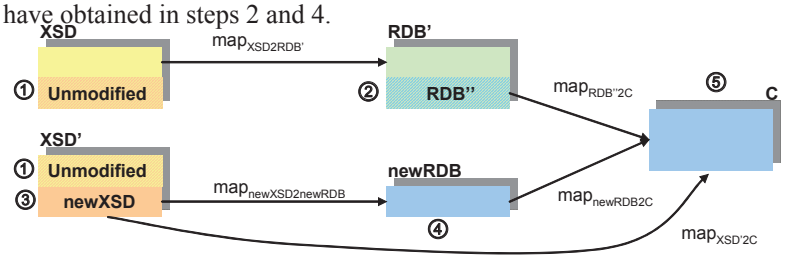


Fig. 6. Solution to the case study problem.

If we want to add traceability support to this operator to generate the traceability model that relates the XSD' model to the new model (C) as well, we only have to add the next step after step 5[4]:

$<map_{XSD'2C}, map_{mapUnmodified2C2mapXSD'2C}, map_{mapnewXSD2C2mapXSD'2C}> = Merge($
$Compose(Unmodified, SelectMappingsByDomain(Unmodified, map_{XSD2RDB'}), RDB'', map_{RDB''2C}, C),$
$Compose(newXSD, map_{newXSD2newRDB}, newRDB, map_{newRDB2C}, C))$

[4] In this step, the operator Compose ($<map_{AC}> = Compose(A, map_{AB}, B, map_{BC}, C)$) has five input parameters: three models (A, B, C), and two traceability models, which are defined

This step merges two traceability models: one is defined between the unmodified part of XSD' and C, and the other is defined between the new part of XSD' and C. This step merges both traceability models by means of the *Merge* operator, in the same way that any two models that belong to the same metamodel are merged. The model $map_{XSD'2C}$ has to be added as a return value in the script.

The resulting complex operator solves the change propagation problem of the case study independently of the metamodels involved so that we can apply it to any combination of metamodels, instead of using the XSD and the relational metamodels.

7 Related work

In the Model Management field, tools do not deal with traceability directly. They usually work on mapping models, which define equivalence relationships between the elements of two models so that a model management operator can be defined generically. Rondo [5] and [6] are good examples of this approach. For instance, Rondo's Merge operator permits the integration of two models. It receives two models (*A* and *B*) and a mapping model (map_{AB}) between them as inputs, and it produces the merged model *C* and two new mapping models (map_{AC} and map_{BC}): $<C, map_{AC}, map_{BC}> = Merge\ (A, B, mapAB)$.

In MOMENT, mapping models are introduced as traceability models. This is because operators do not have to rely on them to be applied to a set of models. In MOMENT, the traceability relationships between the elements of two models, which are needed to apply an operator to them, are defined between the elements of their corresponding metamodels axiomatically within the corresponding operators. The collection of equivalence relationships between two metamodels constitutes a morphism that can be reused for all the operators of the MOMENT algebra. This permits a clearer specification of complex operators. In MOMENT, the Merge operator is as follows: $<C, map_{AC}, map_{BC}> = Merge(A, B)$. Mapping models are produced by the application of a simple operator to a set of models and keep information about the manipulation task that has been performed to a model. Therefore, we deal with these mapping models from a traceability standpoint.

The Generic Model Weaver AMW [25] is a tool that permits the definition of mapping models (called *weaving models*) between EMF models in the ATLAS Model Management Architecture. AMW provide a basic weaving metamodel that can be extended to permit the definition of complex mappings. These mappings are usually defined by the user, although they may be inferred by means of heuristics, as in [23]. In MOMENT, such mappings are generated by model management operators automatically in a traceability model, and can be manipulated by other operators. We also support extension of the traceability metamodel. Although the simplicity of our initial traceability metamodel, it allows us to deal with complex operators satisfactorily.

---

between models *A* and *B* (map_{AB}) and between models *B* and *C* (map_{BC}). It concatenates both traceability models obtaining a new one that directly relates A to C.

8 Conclusions

In this paper, we have presented an overview of our model management approach, focusing on the automatic support that the MOMENT framework provides for traceability. To do this, we have based our approach on the traceability management studies that have been made in the Requirements Engineering field.

We have introduced the Generic Traceability Management concept in the MDA initiative through a Model Management lens. We have also discussed some operators and illustrated how they can be used to solve common software engineering scenarios, like the software maintenance case study presented here. The traceability support has been defined in the MOMENT framework generically so that it can be applied to any context (requirements, workflows, ontologies,…). It can also be customized with a specific traceability metamodel depending on the needs of each working context.

We provide a new vision of the traceability support with respect to previous model management approaches [5, 6], where all operators are based on mappings. In these approaches, the equivalence relationships between elements of two models are defined with specific explicit mappings at the model level. Such mappings are defined by the user or can be inferred by means of heuristics or historical knowledge. However the obtained mappings should be reviewed by a user, which can become burdensome when huge models are involved.

In MOMENT, such equivalence relationships are defined as morphisms between metamodels because algebraic specifications are used as the background formalism. The specification of an equivalence morphism at metamodel level contributes to a more abstract and reusable solution for model management. Users of our model management approach do not have to deal with algebraic specifications directly. The use of EMF to algebraically define models dramatically reduces the learning curve for dealing with models from a formal generic standpoint. It also increases the interoperability with many industrial software development tools.

As we are implementing the MOMENT tool as an Eclipse plugin, AMW constitutes a good environment to define our traceability models by the user. Nevertheless, we are developing our own editor for traceability models in order to add the chance of invoking traceability operators directly from the editor interface. In this way, we will be able to automatically compose traceability models or to navigate them from visual interfaces, using the underlying algebraic specification formalism.

The next step in the MOMENT framework development process is to provide support for the QVT Relations language in order to use it for the definition of equivalence relationships and transformations. The work presented in this paper constitutes the traceability support that MOMENT will provide for the QVT standard.

References

1. Palmer, J.D.: Traceability. Richard H. Thayer and Merlin Dorfman, Software Requirements Engineering, 2nd Edition, IEEE Computer Society, pages 412–422, 2000.
2. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. Software Engineering, 27(1):58–93, 2001.

3. OMG Model-Driven Architecture. http://www.omg.org/mda/
4. Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00)).
5. Melnik, S., E. Rahm, P. A. Bernstein, "Rondo: A Programming Platform for Generic Model Management," Proc. SIGMOD 2003, pp. 193-204 (PDF, 344KB). Extended version in Web Semantics, Volume 1, Number 1.
6. Song, G., Zhang, K., Kong, J.: Model Management Through Graph Transformation. IEEE VL/HCC'04. Rome, Italy. 2004.
7. Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: Two experiencies in software dynamics. *Journal of Universal Science Computer*. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.
8. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. IEEE Computer Society Press. 9th European Conference on Software Maintenance and Reengineering. Manchester, UK. 2005.
9. M. Dorfman and R. Thayer. Guide to software requirements specification. IEEE Standards, Guidelines and Examples on System and Software Requirements Engineering, 1990.
10. Gotel, O. C. Z. and Finkelstein, A. C. W.: An Analysis of the Requirements Traceability Problem, Proceedings of the First International Conference on Requirements Engineering (ICRE '94), IEEE Computer Society Press, Colorado Springs, Colorado, U.S.A., April 18-22, pp. 94-101.
11. Requirements tools in the Volere web site: http://www.volere.co.uk/tools.htm
12. Object Management Group. Request for Proposal: MOF 2.0 Query / Views /Transformations RFP, 2002. ad/2002-04-10.
13. The EMF site: http://download.eclipse.org/tools/emf/scripts/home.php
14. Zhang, L., Yu, Y., Lu, J., Lin, C., Tu, K., Guo, M., Zhang, Z., Xie, G., Su, Z., Pan, Y.: ORIENT: Integrate Ontology Engineering into Industry Tooling Environment. In Proc. of the Third International Semantic Web Conference 2004, Hiroshima, Japan.
15. The Hyena web site: http://www.pst.ifi.lmu.de/~rauschma/hyena/
16. The Graphical Modeling Framework proposal: http://www.eclipse.org/proposals/eclipse-gmf/main.html
17. The JANE Model-specific editor generator web site: http://www.dstc.edu.au/Research/Projects/Pegamento/jane/
18. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.
19. Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F.: An M3-Neutral infrastructure for bridging model engineering and ontology engineering. In Proceedings of INTEROP-ESA'05, Geneve, Switzerland. 2005.
20. Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.
21. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.
22. OMG, "Meta Object Facility 1.4", http://www.omg.org/technology/documents/formal/mof.htm
23. Madhavan, J., P.A. Bernstein, and E. Rahm: Generic Schema Matching using Cupid. VLDB 2001.
24. Madhavan, J., Bernstein, P. A., Chen, K., Halevy, A.Y., Shenoy, P.: Corpus-based Schema Matching," Workshop on Information Integration on the Web, at IJCAI'2003, pp. 59-66.
25. Didonet Del Fabro, M, Bézivin, J, Jouault, F, Breton, E, and Gueltas, G : AMW: a generic model weaver. Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). 2005.

Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos[1]

Artur Boronat, José Iborra, José Á. Carsí, Isidro Ramos, Abel Gómez
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera, s/n
46022 València
{aboronat | jiborra | pcarsi | iramos | agomez}@dsic.upv.es

Resumen

Los métodos formales proporcionan buenas propiedades para abordar problemas en Ingeniería del Software. Sin embargo, en muchos casos no se suelen aplicar en un ámbito industrial debido a prejuicios o malas experiencias. En este artículo, se presenta un caso de éxito de la aplicación de especificaciones algebraicas en un entorno industrial de modelado para dar soporte a la Gestión de Modelos. Esta disciplina es una nueva tendencia dentro de la Ingeniería de Modelos que trata a los modelos como ciudadanos de primer orden y que proporciona una serie de operadores genéricos para manipularlos. Se ha especificado algebraicamente un conjunto de operadores de este tipo utilizando el lenguaje Maude. Estos operadores se utilizan de forma visual desde Eclipse Modeling Framework (EMF). En este artículo se presenta el soporte que se ofrece para la interoperabilidad entre Maude y EMF en una herramienta de gestión de modelos.

1. Introducción

En la Ingeniería del Software aplicada a la industria, la aplicación de formalismos para dar soporte a herramientas industriales no ha sido siempre bien recibida. Pese a que existen casos de exitosa aplicación [1], una serie de mitos y creencias separa a la comunidad teórica de la industria. En [2] se plantea un interesante debate sobre los puntos críticos de este cisma, basándose en una década de discusiones sobre este tema. En este artículo se presenta un caso exitoso de aplicación de especificaciones algebraicas a una herramienta industrial de modelado, en el contexto de la Gestión de Modelos.

En la Ingeniería de Modelos, cualquier artefacto software puede ser tratado o representado como un modelo [3]: ontologías, modelos UML, esquemas relacionales, esquemas XML, etc. Un modelo describe una realidad física, abstracta o hipotética, recogiendo únicamente la información necesaria que permite conseguir unos objetivos específicos: generación de código, integración de aplicaciones, interoperabilidad entre aplicaciones, etc. Trabajar directamente sobre modelos aumenta el nivel de abstracción de estas tareas y permite automatizarlas. Las herramientas que dan soporte a este tipo de tareas normalmente lo consiguen de una forma específica para un determinado contexto de trabajo (ontologías, bases de datos relacionales, etc) o para una determinada tecnología (integración de modelos UML con Rational Rose Integrator), generación de código a partir de herramientas de modelado (como Rational Rose, Visio), etc.

Dentro de la Ingeniería de Modelos, una nueva disciplina, llamada Gestión de Modelos [4], trata con modelos mediante una serie de operadores genéricos que permiten realizar tareas como las descritas anteriormente. Estos operadores constituyen una solución abstracta y reutilizable para trabajar sobre los modelos como ciudadanos de primer orden, independientemente del contexto o tecnología utilizada para representarlos.

[1] Este artículo ha sido financiado por el Proyecto Nacional de Investigación, Desarrollo e Innovación DYNAMICA TIC 2003-07804-C05-01.

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). 2005.

Dada nuestra experiencia en la aplicación del formalismo de especificaciones algebraicas a la recuperación de sistemas legados [5, 6], se está desarrollando una herramienta que da soporte algebraico a este tipo de operadores genéricos desde un entorno de modelado industrial. Nuestra aproximación demuestra la falacia de los mitos basados en la poca productividad de las herramientas formales y en el aumento del coste del proceso software debido a su uso. Para ello hemos elegido un eficiente sistema de reescritura de términos, Maude [7], que ya ha sido utilizado en muchos ámbitos formales [8].

Nuestra herramienta de gestión de modelos, llamada MOMENT (MOdel manageMENT), utiliza el entorno Maude desde un entorno de modelado industrial, como es Eclipse Modeling Framework (EMF) [9]. Esta integración de un método formal en una herramienta industrial de desarrollo de software, combina los esfuerzos que se están realizando sobre ambas herramientas en direcciones divergentes: en Maude sobre aspectos teóricos, y en EMF sobre su aplicación a la Ingeniería del Software. De esta manera, evitamos el aislamiento de Maude en el ámbito industrial, debido a la premisa "construyelo y ya vendrán"[2], frecuentemente mantenida por los desarrolladores de métodos formales. Este hecho permite utilizar Maude para solucionar problemas reales en la Ingeniería del Software, sin limitarse únicamente a la solución de los llamados ejemplos de juguete.

Este artículo presenta el mecanismo que se ha desarrollado en el marco de MOMENT para conseguir una buena interoperabilidad entre Maude y EMF. La estructura del artículo es la siguiente: la Sección 2 presenta una visión global de la aplicabilidad de la Gestión de Modelos presentando un ejemplo que se utilizará a lo largo del trabajo; en la Sección 3, se realiza una discusión sobre herramientas de modelado en las tendencias más relevantes de la Ingeniería de Modelos; en la Sección 4, se presenta el marco conceptual de nuestra solución para la interoperabilidad entre Maude y EMF; en la Sección 5, se presenta cómo se da soporte para la proyección de modelos EMF sobre Maude; finalmente, en la sección 6 se presentan una serie de conclusiones y trabajos futuros.

[2] Del inglés, "built-it-and-they-will-come".

2. Visión global del framework MOMENT

En MOMENT los operadores de gestión de modelos [10] han sido especificados algebraicamente utilizando el formalismo Maude. Los modelos se especifican como conjuntos de elementos de forma independiente del metamodelo, de manera que los operadores pueden acceder a los elementos sin conocer la representación de un modelo. La interfaz de MOMENT está integrada en EMF, de manera que el formalismo de especificaciones algebraicas queda totalmente transparente al usuario.

Para ilustrar el funcionamiento de MOMENT, se indica un pequeño ejemplo de integración de esquemas XML. Para ello se ha definido una parte del metamodelo del lenguaje de definición XML (XSD), mostrado en la Figura 1 utilizando notación UML.

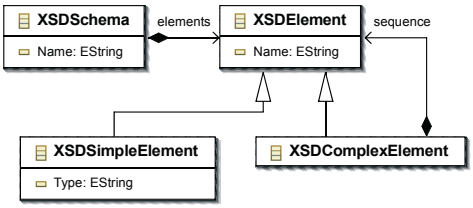


Figura 1. Parte del metamodelo XSD.

Utilizando el editor en forma de árbol que proporciona EMF, definimos los esquemas XML *A* y *B* en la Figura 2. Se aplica el operador *Merge* a ambos, obteniendo el esquema XML integrado *C* y dos modelos de trazas (map_{AC} y map_{BC}) que enlazan los elementos de los modelos de entrada con los elementos del modelo de salida. La invocación del operador es la siguiente: *<C, mapAC, mapBC> = Merge(A, B)*.

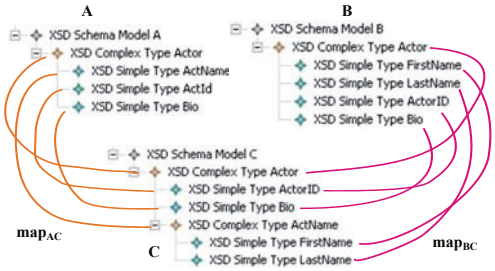


Figura 2. Aplicación del operador Merge.

Para poder realizar la integración de los esquemas XML, estos deben ser traducidos a

términos en Maude, para que el operador Merge pueda aplicarse sobre ellos. En los siguientes apartados se indica como se consigue esta interoperabilidad entre EMF y Maude mediante MOMENT.

3. Herramientas de modelado

En esta sección se indican algunas herramientas que dan soporte para la Ingeniería de Modelos, teniendo en cuenta las últimas tendencias. De este modo, se justifica el entorno de modelado elegido para nuestra aproximación de gestión de modelos.

Debido a la gran variedad de artefactos software, la interoperabilidad con herramientas de modelado de cualquier índole constituye una condición necesaria para que una herramienta de gestión de modelos tenga éxito. La iniciativa Model-Driven Architecture (MDA) [11] proporciona una serie de estándares que permiten compartir metainformación entre herramientas de modelado (XMI) y repositorios de metainformación (MOF).

Algunas herramientas que dan soporte a este enfoque son Eclipse Modeling Framework y NetBeans MetaData Repository. Eclipse Modeling Framework (EMF) es un entorno de modelado que proporciona facilidades para la generación de código. EMF puede ser utilizado como un marco de metamodelado donde el lenguaje común es un subconjunto de UML. NetBeans MetaData Repository da soporte tecnológico al estándar MOF constituyendo un repositorio de metainformación que puede ser modelada en UML a través de entornos visuales de modelado.

Una aproximación similar es seguida en la computación integrada en modelos (MIC – Model Integrated Computing) [12], con herramientas como Generic Modeling Environment y MetaEdit+. Todas estas herramientas constituyen un marco de metamodelado basadas en un lenguaje abstracto para definir la sintaxis, semántica y visualización de lenguajes específicos a dominio.

Siguiendo el enfoque de la Ingeniería Específica a Dominio [13], un nuevo paradigma de desarrollo de software está emergiendo con las llamadas Factorías de Software (Software Factories [14]). Esta aproximación se basa en la definición de lenguajes específicos a dominio para abordar el desarrollo de software en un contexto determinado, aumentando la calidad, la productividad y la automatización del proceso de desarrollo. La apuesta de Microsoft en este campo se enmarca en la herramienta Domain Specific Language tools, aún en fase preliminar. También existen propuestas de nuevas herramientas como Meta Programming System, y Graphical Modelling Framework (que funcionará sobre EMF).

Entre todas estas herramientas, hemos elegido EMF como entorno de modelado para nuestra herramienta de Gestión de Modelos por su situación dentro del marco de la Ingeniería de Modelos. EMF permite tratar con gran variedad de artefactos software, como esquemas XML, modelos UML (definidos en entornos visuales de modelado como Rational Rose), esquemas relacionales (a través de Rational Rose), y ontologías, entre otros. Además, EMF es utilizada por las principales herramientas de IBM, aportando una visión industrial a nuestro enfoque de Gestión de Modelos.

4. Espacios tecnológicos: EMF y Maude

El concepto de espacios tecnológicos fue introducido por Kurtev et al. en la discusión sobre el enlace de tecnologías heterogéneas [15]. Un espacio tecnológico (ET) es un contexto de trabajo en el que se dispone de un conjunto de conceptos bien definidos, una base de conocimiento, herramientas, y una serie de posibilidades de aplicación específicas [16]. Por ejemplo, en este artículo tratamos con los ETs de Eclipse Modeling Framework y de Maude.

En esta sección, se indica el marco conceptual que se ha utilizado para representar artefactos software mediante EMF y cómo estos son proyectados sobre Maude, con el objetivo de utilizar los operadores algebraicos de MOMENT sobre ellos.

4.1. Marco conceptual para la representación de artefactos software en Maude

Siguiendo un enfoque de Ingeniería de Modelos, para tratar con artefactos software utilizamos la terminología que define el estándar Meta-Object Facility de la iniciativa MDA. Este estándar presenta una arquitectura de cuatro capas de modelado que permite clasificar artefactos

software con diferente propósito: M3 (metametamodelos), M2 (metamodelo), M1 (modelo), M0 (sistema real).

Una estrategia para trabajar con metamodelos consiste en definir una sintaxis básica en el nivel M3, que pueda ser utilizada para definir artefactos software en niveles inferiores. En EMF, el metametamodelo se llama Ecore y proporciona una serie de primitivas de modelado: un subconjunto del diagrama de clases del metamodelo UML. Estas primitivas se utilizan para definir metamodelos en el nivel M2, constituyendo un paradigma de modelado. Como por ejemplo, el lenguaje de definición de esquemas XML (XML Schema Definition language - XSD).

Los elementos de un metamodelo son utilizados como tipos para definir los elementos que constituyen un modelo en el nivel M1. En el caso del metamodelo XSD, un modelo es un esquema XML específico. Los elementos de un modelo también se comportan como tipo para definir información en el nivel M0 de la arquitectura MOF. Por ejemplo, un esquema XML define los elementos que se pueden utilizar en un documento XML.

4.2. Proyecciones de artefactos software EMF sobre Maude

Un ET se caracteriza por el soporte tecnológico que se proporciona a un determinado paradigma de modelado. Cada paradigma de modelado se organiza entorno a un metametamodelo común y persigue unos objetivos específicos.

El ET EMF se caracteriza por las facilidades que ofrece para representar una buena variedad de artefactos sofware como modelos y por su interoperabilidad con otras herramientas industriales de modelado. El ET Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas: abstracción, subtipado, modularización, genericidad mediante parametrización, etc. Este ET también puede ser visto como un paradigma de modelado[3], considerando el lenguaje Maude como el lenguaje de definición de metamodelos en el nivel M3. En el nivel M2, los metamodelos son los módulos que proporcionan especificaciones algebraicas Maude.

Una especificación algebraica representa un metamodelo, proporcionando la descripción sintáctica de las primitivas (llamadas constructores en el campo de las especificaciones algebraicas), necesarias para especificar un artefacto software en el nivel M1. Cuando la especificación algebraica es interpretada como álgebra, se obtiene la visión como tipo del metamodelo, donde los constructores se pueden utilizar para definir artefactos software en el nivel M1. Éstos son representados sintácticamente como términos, representando la información en forma de árbol.

Para manipular modelos EMF con los operadores algebraicos de MOMENT se han definido una serie de proyecciones entre ambos ETs (ver Figura 3). Estas proyecciones permiten representar un modelo como un término algebraico, manipularlo desde Maude, y devolverlo como un modelo EMF.

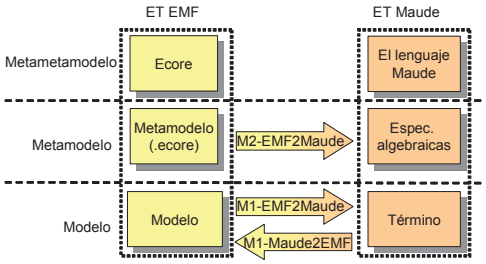


Figura 3. Enlaces entre el ET EMF y el ET Maude.

4.3. Interoperabilidad en el nivel M2

En el nivel de metamodelos se establece un enlace unidireccional que proyecta un metamodelo EMF sobre el ET Maude, obteniendo una especificación algebraica. De este modo, un metamodelo se interpreta como un álgebra que proporciona los constructores necesarios para definir modelos y

[3] Para simplificar el discurso, utilizamos una visión sintáctica de las especificaciones algebraicas y no tenemos en cuenta las capacidades de reflexión que Maude proporciona. Otra visión es la semántica, donde las capacidades de reflexión de Maude sitúan el Álgebra Universal como metametamodelo en el nivel M3.

las operaciones necesarias para manipularlos, en el contexto de la Gestión de Modelos.

Este enlace es unidireccional pues los metamodelos se especifican mediante herramientas visuales de modelado a través de EMF (el nombre que hemos asignado a este enlace es M2-EMF2Maude). Este hecho permite hacer transparente el uso del formalismo Maude al usuario final del framework MOMENT.

MOMENT trabaja directamente sobre un álgebra de operadores genéricos de manipulación de modelos, especificados en el módulo MOMENT-OP(X::TRIV). Estos operadores pueden ser adaptados a un metamodelo específico haciendo uso de las capacidades de parametrización que ofrece Maude, basándose en el concepto formal de Pushout [17] de teoría de categorías.

En el diagrama del mecanismo de paso de parámetros[4] (Figura 4), TRIV constituye el parámetro formal del módulo parametrizado MOMENT-OP(X::TRIV). La especificación sigXSD constituye el parámetro real para el módulo parametrizado. sigXSD proporciona los constructores correspondientes a las primitivas de un metamodelo específico. Esta especificación algebraica está relacionada con el parámetro formal mediante la vista vXSD. Como ejemplo se ha utilizado el metamodelo XSD simplificado. En él, a partir de la metainformación[5] que describe la clase XSDSimpleElement, que indica cómo definir un elemento simple en un esquema XML, se obtiene un constructor que permite definir un elemento simple en un esquema XML como un término del álgebra.

El módulo MOMENT-OP(vXSD) representa la instaciación del módulo parametrizado con el parámetro real sigXSD. Esta especificación valor es importada por el módulo spXSD, en el que se extiende la presentación axiomática de los operadores genéricos adaptándola a un metamodelo específico. Por ejemplo, el operador *Merge*, cuando se utiliza en el metamodelo XSD, permite integrar esquemas XML. Para definir este tipo de integración de forma más precisa, se pueden añadir relaciones de equivalencia que tengan en cuenta las primitivas del metamodelo XSD: elemento simple, elemento complejo, etc. Estas relaciones de equivalencia específicas al metamodelo XSD se añaden al módulo spXSD en forma de axiomas. La especificación algebraica resultante constituye el álgebra de un metamodelo. El módulo spXSD también es generado automáticamente por la herramienta MOMENT a partir de la especificación de los nuevos axiomas en interfaces visuales[6].

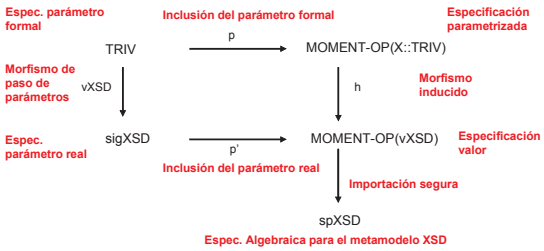


Figura 4. Diagrama del mecanismo de paso de parámetros en Maude.

4.4. Interoperabilidad en el nivel M1

Existe otro tipo de enlace entre el ET EMF y el ET Maude en el nivel de modelos. Este enlace es bidireccional y consta de dos tipos de proyecciones:

- M1-EMF2Maude: Este mecanismo proyecta un modelo EMF, definido mediante un metamodelo EMF, sobre el ET Maude como un término. Para proyectar un modelo sobre el ET Maude, la herramienta MOMENT consulta el correspondiente metamodelo y obtiene el constructor de la correspondiente especificación algebraica, que es necesario

[4] En la figura, p y p' son morfismos que indican inclusión de especificaciones algebraicas, y h es el morfismo inducido a partir de la vista vXSD, que obtiene los elementos de MOMENT-OP(vXSD) a partir de los elementos del módulo parametrizado.

[5] Expresada como una instancia de la clase EClass del metametamodelo Ecore.

[6] El objetivo del artículo se centra en el soporte que MOMENT ofrece para la proyección de artefactos software de EMF a Maude. Como la generación del módulo spXSD concierne más a la semántica de las operaciones de manipulación de modelos, nos centramos en la obtención del parámetro real para el álgebra genérica de operadores (en el caso del metamodelo XSD, es el módulo sigXSD).

para especificar el término de forma automática.

- M1-Maude2EMF: Este mecanismo proporciona la proyección inversa a la anterior, obteniendo un modelo EMF a partir de un término Maude. En este paso, cuando la herramienta MOMENT lee un término que representa un modelo, determina las primitivas del metamodelo EMF que debe utilizar para construir el modelo EMF a partir de los símbolos de los constructores utilizados en el término. Con estas primitivas se construye el modelo dinámicamente y se persiste en formato XMI.

Los enlaces, que han sido descritos entre el ET EMF y el ET Maude, permiten la aplicación de operadores algebraicos sobre modelos definidos de forma gráfica mediante entornos industriales de modelado. Por ejemplo, supongamos que se desea realizar la integración de dos esquemas XML, cuyo metamodelo ha sido definido mediante Ecore. El proceso seguido es el siguiente:

1. Se obtiene la especificación algebraica spXSD a partir del metamodelo XSD.
2. Se proyectan los esquemas XML A y B a términos del álgebra, interpretada a partir de spXSD.
3. Se aplica el operador Merge a ambos términos y se obtiene el término resultante mediante el mecanismo de reducción de Maude. Como resultado de la operación de integración se obtiene el término C[7], que representa el esquema XML integrado.
4. Finalmente, el término C es proyectado al ET EMF como un modelo.

5. Soporte tecnológico para la interoperabilidad entre EMF y Maude

En esta sección, se detalla la implementación del soporte para la interoperabilidad entre el ET Maude y el ET EMF, que ofrece MOMENT. En primer lugar, se indica cómo se ha integrado Maude en la plataforma Eclipse. En segundo lugar, se detalla la implementación de los enlaces entre ETs presentados conceptualmente en la sección anterior.

5.1. Maude en Eclipse

Para hacer uso de las características de Maude, se ha desarrollado dos plug-ins que nos permiten su uso directamente desde el entorno de Eclipse.

El primero consta de todas las funcionalidades para configurar, ejecutar, e interactuar con Maude desde Java. Este plug-in incluye una ventana de configuración integrada en las preferencias de Eclipse para indicar dónde se encuentran los ficheros necesarios, así como otras opciones. El plug-in, a su vez, proporciona dos APIs para invocar a Maude desde el código de una aplicación: de forma interactiva o por lotes.

El segundo plug-in se trata de un editor con coloreado de sintaxis que facilita la escritura de programas y su ejecución directamente en una consola de Maude. Proporciona los correspondientes asistentes para la creación de ficheros nuevos, así como una barra de herramientas para interactuar con el proceso.

5.2. Soporte para la interoperabilidad en el nivel de metamodelos

El enlace del nivel M2 obtiene la especificación algebraica correspondiente a un metamodelo especificado en Ecore. En concreto, para cada tipo del metamodelo se incluye un sort, y para cada tipo no abstracto[8] se incluye en la especificación algebraica un constructor.

La obtención de la especificación algebraica de un metamodelo se ha automatizado mediante técnicas generativas a través de patrones. Con este objetivo, se ha utilizado el motor de plantillas Velocity [18]. Un motor de plantillas toma como entrada una plantilla y un contexto para la plantilla, que proporciona información para generar código, como la resolución de referencias.

En el contexto de la proyección de metamodelos a Maude se ha definido una plantilla que hace posible la generación automática de la

[7] El operador Merge también produce dos modelos de trazabilidad que relacionan los modelos de entrada A y B con el modelo de salida C, respectivamente. Ambos modelos se han obviado para simplificar el ejemplo.

[8] En el contexto de MOF y EMF, el concepto de tipo abstracto incluye las interfaces y las clases abstractas.

especificación algebraica correspondiente. Para realizar la proyección de un metamodelo en el espacio tecnológico de Maude se siguen los siguientes pasos:

1. Previamente a la generación, el metamodelo es analizado y procesado para producir una estructura más cercana a la representación en la plantilla destino. El procesado incluye: la clasificación de los tipos en abstractos y concretos, la definición de la jerarquía de sorts, y la obtención de la signatura de los constructores.
2. El motor de plantillas recibe el resultado de este procesado y la plantilla a utilizar, y produce la correspondiente especificación algebraica del metamodelo.

En la Figura 5, se muestra una captura de pantalla en la que aparece la especificación generada a partir del metamodelo XSD del ejemplo, en la interfaz del editor Maude.

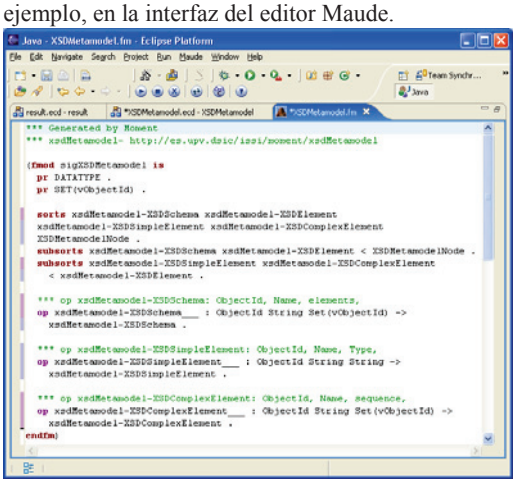


Figura 5. Especificación sigXSD generada a partir del metamodelo XSD simplificado.

5.3. Soporte para la interoperabilidad en el nivel M1: M1-EMF2Maude

La proyección de un modelo en el espacio tecnológico de Maude equivale a un conjunto de términos de la especificación algebraica correspondiente al metamodelo utilizado. Para este mecanismo se ha utilizado la misma técnica generativa que en el caso anterior. Por lo tanto el proceso seguido es similar, y en este caso las responsabilidades del preprocesado son las siguientes:

- Sustitución de tipos de datos de EMF por tipos de datos de Maude.
- Identificación de todos los metamodelos que intervienen en la definición del modelo
- Generación de identificadores únicos. Cada término generado a partir de un elemento de un modelo EMF tiene asignado un identificador, que es utilizado para dar soporte a las referencias en Maude.
- Obtención del constructor de la especificación algebraica, que corresponde al tipo de un elemento del modelo.

5.4. Soporte para la interoperabilidad en el nivel M1: M1-Maude2EMF

El problema de la obtención de modelos EMF a partir de términos algebraicos se ha abordado como un problema de análisis de lenguaje, teniendo en cuenta la sintaxis Maude para especificar términos.

Para generar los analizadores léxico, sintáctico y semántico requeridos se ha empleado el generador de parsers ANTLR [19], cuyo funcionamiento es similar a los conocidos Bison y Flex. La mayor parte de la lógica presente en el proceso de análisis se concentra en el analizador semántico, encargado de recorrer el Arbol Abstracto (Abstract Syntax Tree, AST), que está formado por representantes de los términos algebraicos procedentes del código fuente, y de ir construyendo el modelo en EMF a medida que se analice. Para cada término el proceso es el siguiente:

1. A partir del símbolo del constructor del término se identifica el metamodelo que contiene al tipo que representa a este término, y se carga la clase del elemento representado por el término. El analizador semántico crea una instancia dinámicamente, que ya es la proyección en EMF del término algebraico.
2. Seguidamente se rellenan los atributos de la instancia creada con los valores de los subtérminos encontrados en el término. Las referencias se dejan pendientes.
3. Una vez se han procesado todos los términos se resuelven las referencias entre ellos, de forma similar al proceso seguido con los atributos.

6. Conclusiones

En este artículo se ha presentado una visión global de la herramienta MOMENT, y el marco conceptual en el que se está trabajando dentro de la Ingeniería de Modelos. Se ha presentado el soporte para la interoperabilidad entre los espacios tecnológicos EMF y Maude, que permite aplicar operadores definidos algebraicamente a modelos definidos mediante herramientas visuales de modelado. Además, este hecho permite combinar los esfuerzos que se están realizando actualmente sobre cada una de estas herramientas, en el campo industrial de EMF y en el campo teórico de Maude.

Como trabajo futuro, se está desarrollando el soporte para definir las relaciones existentes entre modelos de forma visual. El objetivo consiste en proporcionar una herramienta que permita definir operadores de gestión de modelos algebraicamente y que permita su aplicación desde una herramienta visual de modelado.

Agradecimientos

A Francisco Durán, por facilitarnos la documentación y la implementación de Maude Workstation, en el que nos hemos basado para ejecutar Maude sobre Windows.

Referencias

[1] Bowen, J.P., Hinchey, M.G. Seven More Myths of Formal Methods. IEEE Software. July 1995 (Vol. 12, No. 4).

[2] Gogolla, M. Benefits and Problems of Formal Methods. *Proc. 9th Int. Conf. Reliable Software Technologies Ada-Europe (RST'2004)*, pages 1-15. Springer, Berlin, LNCS 3063, 2004.

[3] Bézivin, J.: On the Unification Power of Models, in: "Software and System Modeling (SoSym)", 2004, vol. 3, no 4.

[4] Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. MRS Tech. Rep. MSR-TR-2000-53, (short version in SIGMOD Record 29, 4 (Dec. '00)).

[5] Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: Two experiencies in software dynamics. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.

[6] Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. IEEE Computer Society Press. CSMR 2005. Manchester, UK. 2005.

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.

[8] Martí-Oliet N., Meseguer, J. Rewriting logic: Roadmap and bibliography. Theoretical Computer Science, 285(2):121–154, 2002.

[9] El sitio web de EMF: http://download.eclipse.org/tools/emf/scripts/

[10] Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.

[11] OMG Model-Driven Architecture. http://www.omg.org/mda/

[12] Sztipanovits J. y Karsai G., Model-Integrated Computing, Computer, Apr. 1997, pp. 90-92.

[13] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications.Addisson-Wesley (2000). ISBN 0-201-30977-7, pag. 267-304.

[14] Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley. 2004. ISBN: 0-471-20284-3

[15] Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.

[16] Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F.: An M3-Neutral infrastructure for bridging model engineering and ontology engineering. INTEROP-ESA. Switzerland. 2005.

[17] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Springer-Verlag Berlin Heidelberg New York Tokio (1985). ISBN: 3-540-13718-1.

[18] El motor de plantillas Apache Velocity: http://jakarta.apache.org/velocity

[19] El generador de parsers ANTLR: http://www.antlr.org

Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework[1]

Artur Boronat, José Iborra, José Á. Carsí, Isidro Ramos, Abel Gómez
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera, s/n
46022 València
{aboronat | jiborra | pcarsi | iramos | agomez}@dsic.upv.es

Resumen

Los métodos formales proporcionan buenas propiedades para abordar problemas en Ingeniería del Software. Sin embargo, en muchos casos no se suelen aplicar en un ámbito industrial debido a prejuicios o malas experiencias. En este artículo, se presenta un caso de éxito de la aplicación de especificaciones algebraicas en un entorno industrial de modelado para dar soporte a la Gestión de Modelos. Esta disciplina es una nueva tendencia dentro de la Ingeniería de Modelos que trata a los modelos como ciudadanos de primer orden y que proporciona una serie de operadores genéricos para manipularlos. Se ha especificado algebraicamente un conjunto de operadores de este tipo utilizando el lenguaje Maude. Estos operadores se utilizan de forma visual desde Eclipse Modeling Framework (EMF). En este artículo se presenta el soporte que se ofrece para la interoperabilidad entre Maude y EMF en una herramienta de gestión de modelos.

1. Introducción

En la Ingeniería del Software aplicada a la industria, la aplicación de formalismos para dar soporte a herramientas industriales no ha sido siempre bien recibida. Pese a que existen casos de exitosa aplicación [1], una serie de mitos y creencias separa a la comunidad teórica de la industria. En [2] se plantea un interesante debate sobre los puntos críticos de este cisma, basándose en una década de discusiones sobre este tema. En este artículo se presenta un caso exitoso de aplicación de especificaciones algebraicas a una herramienta industrial de modelado, en el contexto de la Gestión de Modelos.

En la Ingeniería de Modelos, cualquier artefacto software puede ser tratado o representado como un modelo [3]: ontologías, modelos UML, esquemas relacionales, esquemas XML, etc. Un modelo describe una realidad física, abstracta o hipotética, recogiendo únicamente la información necesaria que permite conseguir unos objetivos específicos: generación de código, integración de aplicaciones, interoperabilidad entre aplicaciones, etc. Trabajar directamente sobre modelos aumenta el nivel de abstracción de estas tareas y permite automatizarlas. Las herramientas que dan soporte a este tipo de tareas normalmente lo consiguen de una forma específica para un determinado contexto de trabajo (ontologías, bases de datos relacionales, etc) o para una determinada tecnología (integración de modelos UML con Rational Rose Integrator), generación de código a partir de herramientas de modelado (como Rational Rose, Visio), etc.

Dentro de la Ingeniería de Modelos, una nueva disciplina, llamada Gestión de Modelos [4], trata con modelos mediante una serie de operadores genéricos que permiten realizar tareas como las descritas anteriormente. Estos operadores constituyen una solución abstracta y reutilizable para trabajar sobre los modelos como ciudadanos de primer orden, independientemente del contexto o tecnología utilizada para representarlos.

[1] Este artículo ha sido financiado por el Proyecto Nacional de Investigación, Desarrollo e Innovación DYNAMICA TIC 2003-07804-C05-01.

Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos, JISBD'2005, pp.253-258
ISBN: 84-9732-434-X **© 2005 Los autores, Thomson**

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos - JISBD'05. 2005.

Dada nuestra experiencia en la aplicación del formalismo de especificaciones algebraicas a la recuperación de sistemas legados [5, 6], se está desarrollando una herramienta que da soporte algebraico a este tipo de operadores genéricos desde un entorno de modelado industrial. Nuestra aproximación demuestra la falacia de los mitos basados en la poca productividad de las herramientas formales y en el aumento del coste del proceso software debido a su uso. Para ello hemos elegido un eficiente sistema de reescritura de términos, Maude [7], que ya ha sido utilizado en muchos ámbitos formales [8].

Nuestra herramienta de gestión de modelos, llamada MOMENT (MOdel manageMENT), utiliza el entorno Maude desde un entorno de modelado industrial, como es Eclipse Modeling Framework (EMF) [9]. Esta integración de un método formal en una herramienta industrial de desarrollo de software, combina los esfuerzos que se están realizando sobre ambas herramientas en direcciones divergentes: en Maude sobre aspectos teóricos, y en EMF sobre su aplicación a la Ingeniería del Software. De esta manera, evitamos el aislamiento de Maude en el ámbito industrial, debido a la premisa "constrúyelo y ya vendrán"[2], frecuentemente mantenida por los desarrolladores de métodos formales. Este hecho permite utilizar Maude para solucionar problemas reales en la Ingeniería del Software, sin limitarse únicamente a la solución de los llamados ejemplos de juguete.

Este artículo presenta el mecanismo que se ha desarrollado en el marco de MOMENT para conseguir una buena interoperabilidad entre Maude y EMF. La estructura del artículo es la siguiente: la Sección 2 presenta una visión global de la aplicabilidad de la Gestión de Modelos; en la Sección 3, se realiza una discusión sobre herramientas de modelado en las tendencias más relevantes de la Ingeniería de Modelos; en la Sección 4, se presenta el marco conceptual de nuestra solución para la interoperabilidad entre Maude y EMF; finalmente, en la sección 6 se presentan una serie de conclusiones y trabajos futuros.

[2] Del inglés, "built-it-and-they-will-come".

2. Visión global del framework MOMENT

El concepto de espacios tecnológicos fue introducido por Kurtev et al. en la discusión sobre el enlace de tecnologías heterogéneas [15]. Un espacio tecnológico (ET) es un contexto de trabajo en el que se dispone de un conjunto de conceptos bien definidos, una base de conocimiento, herramientas, y una serie de posibilidades de aplicación específicas [16]. Por ejemplo, en MOMENT tratamos con los ETs de Eclipse Modeling Framework y de Maude.

En MOMENT, los operadores de gestión de modelos [10] han sido especificados algebraicamente utilizando el formalismo Maude. Los modelos se especifican como conjuntos de elementos de forma independiente del metamodelo, de manera que los operadores pueden acceder a los elementos sin conocer la representación de un modelo. La interfaz de MOMENT está integrada en EMF, de manera que el formalismo de especificaciones algebraicas queda totalmente transparente al usuario. Los operadores algebraicos permiten manipular modelos definidos en EMF a partir de un determinado metamodelo.

En MOMENT, se va a utilizar una arquitectura de cuatro capas en el nivel de modelado basada en el estándar Meta-Object Facility [18], realizando proyecciones entre los dos espacios, lo cual permitirá la interoperabilidad entre EMF y Maude a través de los siguientes pasos:

- Proyección de metamodelos (nivel M2) a especificaciones algebraicas.
- Proyección de modelos (nivel M1) a términos de alguna de las especificaciones algebraicas obtenidas.
- Aplicación de operadores de gestión de modelos sobre los términos mediante el mecanismo de reducción de Maude.
- Proyección de los términos resultantes a modelos EMF.

3. Herramientas de modelado

En esta sección se indican algunas herramientas que dan soporte para la Ingeniería de Modelos, teniendo en cuenta las últimas tendencias. De este

modo, se justifica el entorno de modelado elegido para nuestra aproximación de gestión de modelos.

Debido a la gran variedad de artefactos software, la interoperabilidad con herramientas de modelado de cualquier índole constituye una condición necesaria para que una herramienta de gestión de modelos tenga éxito. La iniciativa Model-Driven Architecture (MDA) [11] proporciona una serie de estándares que permiten compartir metainformación entre herramientas de modelado (XMI) y repositorios de metainformación (MOF).

Algunas herramientas que dan soporte a este enfoque son Eclipse Modeling Framework y NetBeans MetaData Repository. Eclipse Modeling Framework (EMF) es un entorno de modelado que proporciona facilidades para la generación de código. EMF puede ser utilizado como un marco de metamodelado donde el lenguaje común es un subconjunto de UML. NetBeans MetaData Repository da soporte tecnológico al estándar MOF constituyendo un repositorio de metainformación que puede ser modelada en UML a través de entornos visuales de modelado.

Una aproximación similar es seguida en la computación integrada en modelos (MIC – Model Integrated Computing) [12], con herramientas como Generic Modeling Environment y MetaEdit+. Todas estas herramientas constituyen un marco de metamodelado basadas en un lenguaje abstracto para definir la sintaxis, semántica y visualización de lenguajes específicos a dominio.

Siguiendo el enfoque de la Ingeniería Específica a Dominio [13], un nuevo paradigma de desarrollo de software está emergiendo con las llamadas Factorías de Software (Software Factories [14]). Esta aproximación se basa en la definición de lenguajes específicos a dominio para abordar el desarrollo de software en un contexto determinado, aumentando la calidad, la productividad y la automatización del proceso de desarrollo. La apuesta de Microsoft en este campo se enmarca en la herramienta Domain Specific Language tools, aún en fase preliminar. También existen propuestas de nuevas herramientas como Meta Programming System, y Graphical Modelling Framework (que funcionará sobre EMF).

Entre todas estas herramientas, hemos elegido EMF como entorno de modelado para nuestra herramienta de Gestión de Modelos por su situación dentro del marco de la Ingeniería de Modelos. EMF permite tratar con gran variedad de artefactos software, como esquemas XML, modelos UML (definidos en entornos visuales de modelado como Rational Rose), esquemas relacionales (a través de Rational Rose), y ontologías, entre otros. Además, EMF es utilizada por las principales herramientas de IBM, aportando una visión industrial a nuestro enfoque de Gestión de Modelos.

4. Espacios tecnológicos: EMF y Maude

En esta sección, se indica el marco conceptual que se ha utilizado para representar artefactos software mediante EMF y cómo estos son proyectados sobre Maude, con el objetivo de utilizar los operadores algebraicos de MOMENT sobre ellos.

4.1. Marco conceptual para la representación de artefactos software en Maude

Siguiendo un enfoque de Ingeniería de Modelos, para tratar con artefactos software utilizamos la terminología que define el estándar Meta-Object Facility de la iniciativa MDA. Este estándar presenta una arquitectura de cuatro capas de modelado que permite clasificar artefactos software con diferente propósito: M3 (metametamodelos), M2 (metamodelo), M1 (modelo), M0 (sistema real).

Una estrategia para trabajar con metamodelos consiste en definir una sintaxis básica en el nivel M3, que pueda ser utilizada para definir artefactos software en niveles inferiores. En EMF, el metametamodelo se llama Ecore y proporciona una serie de primitivas de modelado: un subconjunto del diagrama de clases del metamodelo UML. Estas primitivas se utilizan para definir metamodelos en el nivel M2, constituyendo un paradigma de modelado. Como por ejemplo, el lenguaje de definición de esquemas XML (XML Schema Definition language - XSD).

Los elementos de un metamodelo son utilizados como tipos para definir los elementos que constituyen un modelo en el nivel M1. En el caso del metamodelo XSD, un modelo es un esquema XML específico. Los elementos de un modelo también se comportan como tipo para

definir información en el nivel M0 de la arquitectura MOF. Por ejemplo, un esquema XML define los elementos que se pueden utilizar en un documento XML.

4.2. Proyecciones de artefactos software EMF sobre Maude

Un ET se caracteriza por el soporte tecnológico que se proporciona a un determinado paradigma de modelado. Cada paradigma de modelado se organiza entorno a un metametamodelo común y persigue unos objetivos específicos.

El ET EMF se caracteriza por las facilidades que ofrece para representar una buena variedad de artefactos sofware como modelos y por su interoperabilidad con otras herramientas industriales de modelado. El ET Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas: abstracción, subtipado, modularización, genericidad mediante parametrización, etc. Este ET también puede ser visto como un paradigma de modelado[3], considerando el lenguaje Maude como el lenguaje de definición de metamodelos en el nivel M3. En el nivel M2, los metamodelos son los módulos que proporcionan especificaciones algebraicas Maude.

Una especificación algebraica representa un metamodelo, proporcionando la descripción sintáctica de las primitivas (llamadas constructores en el campo de las especificaciones algebraicas), necesarias para especificar un artefacto software en el nivel M1. Cuando la especificación algebraica es interpretada como álgebra, se obtiene la visión como tipo del metamodelo, donde los constructores se pueden utilizan para definir artefactos software en el nivel M1. Éstos son representados sintácticamente como términos, representando la información en forma de árbol.

Para manipular modelos EMF con los operadores algebraicos de MOMENT se han definido una serie de proyecciones entre ambos ETs (ver Figura 1). Estas proyecciones permiten representar un modelo como un término algebraico, manipularlo desde Maude, y devolverlo como un modelo EMF.

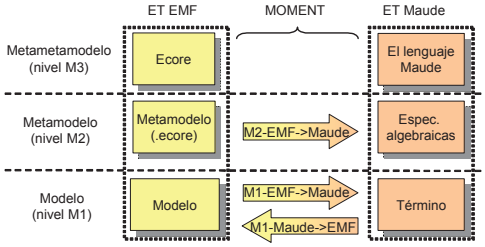


Figura 1. Enlaces entre el ET EMF y el ET Maude.

4.3. Interoperabilidad en el nivel M2

En el nivel de metamodelos se establece un enlace unidireccional que proyecta un metamodelo EMF sobre el ET Maude, obteniendo una especificación algebraica. De este modo, un metamodelo se interpreta como un álgebra que proporciona los constructores necesarios para definir modelos y las operaciones necesarias para manipularlos, en el contexto de la Gestión de Modelos.

Este enlace es unidireccional pues los metamodelos se especifican mediante herramientas visuales de modelado a través de EMF (el nombre que hemos asignado a este enlace es M2-EMF->Maude). Este enlace permite hacer transparente el uso del formalismo Maude al usuario final del framework MOMENT. Este es el motivo por el que el enlace es unidireccional en este nivel, ya que los metamodelos serán diseñados desde herramientas industriales de modelado, y no directamente en el entorno formal Maude.

MOMENT trabaja directamente sobre un álgebra de operadores genéricos de manipulación de modelos, especificados en el módulo MOMENT-OP(X::TRIV). Estos operadores pueden ser adaptados a un metamodelo específico haciendo uso de las capacidades de parametrización que ofrece Maude, basándose en el concepto formal de Pushout [17] de teoría de categorías.

[3] Para simplificar el discurso, utilizamos una visión sintáctica de las especificaciones algebraicas y no tenemos en cuenta las capacidades de reflexión que Maude proporciona. Otra visión es la semántica, donde las capacidades de reflexión de Maude sitúan el Álgebra Universal como metametamodelo en el nivel M3.

En el diagrama del mecanismo de paso de parámetros[4] (Figura 2), TRIV constituye el parámetro formal del módulo parametrizado MOMENT-OP(X::TRIV). La especificación sigXSD constituye el parámetro actual para el módulo parametrizado. sigXSD proporciona los constructores correspondientes a las primitivas de un metamodelo específico. Esta especificación algebraica está relacionada con el parámetro formal mediante la vista vXSD.

A partir de un metamodelo XSD se genera el parámetro sigXSD. El módulo MOMENT-OP(vXSD) representa la instanciación del módulo parametrizado con el parámetro real sigXSD. Esta especificación valor es importada por el módulo spXSD, en el que se extiende la presentación axiomática de los operadores genéricos adaptándola a un metamodelo específico. Por ejemplo, el operador *Merge*, cuando se utiliza en el metamodelo XSD, permite integrar esquemas XML.

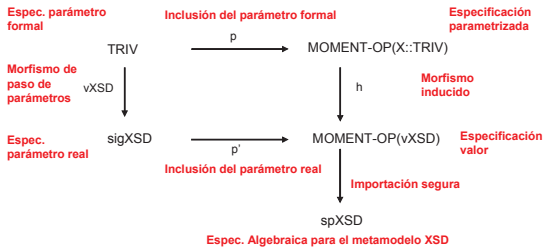


Figura 2. Diagrama del mecanismo de paso de parámetros en Maude.

4.4. Interoperabilidad en el nivel M1

Existe otro tipo de enlace entre el ET EMF y el ET Maude en el nivel de modelos. Este enlace es bidireccional y consta de dos tipos de proyecciones:

- M1-EMF->Maude: Este mecanismo proyecta un modelo EMF, definido mediante un metamodelo EMF, sobre el ET Maude como un término. Para proyectar un modelo sobre el ET Maude, la herramienta MOMENT consulta el correspondiente metamodelo y obtiene el constructor de la correspondiente especificación algebraica, que es necesario para especificar el término de forma automática.
- M1-Maude->EMF: Este mecanismo proporciona la proyección inversa a la anterior, obteniendo un modelo EMF a partir de un término Maude. En este paso, cuando la herramienta MOMENT lee un término que representa un modelo, determina las primitivas del metamodelo EMF que debe utilizar para construir el modelo EMF a partir de los símbolos de los constructores utilizados en el término. Con estas primitivas se construye el modelo dinámicamente y se persiste en formato XMI.

Estos enlaces permiten aplicar operadores algebraicos a modelos EMF. Por ejemplo, para integrar dos esquemas XML, cuyo metamodelo ha sido definido mediante Ecore, se realiza el siguiente proceso automáticamente (ilustrado en la Figura 3):

1. Se obtiene la especificación algebraica spXSD a partir del metamodelo XSD.
2. Se proyectan los esquemas XML A y B a términos del álgebra, interpretada a partir de spXSD.
3. Se aplica el operador Merge a ambos términos y se obtiene el término resultante mediante el mecanismo de reducción de Maude. Como resultado de la operación de integración se obtiene el término C, que representa el esquema XML integrado.
4. Finalmente, el término C es proyectado al ET EMF como un modelo.

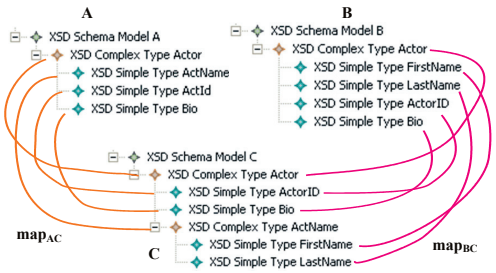


Figura 3. Aplicación del operador Merge. (<C, map_{AC}, map_{BC}> = Merge(A, B))

[4] En la figura, p y p' son morfismos que indican inclusión de especificaciones algebraicas, y h es el morfismo inducido a partir de la vista vXSD, que obtiene los elementos de MOMENT-OP(vXSD) a partir de los elementos del módulo parametrizado.

5. Conclusiones

En este artículo se ha presentado una visión global de la herramienta MOMENT, y el marco conceptual en el que se está trabajando dentro de la Ingeniería de Modelos. Se ha presentado el soporte conceptual para la interoperabilidad entre los espacios tecnológicos EMF y Maude, que permite aplicar operadores definidos algebraicamente a modelos definidos mediante herramientas visuales de modelado. Además, este hecho permite combinar los esfuerzos que se están realizando actualmente sobre cada una de estas herramientas, en el campo industrial de EMF y en el campo teórico de Maude.

Como trabajo futuro, se está desarrollando el soporte para definir las relaciones existentes entre modelos de forma visual. El objetivo consiste en proporcionar una herramienta que permita definir operadores de gestión de modelos algebraicamente y que permita su aplicación desde una herramienta visual de modelado.

Agradecimientos

A Francisco Durán, por facilitarnos la documentación y la implementación de Maude Workstation, en el que nos hemos basado para ejecutar Maude sobre Windows.

Referencias

[1] Bowen, J.P., Hinchey, M.G. Seven More Myths of Formal Methods. IEEE Software. July 1995 (Vol. 12, No. 4).

[2] Gogolla, M. Benefits and Problems of Formal Methods. *Proc. 9th Int. Conf. Reliable Software Technologies Ada-Europe (RST'2004)*, pages 1-15. Springer, Berlin, LNCS 3063, 2004.

[3] Bézivin, J.: On the Unification Power of Models, in: "Software and System Modeling (SoSym)", 2004, vol. 3, no 4.

[4] Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. MRS Tech. Rep. MSR-TR-2000-53, (short version in SIGMOD Record 29, 4 (Dec. '00)).

[5] Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: Two experiencies in software dynamics. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.

[6] Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. IEEE Computer Society Press. CSMR 2005. Manchester, UK. 2005.

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.

[8] Martí-Oliet N., Meseguer, J. Rewriting logic: Roadmap and bibliography. Theoretical Computer Science, 285(2):121–154, 2002.

[9] El sitio web de EMF: http://download.eclipse.org/tools/emf/scripts/

[10] Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.

[11] OMG Model-Driven Architecture. http://www.omg.org/mda/

[12] Sztipanovits J. y Karsai G., Model-Integrated Computing, Computer, Apr. 1997, pp. 90-92.

[13] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications.Addisson-Wesley (2000). ISBN 0-201-30977-7, pag. 267-304.

[14] Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley. 2004. ISBN: 0-471-20284-3

[15] Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.

[16] Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F.: An M3-Neutral infrastructure for bridging model engineering and ontology engineering. INTEROP-ESA. Switzerland. 2005.

[17] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Springer-Verlag Berlin Heidelberg New York Tokio (1985). ISBN: 3-540-13718-1.

[18] OMG, “Meta Object Facility 1.4”, http://www.omg.org/technology/documents/formal/mof.htm

UN MOTOR DE TRANSFORMACIÓN DE MODELOS CON SOPORTE PARA EL LENGUAJE QVT RELATIONS

Pascual Queralt, Luis Hoyos, Artur Boronat, José Á. Carsí e Isidro Ramos

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
C/Camino de Vera s/n
46022 Valencia - España
e-mail: {pqueralt,lhoyos,aboronat,pcarsi,iramos}@dsic.upv.es, web: http://issi.dsic.upv.es

Palabras clave: QVT Relations, Transformaciones de modelos, Ingeniería de Modelos, Maude.

Resumen. *En la actualidad están apareciendo un gran número de herramientas que proporcionan soporte para transformar modelos. No obstante, hasta el momento no existe ninguna que haya tenido en consideración el lenguaje estándar QVT Relations. En este artículo se presenta la arquitectura de un motor de transformación de modelos que proporciona soporte para este lenguaje estándar dentro del framework de gestión de modelos MOMENT. Este motor de transformaciones ha sido definido mediante la utilización de métodos formales como plugin dentro de Eclipse, haciendo uso de Eclipse Modeling Framework para la manipulación de modelos.*

1. INTRODUCCIÓN

El paradigma de desarrollo dirigido por modelos es una disciplina de la Ingeniería del Software donde los modelos son considerados como artefactos de primera clase, y donde la transformación genérica de modelos realiza un papel fundamental. Han sido numerosos los lenguajes de transformación surgidos con el propósito de especificar transformaciones genéricas entre modelos, pero la dificultad de aprendizaje y la escasa productividad de las herramientas que soportan estos lenguajes ha limitado su repercusión dentro de la comunidad del software. En un intento de aunar los esfuerzos que se están llevando a cabo en este campo, la OMG ha propuesto un lenguaje estándar para la especificación de transformaciones llamada *Query View Transformations* (QVT) [1], que depende a su vez de otros dos estándares: MOF 2.0 y OCL 2.0. De esta manera, se pretende promover el uso estándar de las transformaciones entre modelos, lo que llevará asociado notables mejoras en

Queralt, P., Hoyos, L., Boronat, A., Carsí, J. A., Ramos, I. Un motor de transformación de modelos con soporte para el lenguaje QVT Relations. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). 2006.

la productividad, la interoperabilidad y la calidad, permitiendo que el desarrollo de software se lleve a cabo en un nivel de abstracción más elevado y utilizando conceptos más cercanos al dominio del problema.

Para que la visión del desarrollo dirigido por modelos sea una realidad, las herramientas que lo soportan deben ser capaces de proporcionar la automatización de transformaciones de modelos. Éstas, no sólo deben ofrecer transformaciones de modelos de forma *ad-hoc*, sino también proporcionar un lenguaje que permita a los usuarios definir sus propias transformaciones y ejecutarlas posteriormente, con lo que se podrán automatizar tareas como la aplicación de patrones, el refinamiento o la refactorización.

En este trabajo presentamos un prototipo que ofrece soporte para el lenguaje QVT Relations. El motor de transformaciones está integrado en la herramienta de gestión de modelos MOMENT (*MOdel manageMENT*) basada en una aproximación híbrida entre una herramienta formal y un entrono industrial de modelado, Maude y *Eclipse Modelling Framework* (EMF) [2]. Por un lado, Maude es un sistema de reescritura de términos que proporciona soporte para lógica ecuacional de pertenencia y lógica de reescritura. Maude se ha utilizado como entorno de prototipado rápido para desarrollar la herramienta haciendo uso de algunas de sus propiedades: mecanismo de *pattern matching*, parametrización y reflexión entre otros. Por otro lado, EMF es un estándar industrial que incluye un metamodelo (*ecore*[1]) para describir y soportar la ejecución de modelos, incluyendo notificación de cambios, soporte de persistencia mediante serialización XMI y una API reflexiva muy eficiente para la manipulación de modelos EMF. Con todo esto, MOMENT reaprovecha los esfuerzos realizados tanto en el campo teórico como en el aplicado: en Maude sobre aspectos teóricos y en EMF sobre su aplicación a la Ingeniería del Software.

La estructura del documento es la siguiente: en el apartado 2 se presenta la especificación QVT y se describen brevemente las principales características de MOMENT-QVT, seguidamente en el apartado 3 se muestran los distintos componentes funcionales de MOMENT a través de la secuencia del proceso en la ejecución de transformaciones; en el apartado 4 se ven algunos de los trabajos relacionados en el área y finalmente en el apartado 5 se detallan las conclusiones.

2. QVT Y MOMENT

La especificación QVT se define a través de dos dimensiones ortogonales: la dimensión del lenguaje y la dimensión de la interoperabilidad, donde cada una de las cuales tiene una serie de niveles. La dimensión del lenguaje define los diferentes lenguajes de transformación presentes en la especificación QVT. Concretamente son tres: Relations, Core y Operational, y la principal diferencia entre ellos es su naturaleza declarativa o imperativa. En la dimensión de la interoperabilidad se encuentran aquellas características que permiten a una herramienta que cumple el estándar QVT interoperar con otras herramientas. La intersección de niveles de las dos dimensiones especifica un punto de cumplimiento QVT (*QVT-compliance*).

Para integrar QVT en MOMENT se ha elegido el lenguaje Relations para aprovechar su

[1] Ecore es un lenguaje común basado en EMOF, que es parte de la especificación MOF 2.0 usado para la definición de metamodelos

naturaleza declarativa, su especificación de trazabilidad transparente al usuario, así como la sintaxis bastante sencilla, en comparación con otros lenguajes declarativos, que ofrece para definir transformaciones y relaciones de equivalencia entre modelos. En el lenguaje Relations las transformaciones quedan descritas a través de la enumeración de los metamodelos participantes, un conjunto de reglas que especifican la relación existente entre términos de los metamodelos, un conjunto de dominios por regla que se ajusta al conjunto de términos para los que se expresan relaciones y un conjunto de patrones que cumplen con la estructura de los términos y a los que se aplican operaciones OCL. El que una regla sea de transformación o no viene determinado por el dominio destino, que estará marcado como *checkonly* o *enforce*. La marca *checkonly* comprueba si existe una correspondencia válida entre los modelos, mientras que la marca *enforce* fuerza a que los términos de los respectivos modelos cumplan la relación descrita en la regla.

Respecto a la interoperabilidad en MOMENT, se permite la ejecución de modelos que conforman al metamodelo QVT Relations, la ejecución de cualquier especificación textual de una transformación expresada con el lenguaje Relations, así como la importación y exportación de modelos serializados con formato XMI (véase Tabla 1).

En MOMENT, a través de QVT Relations, es posible la especificación de transformaciones endógenas y exógenas. Las transformaciones endógenas son aquellas transformaciones que comparten el mismo metamodelo, mientras que las transformaciones exógenas se definen entre modelos que conforman a metamodelos diferentes[2] [3].

		Interoperabilidad			
Lenguaje		Sintaxis Ejecutable	XMI Ejecutable	Sintaxis Exportable	XMI Exportable
	Core				
	Relations	**X**	**X**		**X**
	Operational				

Tabla 1. Cumplimiento de QVT**.**

Por un lado, el hecho de que MOMENT se base en estándares como MOF, QVT, OCL y XMI, conlleva implícitamente un alto nivel de interoperabilidad con herramientas similares. Por otro, la utilización y aplicación de Maude como método formal subyacente permitirá el estudio formal de ciertas propiedades de las transformaciones de modelos: como la terminación o el indeterminismo.

En cuanto a la cardinalidad de modelos participantes en una transformación, el número máximo de modelos origen no está limitado. Por otra parte, el número de modelos destino queda restringido a uno tal y como se especifica en el estándar QVT, aunque es posible

[2] Si en una transformación se utilizan varios modelos origen y/o varios modelos destino, entonces pueden estar involucrados varios metamodelos diferentes.

alcanzar cardinalidad múltiple en el resultado aplicando composición de transformaciones que tengan diferentes modelos resultantes y que pasen a formar parte del resultado final. La forma con la que actualmente se lleva a cabo la composición de transformaciones es manual y mediante interacción del usuario, pero en un futuro próximo se tiene previsto que este proceso se lleve a cabo de manera automática. Dado que en MOMETN-QVT las transformaciones se proyectan como funciones en Maude, la composición de transformaciones se basa en la composición algebraica de funciones. En definitiva, son posibles las transformaciones 1-1, n-1, 1-n y n-n, bien aplicando una sola transformación o mediante composición de transformaciones.

3. EJECUCIÓN QVT RELATIONS EN MOMENT

El proceso de transformación que se lleva a cabo en MOMENT-QVT consta de tres fases: análisis, proyección a Maude y ejecución y proyección a EMF (Figura 1). Todas ellas quedan ocultas al usuario, siendo únicamente necesaria la especificación de la transformación por parte del mismo.

Para llevar a cabo estas tres fases, son necesarios un conjunto de componentes funcionales que forman la arquitectura de MOMENT-QVT.

- QVT Parser: encargado de analizar un programa QVT Relations de entrada y generar su modelo correspondiente.
- MOMENT Registry: repositorio de artefactos generados y/o utilizados por MOMENT.
- MOMENT Relations: encargado de generar y proyectar código Maude a partir de la especificación de una transformación.
- OCL Parser: encargado de analizar expresiones OCL y generar su código Maude correspondiente.

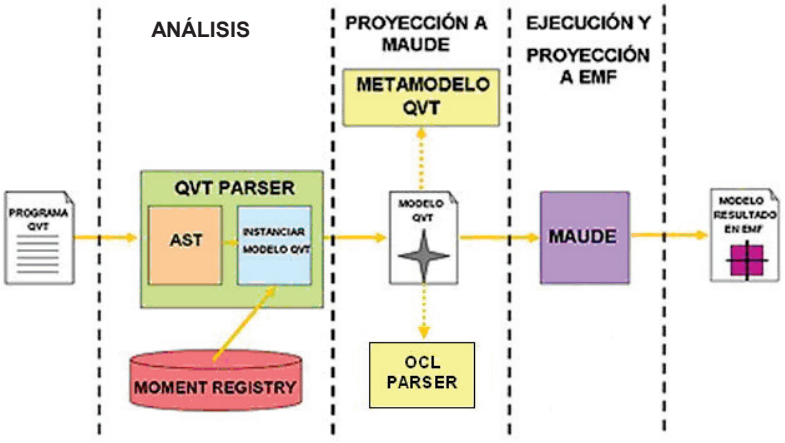


Figura 1. Proceso de ejecución de un programa QVT Relations en MOMENT

3.1. Fase de análisis

Esta etapa comienza con un análisis léxico y sintáctico del programa QVT Relations.

Recorriendo el árbol de sintaxis abstracta (AST) generado por esta etapa, se crea un modelo instancia del metamodelo de QVT Relations que se encuentra integrado en la plataforma MOMENT. Durante este recorrido es necesario consultar los metamodelos respectivos a los modelos participantes en la transformación y obtener de este modo los tipos y las propiedades de sus elementos. Por consiguiente, es prerrequisito imprescindible que estos metamodelos estén registrados en el MOMENT *Registry*.

Para el ejemplo, nos hemos basado en la transformación que aparece en el apéndice de la especificación del estándar MOF-QVT, que especifica una transformación de un diagrama de clases UML a su correspondiente modelo relacional. En la parte izquierda de la Figura 2, se muestra el segmento de código de la regla de transformación "ClassToTable". Esta regla establece la correspondencia entre una clase perteneciente al metamodelo origen (UML) y una tabla perteneciente al metamodelo destino (relacional), de forma que a partir del término clase perteneciente a uno de los dominios de la regla se obtienen los términos tabla, columna y clave primaria, que pertenecen al dominio destino de la regla tal como indica la palabra reservada *enforce*. En la parte derecha de la Figura 2 y tras llevar a cabo la fase de análisis, se muestra la parte del modelo QVT Relations correspondiente a la regla "ClassToTable".

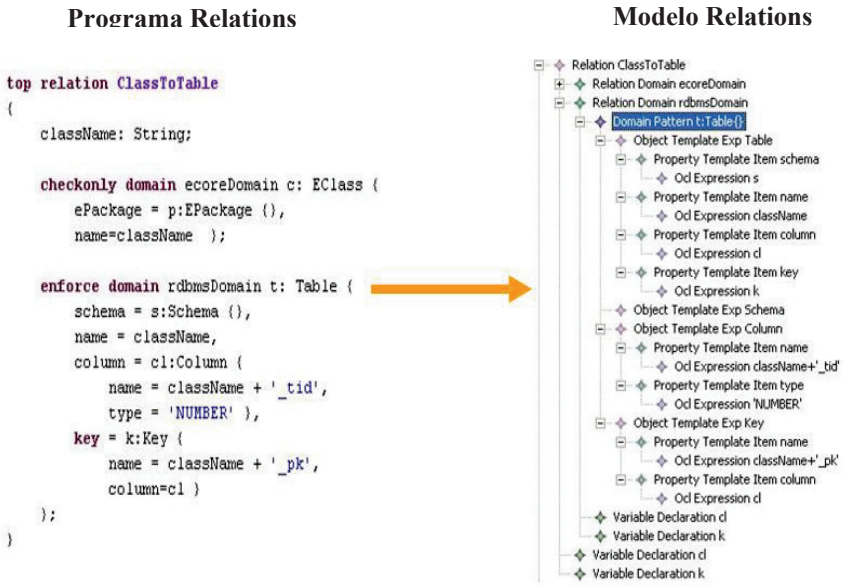


Figura 2. Vistas de programa Relations y modelo asociado.

3.2. Fase de proyección a Maude

Los metamodelos origen y destino son compilados en teorías basadas en lógica ecuacional [4]. Los conceptos del metamodelo se proyectan como *sorts*[3] y las propiedades como

[3] Un sort es el nombre que recibe un tipo den Maude.

constructores[4] y operadores de consulta. Así, se crea una especificación algebraica en el espacio tecnológico de Maude para cada metamodelo participante en la transformación. Además, se toman todos los modelos origen (en la capa M1 de MOF) y se obtiene para cada uno de ellos el término algebraico que conforma la especificación algebraica definida para su respectivo metamodelo [5].
Seguidamente, el modelo QVT Relations resultado de la etapa de análisis se proyecta en otra especificación algebraica. Una parte muy importante de esta proyección es la traducción de las expresiones OCL utilizadas en la especificación de la transformación, y que se realiza a través del componente OCL Parser [6].
La nueva especificación algebraica obtenida extiende a las especificaciones ya existentes de los metamodelos participantes en la transformación, mediante un conjunto de reglas de reescritura que describen las guías para realizar la transformación, constituyendo el contexto donde se definen las relaciones semánticas entre los metamodelos origen y destino.
La Figura 3 muestra el resultado de proyectar a Maude la regla de transformación "ClassToTable" obtenida en la fase anterior. En ella se aprecia la ecuación "TransformElements", que lleva a cabo su ejecución a través de los mecanismos de *pattern-matching* y de recursión. La generación de los términos resultantes "Table", "Column", "Key" se realiza a través del operador "New", el cálculo de sus respectivos identificadores se consigue con el operador "AddOID" y la asignación de valores y contenido a cada una de las propiedades de estos términos se obtiene por medio del operador asignación "←" y las navegaciones del modelo origen partiendo de su respectivo dominio en la regla como es el caso de "ecoreEClass0 :: name".

```
*** regla 1: ClassToTable
   eq TransformElements( ClassToTable , ? Set{ecoreEClass0} ? ecoreModel0 ,  TargetModel, Tuple2 ) =
          Set(
          AddOID(
             ( (New("Table", MM((empty-set).Set{rdbms}))).rdbmsNode
                 :: schema <-- ((p1
                         ((ModelGenRule ( PackageToSchema ,
                             ? ( Set{((ecoreEClass0 :: ePackage ( ecoreModel0 ))} -> flatten ) ? ecoreModel0 ,
                              TargetModel , Tuple2)
                         )) ))
                 :: name <-- (((ecoreEClass0 :: name)))
                 :: column <-- (Set ( AddOID(
                                 (New("Column", MM((empty-set).Set{rdbms}))).rdbmsNode
                                 :: name <-- (((((ecoreEClass0 :: name) + "_tid"))))
                                 :: type <-- ("NUMBER") )))
                 :: key <-- (Set ( AddOID(
                             (New("Key", MM((empty-set).Set{rdbms}))).rdbmsNode
                             :: name <-- (((((ecoreEClass0 :: name) + "_pk"))))
                             :: column <-- (Set ( AddOID(
                                         (New("Column", MM((empty-set).Set{rdbms}))).rdbmsNode
                                         :: name <-- (((((ecoreEClass0 :: name) + "_tid"))))
                                         :: type <-- ("NUMBER") )))
                          )))
             )))
   .
```

Figura 3. Proyección a Maude.

[4] Un constructor en Maude es un operador que permite definir información. En nuestro caso tendríamos un constructor para definir la instancia de una clase.

3.3. Ejecución y proyección a EMF

Llegado este punto, se dispone en el espacio tecnológico Maude de toda la información necesaria para lanzar la transformación a ejecución. Desde Maude, se aprovecha el mecanismo de reducción modulo las ecuaciones generadas a partir de un programa QVT Relations, como brevemente se indica en el apartado anterior y de *pattern-matching*, obteniendo el término resultante que en el ejemplo anterior se corresponde con un esquema relacional. Finalmente, este término es proyectado de vuelta a EMF utilizando los puentes definidos a nivel M1 entre ambos espacios tecnológicos. Se pueden encontrar más detalles del proceso de compilación a código Maude en [5].

4. TRABAJOS RELACIONADOS

Diversas son las herramientas y lenguajes que están apareciendo en el área de transformaciones de modelos. Estas aproximaciones generalmente se diferencian según la naturaleza del lenguaje (declarativo o imperativo) y la representación de los modelos que utilice. Como representación de modelos existen dos tendencias principales: grafos y conjuntos. A continuación detallamos alguna de las características de las herramientas más extendidas.

4.1. XSLT

XSLT es un lenguaje estándar de la *World Wide Web Consortium* (W3C) y cuenta con un extenso soporte tecnológico [7]. A pesar de satisfacer la gran mayoría de los requisitos necesarios por un lenguaje de transformación usado en *Model Driven Engineering* (MDE), resulta bastante complicado escribir transformaciones XSLT porque el usuario ha de tener en cuenta los esquemas XSD de entrada [8].

Como principales características, podemos decir que XSLT tiene una naturaleza híbrida pues permite tanto construcciones imperativas como declarativas. En su parte declarativa, la aplicación del *pattern-matching* se hace de forma ordenada y recursiva, por lo que XSLT es determinista. Además, desde la perspectiva de metamodelado, efectúa la transformación de grafos, que son árboles, como representación de modelos, y utiliza XPath como lenguaje de consulta.

Algunas de las restricciones existentes son: no tener disponibles transformaciones bidireccionales, ausencia de soporte de trazabilidad implícita, carencia de mecanismos automáticos de composición y no tener definida su sintaxis como instancia de MOF. Asimismo, XSLT no tiene una equivalencia directa con QVT y es difícil establecer una correspondencia entre ambos.

4.2. ATL

ATL forma parte del *framework* de gestión de modelos AMMA [9] que se encuentra integrado en Eclipse y EMF. Utiliza un lenguaje propietario llamado ATLAS para definir transformaciones que se ejecuta sobre JAVA y que proporciona un entorno de depuración. La naturaleza de ATLAS es declarativa, aunque lleva mucho tiempo utilizando también

construcciones imperativas, y las transformaciones son expresadas como reglas de transformación ya que ATL cumple el estándar MOF.
ATL posee un algoritmo de ejecución preciso y determinista, sin embargo no se apoya sobre ningún método formal. No proporciona mecanismos para la validación de las transformaciones. Para llevar a cabo transformaciones compuestas se necesita ejecutar cada una de las transformaciones participantes una a una.
Aunque la sintaxis de ATL es muy similar a la de QVT, ésta no es interoperable con QVT. No obstante, es posible definir transformaciones entre ambos espacios tecnológicos tal como se explica en [10].

4.3. VIATRA

Viatra actualmente forma parte del framework VIATRA2 [11], que ha sido escrito en Java y que se encuentra integrada en Eclipse. Viatra provee un lenguaje textual para describir modelos y metamodelos, y transformaciones llamados VTML y VTCL respectivamente.
La naturaleza del lenguaje es declarativa y está basada en técnicas de descripción de patrones, sin embargo es posible utilizar secciones de código imperativo. Se apoya en métodos formales como la transformación de grafos (GT) y la máquina de estados abstractos (ASM) para ser capaz de manipular modelos y realizar tareas de verificación, validación, seguridad, así como una temprana evaluación de características no funcionales como fiabilidad, disponibilidad y productividad del sistema bajo diseño.
Como puntos débiles podemos resaltar que Viatra no se basa en los estándares MOF y QVT. No obstante, pretende soportarlos en un futuro mediante mecanismos de importación y exportación integrados en el *framework*.

4.4. EPSILON

Epsilon es una plataforma desarrollada como un conjunto de *plug-ins* (editores, asistentes, pantallas de configuración, etc) sobre Eclipse. Presenta el lenguaje metamodelo independiente *Epsilon Object Language* que se basa en OCL [12]. Puede ser utilizado como lenguaje de gestión de modelos o como infraestructura a extender con nuevos lenguajes específicos de dominio. Tres son los lenguajes definidos en la actualidad: *Epsilon Comparison Language* (ECL), *Epsilon Merging Language* (EML), *Epsilon Transformation Language* (ETL), para comparación, composición y transformación de modelos respectivamente. Se da soporte completo al estándar MOF mediante modelos EMF y documentos XML a través de JDOM.
La finalidad perseguida con la extensión del lenguaje OCL es: dar soporte al acceso de múltiple modelos, ofrecer constructores de programación convencional adicionales (agrupación y secuencia de sentencias), permitir modificación de modelos, proveer depuración e informe de errores, así como conseguir una mayor uniformidad en la invocación. Soporta mecanismos de herencia, trazabilidad e introduce mecanismos de comprobación automática del resultado en composición y transformación de modelos [13].

5. CONCLUSIÓN

Las transformaciones de modelos juegan un papel importante en la iniciativa MDA [14]. Dadas estas necesidades, OMG ha propuesto el estándar QVT que proporciona diferentes aproximaciones para lenguajes de transformación de modelos. Nuestra aproximación constituye un primer prototipo ejecutable de transformaciones especificadas a través del lenguaje QVT Relations, proporcionando soporte para el *pattern matching* de reglas de transformación, trazabilidad y navegación mediante expresiones OCL. De este modo, consideramos que MOMENT-QVT es una buena alternativa en el campo de las transformaciones, ya que al estar basada en estándares aprovecha la reutilización de tecnología y se consigue reducir la curva de aprendizaje de la herramienta.

De acuerdo con nuestra experiencia trabajando con transformaciones en MOMENT-QVT, el lenguaje Relations es capaz de expresar transformaciones tanto sencillas como complejas; eso sí, es imprescindible conocer el lenguaje OCL con el propósito de añadir más potencia y funcionalidad a la propia transformación en su aplicación tanto a modelos origen como destino. Diversos ejemplos de transformaciones nos han permitido conocer el rendimiento actual del prototipo, concluyendo que la primera ejecución que tiene lugar en MOMENT-QVT tiene asociado un coste temporal cuatro veces superior al resto con un coste aceptable e inmediato, debido a la carga de módulos del *kernel*[5].

En cuanto al soporte actual, MOMENT-QVT únicamente soporta transformaciones unidireccionales. Sin embargo, es posible soportar bidireccionalidad en transformaciones de carácter endógeno o en aquellas de carácter exógeno, siempre que no haya pérdida de información, siendo además necesario que las manipulaciones que se realicen mediante expresiones OCL sean reversibles; por ejemplo la concatenación de *strings* no es reversible. También se está trabajando en el soporte de ejecución de transformaciones QVT Relations en modo *checkonly*, que se encargan de comprobar que las relaciones establecidas por los modelos participantes en la transformación se sostienen en todas direcciones. Así pues, en un breve espacio de tiempo, MOMENT-QVT será uno de los primeros prototipos ejecutables que cumplan completamente el lenguaje Relations de la especificación QVT. Como trabajos futuros pretendemos implementar mecanismos que permitan la herencia de transformaciones, así como comprobar y preservar la semántica de los modelos resultantes de las transformaciones.

REFERENCIAS

[1] MOF QVT Standard Specification. http://www.omg.org/docs/ptc/05-11-01.pdf
[2] Sitio web de EMF: http://www.eclipse.org/emf/
[3] Tom Mens, Pieter Van Gorp : *A Taxonomy of Model Transforamtion.* Proc. Int'l Workshop on Graph and Model Transformation (GraMoT 2005)

[5] Núcleo de MOMENT que contiene todos los módulos necesarios para ejecución.

[4] Artur Boronat, José Iborra, José A. Carsí, Isidro Ramos, Abel Gómez: *Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos*. Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). September 2005. Granada, Spain.

[5] Artur Boronat, José Á. Carsí, Isidro Ramos : *Algebraic Specification of a Model Transformation Engine*. LNCS. Fundamental Approaches to Software Engineering (FASE'06). ETAPS'06. Vienna (Austria). March 27-29. 2006.

[6] Artur Boronat, Joaquin Oriente, Abel Gómez, Isidro Ramos, José A. Carsí : *An Algebraic Specification for Generic OCL Queries within the Eclipse Modeling Framework*. Proceedings of Second European Conference on Model Driven Architecture. Bilbao (Spain). Springer LNCS. July 10th-13th 2006 (to appear)

[7] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel and Andrew Wood: *Transformation: The Missing Link of MDA*. First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002

[8] Boronat A.,Carsí J.A., Ramos I.: *An Algebraic Baseline for Automatic Transformations in MDA*. Software Evolution Through Transformations: Model-based vs. Implementation-level Solutions Workshop (SETra'04), Second International Conference on Graph Transformation (ICGT2004), Electronic Notes in Theoretical Computer Scienc.

[9] Bézivin, J., Valduriez. P., Jouault, F. *The ATL home page*. http://www.sciences.univ-nantes.fr/lina/atl

[10] Frédéric Jouault, Ivan Kurtev: *On the ArchitecturalAlignment of ATL and QVT*. Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, chapter Model transformation (MT 2006)

[11] Sitio web VIATRA: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/VIATRA2/index.html

[12] Dimitiros S. Kolovos, Richard F. Paige, and Fiona A.C Polack: *The Epsilon Object Language (EOL)*. Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 2006.

[13] Dimitiros S. Kolovos, Richard F. Paige, and Fiona A.C Polack: *Model Comparison: A Foundation for Model Composition and Model Transformation Testing*. First International Workshop on Global Integrated Model Management (G@MMA) 2006, co-located with ICSE'06, Shanghai, China, May 2006.

[14] Shane Sendall and Wojtek Kozaczynski. *Model Transformation – the Heart and Soul of Model-Driven Software Development*. IEEE Software, Special Issue on Model Driven Software Development, pages 42--45, Sept/Oct 2003.

ECEASST

Exogenous Model Merging by means of Model Management Operators

Artur Boronat[1], José Á. Carsí [2] and Isidro Ramos[3]

[1] aboronat@dsic.upv.es, [2] pcarsi@dsic.upv.es, [3] iramos@dsic.upv.es
http://issi.dsic.upv.es
Information Systems and Computation Department
Technical University of Valencia, Spain

Abstract: In Model-Driven Engineering, model merging plays a relevant role in the maintenance and evolution of model-based software. Depending on the amount of metamodels involved in a model merging process, we can classify model merging techniques in two categories: endogenous merging, when all the models to be merged conform to the same metamodel; and exogenous merging, when the models to be merged conform to different metamodels. MOMENT (MOdel manageMENT) is a framework that is integrated in the Eclipse platform, and provides a collection of generic set-oriented operators to manipulate MOF models, following the Model Management discipline. In this paper, we study how model transformations are useful in a model merging process and we provide a solution for both kinds of model merging by means of model management operators and the QVT Relations language.

Keywords: Model-Driven Architecture, Model Management, Exogenous Model Merging, QVT Relations

1 Introduction

Software merging is an essential aspect of the maintenance and evolution of large-scale information systems. Information systems can be specified by means of models in Model-Driven Engineering. Models collect the information that describes the information system at a high level of abstraction, which permits the development of the application in an automated way using generative programming techniques. The consolidation of the Meta-Object Facility standard [OMG04] as a four-layer architecture, where metamodels can be specified as a set of syntactical well-formedness rules to define models, permits the definition of modeling domains where merging processes can be performed. A model merging process can be defined over a metamodel. Then, any two well-formed models in this metamodel can be merged. Traditionally, the tasks that are involved in this process have usually been solved in an ad-hoc manner for a specific context or metamodel: relational databases [BLN86, BDK92], XML schemas [Beh00], OWL-DL ontologies [HM05] , aspect-oriented modeling [SGS+04], UML models [OWK03], etc.

[Men02] presents a classification of merge approaches, where domain independence and customizability of a generic merge operator to a specific domain are desired features. However,

Boronat, A., Carsí, J. A., Ramos, I. Exogenous Model Merging by means of Model Management Operators. Proc. of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change at ICGT 2006. 2006.

the definition of metamodels by means of a common metamodeling language (like MOF, or any MOF-like implementation) is a desired feature that should be preserved on the grounds that it permits the development of generic infrastructures to manipulate models.

Following this direction, Model Management [BHP00] is a new emergent discipline that pursues an abstract reusable solution for problems of this kind, independently of the metamodel under study. The Model Management discipline deals with software artifacts by means of generic operators that do not depend on their internal implementation because they work on mappings between models [Ber03]. These operators treat models as first-class citizens and increase the level of abstraction of the solution avoiding programming tasks and improving the reusability of the solution.

As stated in [BLN86], a model merging process consists of three main phases: a model comparison phase, where elements of different models that are equivalent are found; a consistency checking phase, where conflicts that may appear if we merge equivalent elements are identified, defining a conflict resolution strategy to eliminate them; and a merging phase, where the equivalent elements that are found in the first step are merged taking into account the conflict strategy defined in the second step.

Generic model merging approaches provide support for these three phases in different ways. [AP03] uses MOF identifiers to compare elements in different versions of a same base model. [BP03, BCE+06] provide a set of model management operators to define equivalence relationships between elements of different models by means mappings, which are used by a merge operator later on. [KPP06] proposes several domain-specific languages to define model comparison and model merging over metamodels. The model comparison language permits the definition of equivalence relationships between elements of a metamodel that can be applied over elements of the corresponding models afterwards. The model merging language embeds the comparison language so that these equivalence relationships can be used in the merging process.

In our approach, we propose a set of model management operators that use the QVT Relations language [OMG05] to perform model comparison and model transformation. In a model merging process where two models are involved, the comparison phase is achieved by defining relations between elements of the same metamodel. The consistency phase is solved by defining a model transformation that takes the two models to be merged as input models. Finally, the merging phase is performed by a generic operator that uses the QVT Relations programs defined in the previous phases. Thus, we enhance the use of the QVT Relations language within the Model Management field, avoiding the definition of a new DSL for every model management operator. In this paper, we show how this approach can be used by providing an example of exogenous model merging, where the models to be merged conform[1] to different metamodels.

The structure of the paper is as follows: Section 2 presents the exogenous model merging problem; Section 3 introduces the *ModelGen* operator for model transformation; Section 4 introduces the *Merge* operator for model merging; Section 5 provides the solution for the example in Section 2; Section 6 provides some related works. Finally, Section 7 summarizes the main contributions of the paper.

[1] A model conforms to a metamodel if it is syntactically well-formed by using the constructs of the metamodel.

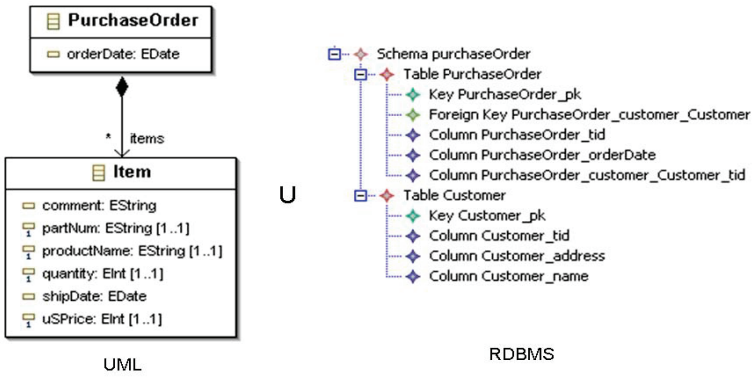


Figure 1: Exogenous model merging of a UML model and a relational schema.

2 Exogenous Model Merging Scenario

When two models are merged, an equivalence relation must be defined between their corresponding metamodels, associating their elements using a set of relationships. These relationships are used to identify equivalent elements in different models in order to avoid duplicated information in the merged model.

Generic approaches to merge models use this concept of equivalence relation, but they do not usually differentiate between an endogenous and an exogenous model merging. In Fig. 1, we provide an example of exogenous model merging: the integration of a UML model and a relational schema. We have used the Ecore metamodel [BBM03] as an implementation for the UML class diagram metamodel, and the relational metamodel that appears in the Query/View/Transformation (QVT) standard specification [OMG05]. In Fig. 1, the relational schema is shown in a tree-like form.

In this paper, we use this example to show that an exogenous model merging process is a generalization of an endogenous model merging process. Therefore, it can be broken down into simpler processes, which can be solved by means of model management operators. Our approach for solving the example consists of two steps: a model transformation that permits representing the UML model as a relational schema; and a model merging between relational schemas. We present how we deal with model transformation and endogenous model merging in the following sections.

3 The QVT Relations Language and the *ModelGen* Operator

In the QVT Relations language, a model transformation is defined among several metamodels, which are called the domains of the transformation. A QVT transformation is constituted by QVT relations, which become declarative transformation rules. A QVT relation specifies a relationship that must hold between the model elements of different candidate models. The direction of the transformation is defined when it is invoked by choosing a specific domain as target. If

the target domain is defined in the QVT transformation as *enforce*, a transformation is performed by creating the corresponding elements in the target model. If the target domain is defined as *checkonly*, just a checking is performed without creating any new element in the target model. Both kinds of transformations are used in our approach.

A relation can be also constrained by two sets of predicates, a *when* clause and a *where* clause. The *when* clause specifies the conditions under which the relationship needs to hold. The *where* clause specifies the condition that must be satisfied by all model elements participating in the relation.

A transformation contains two kinds of relations: top-level (marked with the *top* keyword) and non-top-level. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the where clause of another relation.

As example, we have taken the *UmlToRdbms* transformation that is presented in the MOF QVT final specification[2]. The top relation below specifies the transformation of a *Class* into a *Table*. By means of the *where* clause, the relation *ClassToTable* needs to hold only when the *PackageToSchema* relation holds between the package containing the class and the schema containing the table. By means of the *when* clause, the *ClassToTable* relation holds, the relation *AttributeToColumn* must also hold.

```
top relation ClassToTable {
    className: String;
    checkonly domain ecoreDomain c: EClass {
        ePackage = p:EPackage {},
        name=className
    };
    enforce domain rdbmsDomain t: Table {
        schema = s:Schema {},
        name = className,
        column = cl:Column {
            name = className + '_tid',
            type = 'NUMBER'
        },
        key = k:Key {
            name = className + '_pk',
            column=cl
        }
    };
    when {
        PackageToSchema(p, s);
    }
    where {
        AttributeToColumn(c, t, className);
    }
}
```

In MOMENT, a model transformation can be applied to several source models, which may or may not conform to the same metamodel. When the transformation is invoked, it generates one target model and a set of traceability models. A traceability model contains a set of traces

[2] In this paper, we are using a version of this transformation in which we consider Ecore as an implementation of the UML Class Diagram metamodel. The version of the transformation that is used is presented in Appendix B.

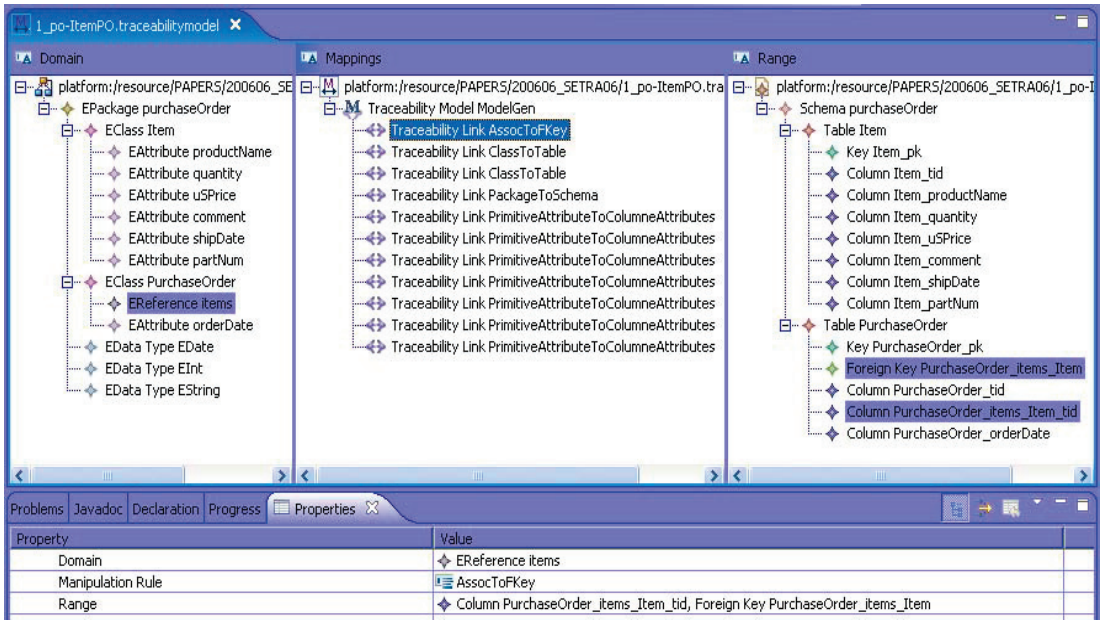


Figure 2: Traceeability Editor in the MOMENT Framework.

that relate the elements of the source model to the elements of the target model, indicating which transformation rule has been applied to each source element. A QVT Relations enforced transformation is executed by means of the *ModelGen* operator as follows:

$$< output\_model,\ trac_1, ...,\ trac_n >= ModelGen(transformation,\ input\_model_1, ...,\ input\_model_n)$$

where $transformation$ is the name of the QVT transformation; $input\_model_1, ..., input\_model_n$ are the input models, which may conform to different metamodels; $output\_model$ is the generated model; and $trac_1, ..., trac_n$ are the trace models that are generated for each one of the corresponding input models.

Fig. 2 presents the traceability editor of the MOMENT framework. This editor shows the trace model that is generated by the *UmlToRdbms* transformation, when it is applied to the UML model that is defined in Fig. 1. This transformation constitutes the first step of the exogenous model merging process. Trace models in our framework conform to our traceability metamodel, which was presented in [BCR05]. The traceability editor is constituted by three main frames, the left frame shows an input model of the transformation, the right frame shows the output generated model and the frame in the middle shows the traces that relate elements of the input model to elements of the target model. Traces also provide information about the transformation rule (or relation) that has been applied to source elements to generate the corresponding trace and the related target elements.

4 The *Merge* Operator

The *Merge* operator takes two models as input and produces a third one. If A and B are models that conform to the same metamodel, the application of the *Merge* operator on them produces a model C, which consists of the members of A together with the members of B, i.e. the union of A and B. Taking into account that duplicates are not allowed in a model, the union is disjoint.

To understand the semantics of the *Merge* operator in our example, we need to introduce two concepts: the equivalence relation, for finding duplicates by comparing models, and the conflict resolution strategy, for integrating them.

4.1 The equivalence relation

In an endogenous model merging, an equivalence relation is defined between elements that belong to different models that conform to the same metamodel. To define an equivalence relation among the elements of a model in our approach, the user can use the QVT Relation language in the checkonly mode. Only checkonly transformations with two domains are accepted in this context. Both domains have to refer to the same metamodel in our approach. For the example, we customize the *Merge* operator to merge relational schemas, i.e., models that conform to the *RDBMS* metamodel of Appendix A. To do so we use a checkonly QVT Transformation whose domains refer to the *RDBMS* metamodel. The user can add a QVT relation for each of the classes that appear in the metamodel when it is desired. Such QVT relations act as equivalence relationships that must hold over the elements of two *RDBMS* models. These QVT relations are used in the merging process to check when two elements are equivalent in order to eliminate duplicates.

For instance, the following relation can be defined to indicate that two tables are the same if they belong to the same schema and they have the same name by means of the *tableName* variable[3]:

```
top relation TableEquivalence {
        tableName: String;
        checkonly domain rdbmsDomain1 t1: Table {
                schema = s1:Schema {},
                name=tableName
        };
        checkonly domain rdbmsDomain2 t2: Table {
                schema = s2:Schema {},
                name=tableName
        };
        when {
                SchemaEquivalence(s1, s2);
        }
}
```

where the *SchemaEquivalence* is another QVT Relation defined within the same transformation, describing when two Schema instances are equivalent (for instance, by name). In our approach, this kind of equivalences may involve several instances of two models as in the above example,

[3] We have chosen these criteria for the example. Nevertheless, they can be customized to a specific metamodel by the user. Nothing impedes us to add semantic annotations to the elements of a model and use this information to determine which elements are equals or not.

where *Table* instances and *Schema* instances are used to check whether two tables are equivalent or not.

During the merging process, this checkonly transformation permits checking when groups of elements of different models represent duplicate elements so that they will be merged. In a checkonly QVT transformation, helper functions can be defined by using OCL expressions to manipulate and compare names, and to navigate the structure of the corresponding model. Thus, the user only has to be aware of the standard QVT Relations language and the domain-specific knowledge.

4.2 The conflict resolution strategy

During a model merging process, when two software artifacts (each of which belongs to a different model) are supposed to be equivalent, one of them must be erased. Their syntactical differences may cast doubt on which should be the syntactical structure for the merged element. Here, the conflict resolution strategy comes into play. The conflict resolution strategy is a model transformation that has two input models and one output model, the merged one. The generic semantics of this strategy in our framework consists of the preferred model strategy. When the *Merge* operator is applied to two models, one has to be chosen as preferred (the first argument of the *Merge* operator). In this way, when two groups of elements (that belong to different models) are equivalent due to an equivalence relation, the elements of the preferred model prevail although they may differ syntactically.

To refine the *Merge* operator, the conflict resolution strategy can also be customized. During the merging process, when the *Merge* operator finds two duplicates, they should be integrated. This integration involves a transformation process where information of both duplicates may be taken into account to define the merged model. Thus, an enforced QVT transformation can be used to customize the conflict resolution strategy in the same way a checkonly QVT transformation is used to customize the generic equivalence relation.

A QVT transformation that is used to define a specific conflict resolution strategy has three domains. All of them refer to the metamodel under study (*RDBMS* in our example). The first two domains are defined as checkonly and they only query the two input models of the *Merge* operator. The third domain is defined as enforce and is the one that produces merged elements. In the case study, when we integrate two tables that are equivalent (because they have the same name), we have to integrate their respective columns, primary keys and foreign keys. The following QVT Relation is intended to perform this task:

```
top relation TableMerging {
    tableName: String;
    checkonly domain rdbmsDomain1 t1: Table {
        schema = s1:Schema {},
        name = tableName
    };
    checkonly domain rdbmsDomain2 t2: Table {
        schema = s2:Schema {},
        name = tableName
    };
    enforce domain rdbmsDomain3 t3: Table {
        schema = s3:Schema {},
```

```
        name = tableName
    };
    when {
        SchemaMerging(s1, s2, s3);
    }
    where {
        ColumnMerging(t1, t2, t3);
        PKMerging(t1, t2, t3);
        FKMerging(t1, t2, t3);
    }
}
```

where the *SchemaMerging* QVT relation, which is invoked in the *when* clause, ensures that the container schemas of both *Table* instances must be equivalent in order to apply the current relation to the involved tables. The QVT Relations that are invoked in the *where* clause ensure that the merging process will go on by merging columns, primary keys and foreign keys of the involved tables.

The enforce QVT transformation that the user defines to customize the conflict resolution strategy is automatically compiled into a *ModelGen* equation as briefly introduced in the previous section [4].

4.3 The *Merge* operator

The *Merge* operator takes two models that conform to the same metamodel as inputs. The outputs of the *Merge* operator are a merged model (*merged_model*) and two models of traces ($trac_1$ and $trac_2$) that relate the elements of each input model ($model1$ and $model2$) to the elements of the output merged model. The operator is used as follows:

$$< merged\_model, trac_1, trac_2 >= Merge(model1, model2)$$

The *Merge* operator uses the equivalence relation that is defined for a metamodel to detect duplicated elements between the two input models. When two duplicated elements are found, the conflict resolution strategy is applied to them in order to obtain merged elements, which are then added to the output model. The elements that belong to only one model, without being duplicated in the other one, are copied into the merged model.

The two output trace models are automatically generated by the *Merge* operator on the grounds that it reuses the model transformation mechanism that is described in Section 3, through the conflict resolution strategy. These trace models provide full support for keeping traceability between the input models and the new merged one. The second step of the exogenous model merging in the example constitutes a merging process that involves the model *RDBMS'* and the model *RDBMS*. The model *RDBMS'* is the result of applying the *UmlToRdbms* transformation (defined in Appendix B) to the model *UML* that is defined in Fig. 1, as explained in Section 3. The model *RDBMS* is provided in Fig. 1. In Fig. 3, we show the trace model that is generated during this merging process for the *RDMBS'* model (shown in the left frame of the editor). The model that appears in the right frame of the editor is the final merged relational schema.

[4] More information about the semantics of the *Merge* operator can be found in [BCRL06]

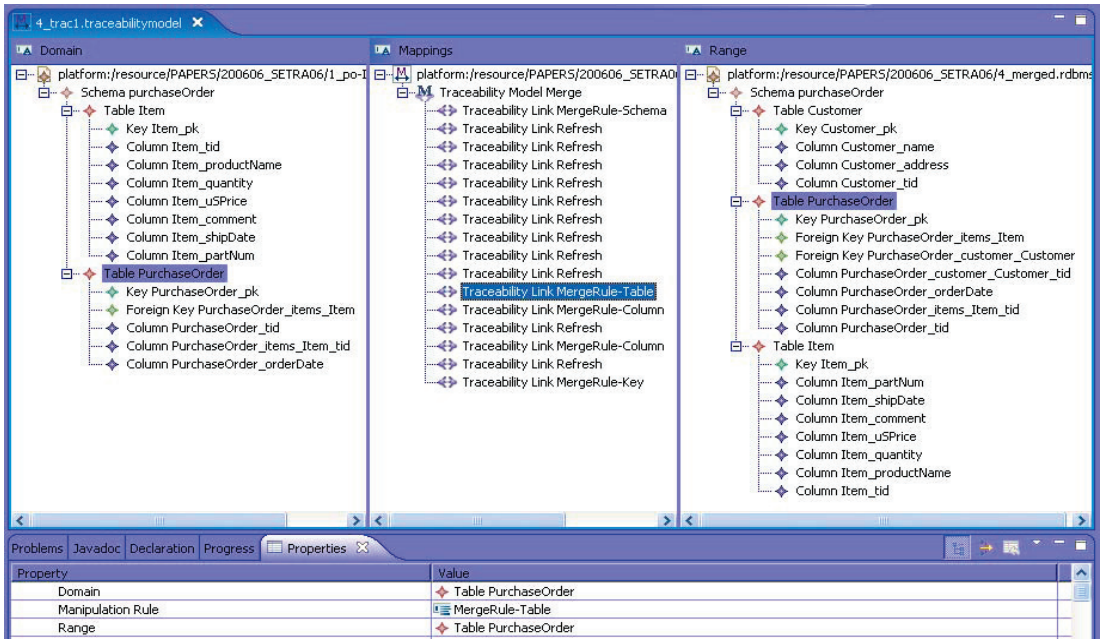


Figure 3: Trace model that is produced during the merging of the models *RDBMS'* and *RDBMS*.

5 Exogenous Model Merging in MOMENT

The exogenous model merging problem consists in the merging of two models that conform to different metamodels, as in the example in Section 2. This problem can be divided into simpler ones that can be solved by two simple model management operators. A composite operator, called *ExogenousMerge*, can be defined for this purpose by composing the *Merge* operator and the *ModelGen* operator. This operator has three arguments: the model A, which conforms to the metamodel *MMA* (the *Ecore* metamodel in our example); the model B, which conforms to the metamodel *MMB* (the *RDBMS* metamodel in our example); and the name of the QVT transformation that must be defined between between the metamodels *MMA* and *MMB* (*umlToRdbms* in our example). In the first step, model A is transformed into a model B', which conforms to the metamodel *MMB* by means of the operator *ModelGen*. This step has been performed in Section 3. In the second step, models B and B' are merged within the metamodel *MMB*. This step has been performed in Section 4. Finally, the merged model *result* is the output of the composite operator. The definition of the *ExogenousMerge* composite operator is as follows:

operator ExogenousMerge (A : MMA, B : MMB, T : Transformation) =
$\quad <B', map_{A->B'} > = ModelGen(T, A)$ (1)
$\quad <result, map_{A->B'}, map_{A->B}> = Merge\ (B', B)$ (2)
return (result)

The *ExogenousMerge* operator is defined independently of any metamodel so that it can be reused to merge two models that conform to any metamodel. In this example, we have not

taken into account the trace models that are generated by the *ModelGen* and *Merge* operators. Nevertheless, another version of the operator could generate traceability models as result of the *ExogenousMerge* operator.

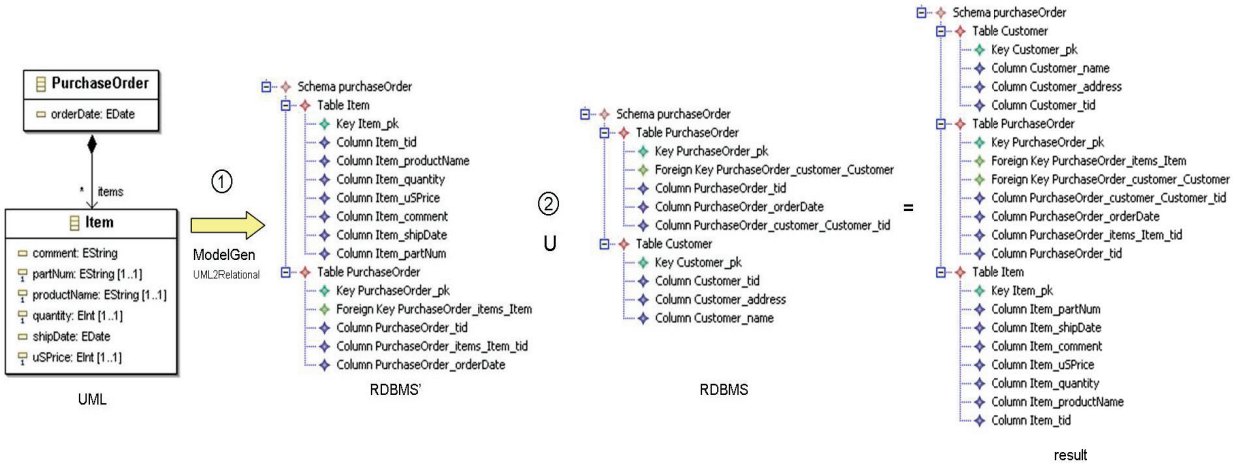


Figure 4: Application of the *ExogenousMerge* operator to the example in Section 2.

Fig. 4 graphically represents the merging process that is performed by the operator *ExogenousMerge* for solving the example that is shown in Section 2. In the example, parameter A corresponds to the UML model and parameter B corresponds to the RDBMS model in Fig. 4. To be able to apply the operator, the equivalence relation for the RDBMS metamodel and the transformation function between the UML and the Relational metamodels must be previously defined by the user.

6 Related Work

Generic model merging approaches take into account the phases that were discussed in [BLN86] to merge database schemas. These approaches can be differentiated by the mechanism that is used to perform model comparison.

[AP03] uses MOF identifiers to compare elements in different versions of a same base model. Although this approach is effective, only versions of a same base model can be compared and merged.

[BP03, BCE+06] provide a set of model management operators to define equivalence relationships between elements of different models by means of mappings, which are used by a merge operator later on. In this approach, the *Merge* operator receives two models (A and B) and a mapping model (mapAB) between them as inputs, and it produces the merged model C and two new mapping models *(mapAC and mapBC): <C, mapAC, mapBC> = Merge (A, B, mapAB)*.

In the AMMA platform [FJ05], the Generic Model Weaver AMW is a tool that permits the definition of mapping models (called weaving models) between MOF models in the ATLAS

Model Management Architecture. AMW provides a basic weaving metamodel that can be extended to permit the definition of complex mappings. These mappings are usually defined by the user, although they may be inferred by means of heuristics, as in [MBR01]. These mapping models are used, together with the mapped models in a model transformation to perform a model composition.

In MOMENT, mapping models are introduced as trace models that are generated by model management operators. This is because operators do not have to rely on them to be applied to a set of models. In MOMENT, mappings between the elements of two models are defined between the elements of their corresponding metamodels by means of checkonly QVT Relations. This permits a clearer specification of composite operators. Trace models are produced by the application of a simple operator to a set of models and keep information about the manipulation task that has been performed to a model.

[KPP06] proposes several domain-specific languages to define model comparison and model merging over metamodels. The model comparison language enhances the definition of equivalence relationships between elements of a metamodel that can be applied over the elements of the corresponding models afterwards. In this language, a differentiation between matching and conformance is provided. While a matching mapping indicates when two elements are equivalent, a conformance mapping indicates when two elements are equivalent and consistent to be merged. In this approach, when an equivalence relationship based on names is used, two elements do not conform to each other if they have different types, for instance. In our approach, we use the QVT Relations language to perform model comparison and model transformation. This feature aims at decreasing the learning curve of our framework since there is only one language, which has been specified as an standard. The QVT Relation language does not provide such a differentiation between conformance and matching. Since two elements that do not conform to each other are usually interpreted as an error, we collapse the conformance and matching conditions in a relation. However, a transformation with conformance relations could be defined for a specific metamodel. Then, this transformation could be specialized with user-defined checkonly relations for defining equivalence relationships.

7 Conclusions

Model merging plays a relevant role in the maintenance and evolution of model-based software. Systems of this kind are usually represented by models that conform to different metamodels. Thus, two kinds of merging processes arise by considering the amount of metamodels that are involved: endogenous merging and exogenous merging. In an endogenous merging process, the models that are merged conform to the same metamodel. In an exogenous merging process, the models that are merged conform to different metamodels.

The MOMENT framework is a model management framework that provides operators to manipulate models on top of a MOF architecture, such as *Merge* for model merging and *ModelGen* for model transformations. In our approach, model management operators are defined independently of any metamodel, keeping a generic infrastructure, but they might be customized by an expert user with domain-specific knowledge by means of standard languages, such as OCL and QVT.

In this paper, we have presented how model transformations are supported in MOMENT through the QVT Relations language and how model transformations play an important role in a model merging process. We have used the standard QVT Relations for this purpose instead of providing new languages for model comparison and model merging. To study the aforementioned kinds of model merging, we have described a solution for an endogenous model merging process by using model transformations through the *Merge* operator. Finally, we have provided a generic solution for exogenous model merging by reusing model transformations and endogenous model merging.

Acknowledgements:

This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

We are grateful to Abel Gómez, Pascual Queralt, Joaquín Oriente and Luis Hoyos for their effort in the development of the MOMENT Framework.

Bibliography

[AP03] M. Alanen, I. Porres. Difference and Union of Models. In Stevens et al. (eds.), *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*. LNCS 2863, pp. 2–17. Springer, 2003.

[BBM03] F. Budinsky, S. A. Brodsky, E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BCE+06] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, M. Sabetzadeh. A manifesto for model merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*. Pp. 5–12. ACM Press, New York, NY, USA, 2006.

[BCR05] A. Boronat, J. A. Carsí, I. Ramos. Automatic Support for Traceability in a Generic Model Management Framework. In Hartman and Kreische (eds.), *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005*. Lecture Notes in Computer Science 3748, pp. 316–330. Springer, 2005.

[BCRL06] A. Boronat, J. A. Carsí, I. Ramos, P. Letelier. Formal Model Merging Applied to Class Diagram Integration. *Electr. Notes Theor. Comput. Sci.*, 2006.

[BDK92] P. Buneman, S. B. Davidson, A. Kosky. Theoretical Aspects of Schema Merging. In *Extending Database Technology*. Pp. 152–167. 1992.

[Beh00] R. Behrens. A Grammar Based Model for XML Schema Integration. *Lecture Notes in Computer Science* 1832:172, 2000.

[Ber03] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*. 2003.

[BHP00] P. A. Bernstein, A. Y. Halevy, R. A. Pottinger. A vision for management of complex models. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 29(4):55–63, 2000.

[BLN86] C. Batini, M. Lenzerini, S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.* 18(4):323–364, 1986.

[BP03] P. A. Bernstein, R. A. Pottinger. Merging Models Based on Given Correspondences. In *Proceedings of the 29th VLDB Conference*. Berlin, 2003. http://www.cs.washington.edu/homes/rap/publications/pottinger-bernstein-vldb03.pdf

[FJ05] M. D. D. Fabro, F. Jouault. Model Transformation and Weaving in the AMMA Platform. In *Pre-proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE'05), Workshop*. Pp. 71–77. Centro de Ciências e Tecnologias de Computaao, Departemento de Informatica, Universidade do Minho, Braga, Portugal, 2005.

[HM05] P. Haase, B. Motik. A mapping system for the integration of OWL-DL ontologies. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*. Pp. 9–16. ACM Press, New York, NY, USA, 2005.

[KPP06] D. S. Kolovos, R. F. Paige, F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*. Pp. 13–20. ACM Press, New York, NY, USA, 2006.

[MBR01] J. Madhavan, P. A. Bernstein, E. Rahm. Generic Schema Matching Using Cupid. In *Proc. VLDB 2001*. Pp. 49–58. 2001. http://www.research.microsoft.com/research/db/ModelMgt/CupidVLDB01.pdf

[Men02] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering* 28(5):449–462, 2002.

[OMG04] OMG, Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/04-10-15). 2004. http://www.omg.org/cgi-bin/doc?formal/2006-01-01

[OMG05] OMG, Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[OWK03] D. Ohst, M. Welle, U. Kelter. Differences between versions of UML diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. Pp. 227–236. ACM Press, New York, NY, USA, 2003.

[SGS+04] G. Straw, G. Georg, E. Song, S. Ghosh, R. B. France, J. M. Bieman. Model Composition Directives. In *UML*. Pp. 84–97. 2004.

A RDBMS Metamodel

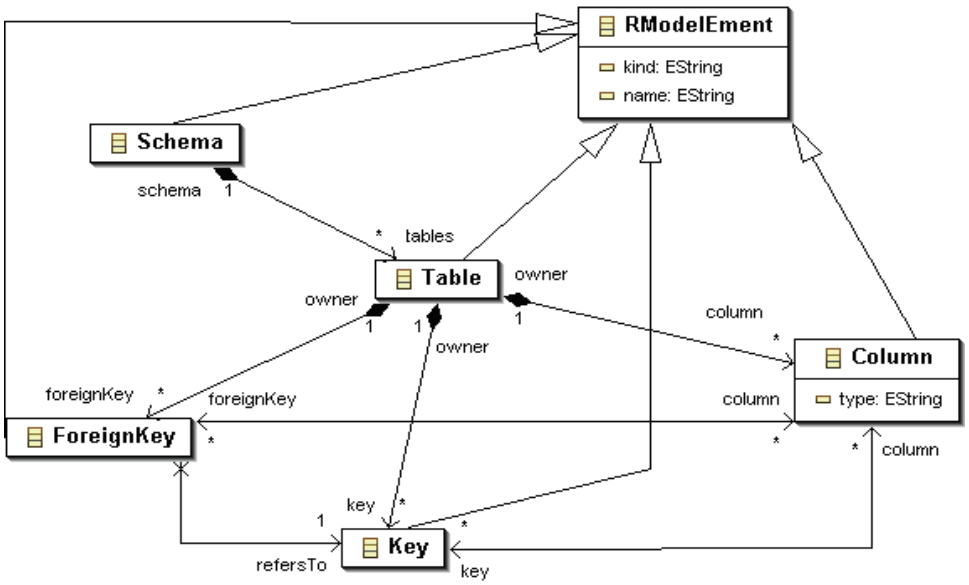


Figure 5: RDBMS metamodel.

B An Ecore to RDBMS Transformation by means of the QVT Relations Language

```
transformation umlToRdbms(ecoreDomain:ecore, rdbmsDomain:rdbms) {
        key Schema {name};
        key Table {schema,name};
        key Column {owner,name};
        key ForeignKey {owner,name};

        top relation PackageToSchema {
            packageName: String;
            checkonly domain ecoreDomain p:EPackage {
                name=packageName
            };
            enforce domain rdbmsDomain s:Schema {
                name=packageName
            };
        }//end PackageToSchema

        top relation ClassToTable {
            className: String;
```

```
    checkonly domain ecoreDomain c: EClass {
        ePackage = p:EPackage {},
        name=className
    };
    enforce domain rdbmsDomain t: Table {
        schema = s:Schema {},
        name = className,
        column = cl:Column {
            name = className + '_tid',
            type = 'NUMBER'
        },
        key = k:Key {
            name = className + '_pk',
            column=cl
        }
    };
    when {
        PackageToSchema(p, s);
    }
    where {
        AttributeToColumn(c, t, className);
    }
}//end ClassToTable

relation AttributeToColumn
{
        checkonly domain ecoreDomain c:EClass {};
        checkonly domain rdbmsDomain t:Table {};
    primitive domain prefix:String;
    where {
        PrimitiveAttributeToColumn(c, t, prefix);
        SuperAttributeToColumn(c, t, prefix);
    }
}//end AttributeToColumn

relation PrimitiveAttributeToColumn
{
    attributeName, columnName, sqltype: String;
    checkonly domain ecoreDomain c:EClass {
        eAttributes = a:EAttribute {}
    };
    checkonly domain rdbmsDomain t:Table {};
    primitive domain prefix:String;
    where {
```

```
        PrimitiveAttributeToColumneAttributes(a,t,prefix);
    }
}//end PrimitiveAttributeToColumn

relation PrimitiveAttributeToColumneAttributes
{
    attributeName, columnName, ecoreTypeName, sqltype: String;
    checkonly domain ecoreDomain a:EAttribute {
        name = attributeName,
        eType = ecoretype: EDataType {
            name = ecoreTypeName
        }
    };
    checkonly domain rdbmsDomain t:Table{};
    enforce domain rdbmsDomain cl:Column {
        name = (
            if (prefix = ”) then
                attributeName
            else
                prefix + ’_’ + attributeName
            endif
        ),
        type = PrimitiveTypeToSqlType(ecoreTypeName),
        owner = t
    };
    primitive domain prefix:String;
    when {
        IsPrimitiveDatatype(ecoreTypeName);
    }
}//end relation

relation SuperAttributeToColumn
    {
    checkonly domain ecoreDomain c: EClass {
        eSuperTypes = sc:EClass {}
    };
    checkonly domain rdbmsDomain t:Table {};
        primitive domain prefix: String;
    where {
        AttributeToColumn(sc, t, prefix);
    }
}

top relation AssocToFKey
{
```

```
        srcTbl, destTbl: Table;
        pKey: Key;
        referenceName, sourceClassName, targetClassName: String;
            checkonly domain ecoreDomain ref: EReference {
            name = referenceName,
            eContainingClass = sc:EClass {
                name = sourceClassName
            },
            eType = tc:EClass {
                name = targetClassName
            }
        };
        enforce domain rdbmsDomain fk:ForeignKey {
            name = sourceClassName + '_' + referenceName + '_' + targetClassName,
            owner = srcTbl,
            column = fkc:Column {
                name = sourceClassName + '_' + referenceName + '_' + targetClassName +
'_tid',
                type = 'NUMBER',
                owner = srcTbl
            },
            refersTo = ObtainReferredPrimaryKey(destTbl)
        };
        when {
            ClassToTable(sc, srcTbl);
            ClassToTable(tc, destTbl);
        }
    }

    function ObtainReferredPrimaryKey(table: Table):Key
    {
        table.key
    }

    function IsPrimitiveDatatype(datatype: String):Bool
    {
        ((datatype = 'EInt') or (datatype = 'EBoolean') or (datatype = 'EString') or (datatype
= 'EDate'))
    }

    function PrimitiveTypeToSqlType(primitiveType:String):String
    {
        if (primitiveType='EInt')
            then 'NUMBER'
        else if (primitiveType='EBoolean')
```

```
            then 'BOOLEAN'
            else
                'VARCHAR'
            endif
        endif
    }
    function IsDirectedReference(ref:EReference):Bool
    {
        (ref.eOpposite -> size() = 0)
    }
}
```

Formal Model Merging Applied to Class Diagram Integration

Artur Boronat, José Á. Carsí, Isidro Ramos, Patricio Letelier
Department of Information Systems and Computation
Polytechnic University of Valencia
Camí de Vera s/n
46022 Valencia-Spain
{aboronat | pcarsi | iramos | letelier}@dsic.upv.es

ABSTRACT

The integration of software artifacts is present in many scenarios of the Software Engineering field: object-oriented modeling, relational databases, XML schemas, ontologies, aspect-oriented programming, etc. In Model Management, software artifacts are viewed as models that can be manipulated by means of generic operators, which are specified independently of the context in which they are used. One of these operators is *Merge*, which enables the automated integration of models. Solutions for merging models that are achieved by applying this operator are more abstract and reusable than the ad-hoc solutions that are pervasive in many contexts of the Software Engineering field. In this paper, we present our automated approach for generic model merging from a practical standpoint, providing support for conflict resolution and traceability between software artifacts. We focus on the definition of our operator *Merge*, applying it to Class Diagrams integration.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: *Computer-aided software engineering, Evolutionary prototyping, Object-oriented design methods*

D.2.7 [**Distribution, Maintenance, and Enhancement**]: *Restructuring, reverse engineering, and reengineering*

D.2.13 [Reusable Software]: *Reuse models*

I.1 [**SYMBOLIC AND ALGEBRAIC MANIPULATION**]: I.1.4 Applications

I.6.5 [**Model Development**]: *Modeling methodologies*

General Terms

Design, Experimentation, Languages.

Keywords

Model-Driven Engineering, Model Management, model merging, conflict resolution.

1. INTRODUCTION

The Model-Driven Development philosophy [1] considers models as the main assets in the software development process. Models collect the information that describes the information system at a high abstraction level, which permits the development of the application in an automated way following generative programming techniques [2]. In this process, models constitute software artifacts that experience refinements from the problem space (where they capture the requirements of the application) to the solution space (where they specify the design, development and deployment of the final software product).

During this refinement process, several tasks are applied to models such as transformation and integration tasks. These tasks can be performed from a model management point of view. Model Management was presented in [3] as an approach to deal with software artifacts by means of generic operators that do not depend on metamodels by working on mappings between models. Operators of this kind deal with models as first-class citizens, increasing the level of abstraction by avoiding working at a programming level and improving the reusability of the solution.

Based on our experience in formal model transformation and data migration [4], we are working on the application of the model management trend in the context of the Model-Driven Development. We have developed a framework, called MOMENT (MOdel manageMENT) [22], which is embedded into the Eclipse platform [5] and that provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF) [6]. Some of the simple operators defined are: the union, intersection and difference between two models, the transformation of a set of models to other model applying a QVT transformation, the navigation through mappings, and so on. Complex operators can be defined by composition of other operators. In this paper, we present the operator *Merge* of the MOMENT framework from a practical point of view. The underlying formalism of our model management approach is Maude [7]. We apply it as a novel solution for the integration of UML Class Diagrams in a Use Case Driven software development process.

The structure of the paper is as follows: Section 2 presents a case study used as an example in the rest of the paper; Section 3 describes our approach for dealing with models by means of an industrial modeling tool, and also informally presents the generic semantics of the operator *Merge*; Section 4 presents the customization of the operator *Merge* to the UML metamodel; Section 5 explains the application of the operator *Merge* to the

Boronat, A., Carsí, J. A., Ramos, I.,Letelier, P. Formal Model Merging Applied to Class Diagram Integration. Proc. of the International ERCIM Workshop on Software Evolution (2006). 2006.

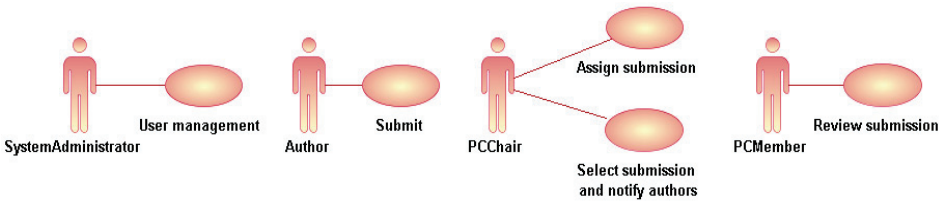


Figure 1. Use Case Model

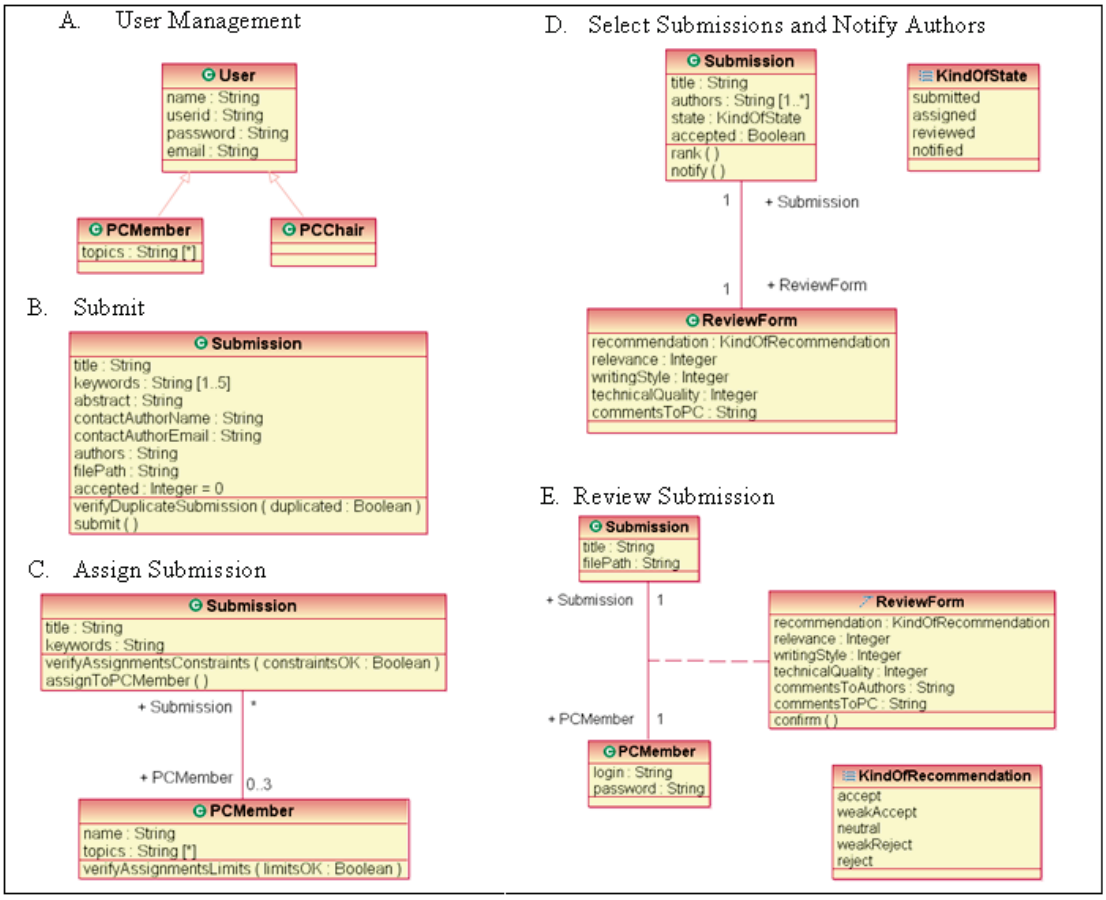


Figure 2. Partial models associated to the corresponding Use Cases

case study; Section 6 discusses some related work; and Section 7 summarizes the advantages of our approach.

2. CASE STUDY: USE CASE ANALYSIS USING PARTIAL CLASS DIAGRAMS

Software development methodologies based on UML propose an approach where the process is Use Case Driven [8, 9]. This means that all artifacts (including the Analysis and Design Model, its implementation and the associated test specifications) have traceability links from Use Cases. These artifacts are refined through several transformation steps. Obtaining the Analysis Model from the Use Case Model is possibly the transformation that has the least chance of achieving total automation. The Use Case Model must sacrifice precision in order to facilitate readability and validation so that the analysis of use cases is mainly a manual activity.

When the Use Case Model has many use cases, managing traceability between each use case and the corresponding elements in the resulting class diagram can be a difficult task. In this scenario, it seems reasonable to work with each use case separately and to register its partial class diagram (which is a piece of the resulting class diagram that represents the Analysis Model). Regarding traceability, this strategy is a pragmatic solution, but when several team members work in parallel with different use cases, inconsistencies or conflicts among partial models often arise, which must be solved when obtaining the integrated model.

We present a case study that illustrates how our operator *Merge* can be used effectively to deal with the required needs established above. We present part of a system for managing submissions that are received in a conference. In our example, we will focus on the fragment of the Use Case Model shown in Figure 1. The actor *System Administrator* manages user accounts. *Authors* submit papers to the conference. The *PCChair* assigns submissions to

PCMembers. Each submission is assessed by several *PCMembers* using review forms. When all the reviews are completed, the *PCChair* ranks the submissions according to the assessment contained in the review forms. Since there is a limit to the numbers of papers that can be presented, and taking into account the established ranking, some submissions are selected, and the rest are rejected. Then, all authors are notified by email attaching the review forms of their submission. Figure 2 shows the Class Diagrams that support the functionality required for the corresponding Use Case.

3. THE GENERIC SEMANTICS OF THE OPERATOR *Merge*

In a Model-Driven Development context [10], models consist of sets of elements that describe some physical, abstract, or hypothetical reality. In the process of defining a model, abstraction and classification are guidelines to be taken into account. A metamodel is simply a model of a modeling language. It defines the structure, and constraints for a family of models.

In our framework, a metamodel is viewed as an algebraic specification where the model management operators are defined so that they can be applied to all the models of the metamodel. To fulfill this in our framework, the Ecore metamodel can be broken down into three well-distinguished parts:

1. A parameterized module called *MOMENT-OP*, which provides our generic model management operators independently of any metamodel (the operator *Merge* is one of them). This module also provides the needed constructors to specify a model as a set of elements, based on an axiomatized specification of a set theory.
2. A signature called *sigEcore*, which provides the constructors of a specific metamodel which is specified in order to represent a model by means of algebraic terms. For example, in the *Ecore* metamodel algebraic specification, we have the constructs that define a class, an attribute, an operation, and so on. This signature is automatically generated from a metamodel (see [23] for further details), which is specified by using the meta-metamodel Ecore in the EMF. In this case, the proper metamodel Ecore has been used to define itself. This signature constitutes the actual parameter for the module MOMENT-OP.
3. A module called *spEcore*, which instantiates the parameterized module *MOMENT-OP* by passing *sigEcore* as actual parameter. In the instantiation process, the generic operators are customized to the constructs of the metamodel. This provides the constructors that are needed to specify a model of this metamodel as a set of elements. This fact also provides the generic operators that can be automatically applied to models of this kind. In this module, the specification of the operator *Merge* can also be customized to a metamodel by simply adding new axioms to the operators. This module constitutes the algebraic specification of a metamodel in MOMENT. To enable the manipulation of UML models, they have to be represented as terms of the spEcore algebraic specification. This task is automatically performed by MOMENT from models expressed in Ecore format in the EMF.

In MOMENT, the operator *Merge* is defined axiomatically using the Maude algebraic language. Maude allows us to specify the operator in an abstract, modular and scalable manner so that we can define its semantics from a generic and reusable point of view. The operator can also be customized in an ad-hoc and more accurate way, taking advantage of both complementary standpoints.

3.1 The Generic Semantics of the Operator *Merge*

The operator *Merge* takes two models as input and produces a third one. If A and B are models (represented as terms) in a specific metamodel algebraic specification, the application of the operator *Merge* on them produces a model C, which consists of the members of A together with the members of B, i.e. the union of A and B. Taking into account that duplicates are not allowed in a model, the union is disjoint.

To define the semantics of the operator *Merge*, we need to introduce three concepts: the equivalence relationship, the conflict resolution strategy and the refreshment of a construct.

First, a semantic equivalence relationship is a bidirectional function between elements that belong to different models but to the same metamodel. This indicates that they are semantically the same software artifact although they may differ syntactically. This relation is embodied by the operator *Equals*. The generic semantics of *Equals* coincides with the syntactical equivalence, although this generic semantics can be enriched by means of OCL expressions that take into account the structure and semantics of a specific metamodel.

Second, we have to deal with conflicts. During a model merging process, when two software artifacts (each of which belongs to a different model) are supposed to be semantically the same, one of them must be erased. Their syntactical differences cast doubt on which should be the syntactical structure for the merged element. Here, the conflict resolution strategy comes into play. The conflict resolution strategy is provided by the operator *Resolve*, whose generic semantics consists of the preferred model strategy. When the operator *Merge* is applied to two models, one has to be chosen as preferred. In this way, when two groups of elements (that belong to different models) are semantically equivalent due to the *Equals* morphism, although they differ syntactically, the elements of the preferred model prevail. The semantics of the Resolve operator can also be customized for a specific metamodel in the same way that we can do with the Equals operator.

Third, refreshments are needed to copy non-duplicated elements into the merged model in order to maintain its references in a valid state. If we merge models B and C in our case study, taking model B as the preferred one, the reference *Submission* of the class *PCMember* of model C is copied to the merged model. As the class *Submission* of model C has been replaced by the one from model B, the reference, which points to the class *Submission* of model C, is no longer valid. Thus, this reference must be updated. The update of a specific metamodel construct term is embodied by the operator *Refresh*.

The operator *Merge* uses the equivalence relationship defined for a metamodel to detect duplicated elements between the two input models. When two duplicated elements are found, the conflict resolution strategy is applied to them in order to obtain a merged

	Model Merging	Conflicts	Models	Resolution
1	<BC, map_{B2BC}, map_{C2BC}> = Merge(B, C)	The multiplicity of the attribute *keywords* (class *Submission*).	B – C	Multiplicity [1..5] (preferred model)
2	<DE, map_{D2DE}, map_{E2DE}> = Merge(D, E)			
3	<BCDE,$map_{DE2BCDE}$, $map_{BC2BCDE}$ > = Merge(DE, BC)	3.1. The multiplicity of the attribute *authors* (class *Submission*)	B – E	Multiplicity [1..*] (preferred model)
		3.2. Type of the attribute *accepted* (class *Submission*)	B – E	Type Boolean (preferred model)
		3.3. Multiplicities of the association between the classes *Submission* and *PCMember*	C – D	Multiplicities 1..1 – 1..1 (preferred model)
4	<ABCDE, $map_{A2ABCDE}$, $Map_{BCDE2ABCDE}$> = Merge(A, BCDE)	4.1. The attribute *userid* (class *User*) and the attribute *login* (class *PCMember*) are identified as the same, by means of the thesaurus.	A – D	The inherited feature prevails by means of the EClass axiom for the operator *Resolve*.

Table 1. The steps of the Class Diagram merging process

```
ceq Equals N1 Model1 N2 Model2 =
  if ((N1.name) == (N2.name)) and (Equals (N1.eContainingClass(Model1)) Model1 (N2.eContainingClass(Model2)) Model2) then true
  else if (Synonym (N1.name) (N2.name)) and (Equals (N1.eContainingClass(Model1)) Model1 (N2.eContainingClass(Model2)) Model2) then true
    else if (Similar (N1.name) (N2.name) 95.0) and (Equals (N1.eContainingClass(Model1)) Model1 (N2.eContainingClass(Model2)) Model2) then true
      else false
  fi fi fi
if (N1.oclIsTypeOf(? "EAttribute") Model1)) and (N2.oclIsTypeOf(? "EAttribute" Model2))          *** Condition
```

Figure 3. Equivalence relationship for the EAttribute primitive

element, which is then added to the output model. The elements that belong to only one model, without being duplicated in the other one, are refreshed and directly copied into the merged model.

The outputs of the operator *Merge* are a merged model and two models of mappings that relate the elements of the input models to the elements of the output merged model. Therefore, these mappings, which are automatically generated by the operator *Merge*, provide full support for keeping traceability between the input models and the new merged one.

4. SPECIFIC SEMANTICS FOR THE ECORE METAMODEL TO MERGE UML CLASS DIAGRAMS

In this section, we present the specific semantics of the operator *Merge* to integrate UML Class Diagrams, which are implemented in the EMF by means of the Ecore metamodel. To define the specific semantics for the Ecore metamodel, we only have to add specific axioms for the operators *Equals* and *Resolve*. Consequently, the axiomatic definition of the operator *Merge* remains the same. The equivalence relationships that relate two elements of the metamodel Ecore take into account the type of a construct, its name, and its container. In an equivalence relationship, the names of two instances that have the same type are analyzed in three steps[1]: two instances may be equal if they have exactly the same name; if not, two instances may be equal if their names are defined as synonyms in a thesaurus; if not, they may be equal if a heuristic function for comparing strings establishes that they are similar within an acceptable range.

Moreover, almost all the relationships add a container condition. This means that two instances of the same type are equal if, in addition to the name condition, they have an equivalent container instance. In Figure 3, we provide the conditional equation that represents an equivalence relationship for EAttribute primitive in MAUDE notation. This equation is automatically obtained by compilation of an OCL expression [23]. In the merging of the partial models B and C of our case study, when this equivalence relationship is applied, the attribute *title* of the class *Submission* (of model B) is equivalent to the attribute *title* of the class *Submission* (of model C) because they have the same name and they belong to equivalent classes.

Several axioms have also been added to the *Resolve* operator in order to take into account the constructs of the Ecore metamodel that contain other elements. For example, a class can contain attributes, references and operations, among others. In the case study, when we integrate the class *Submission* of model B with the class *Submission* of model C, we have to integrate their respective attributes, references and operations.

5. MERGING PROCESS

In this section, we present the merging process that is used to integrate the five partial class diagrams of the case study. The four steps followed are indicated in Table 1, where the first argument for the merge operator is the preferred one. In this table, the first column indicates the step number; the second column shows the invocation of the operator *Merge*; the third column describes some of the main conflicts that have appeared during the merging step; the fourth column indicates the partial models

[1] We have chosen these principles for the example. Nevertheless, they can be customized to a specific metamodel by the user. Nothing impedes us to add semantic annotations to the elements of a model and use this information the determine which elements are equals or not

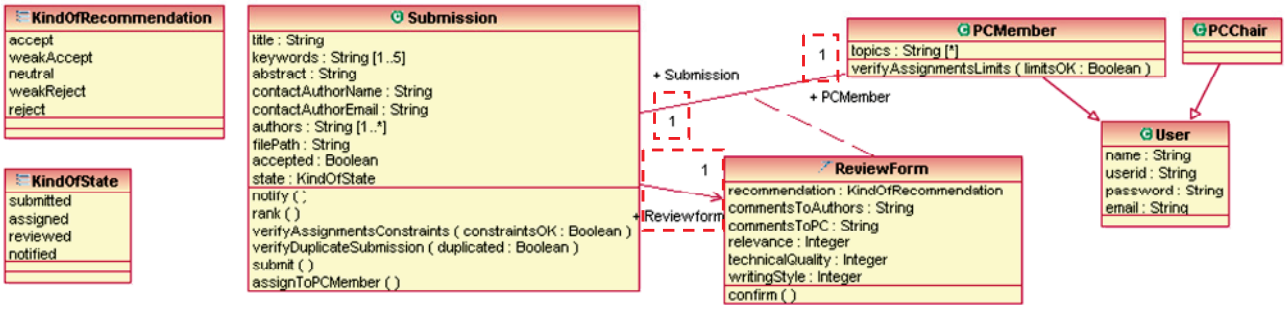


Figure 4. Resulting merged model for the case study.

involved that contain the conflicting elements; and the latter indicates the solution of the conflict by the *Resolve* operator.

After each step of the merging process, two models of mappings are automatically generated. These mappings provide full support for traceability by registering the transformation applied to the elements of the source partial models and by relating them to elements of the merged model. In the MOMENT framework, a set of operators is provided to navigate mappings bidirectionally: from a partial model to the merged model (providing support for the propagation of changes from a specific use case to the merged model, as well as preserving the changes applied to the latter); or from the merged model to a partial class diagram (providing support in order to update a specific use case). Moreover, such mappings are considered as models so that generic model management operators can also be applied to them.

In Figure 4, we show the resulting merged model resulting from step 4. Although the user describes the semantics of the operator *Merge* for a specific metamodel, as the model merging is completely automated, there might exist some undesired results in the merged model that should be fixed. In this figure, elements of this kind are highlighted by a discontinuous line. Therefore, the directed association that comes from partial model D should be deleted, and the multiplicity of the existing association between the *Submission* and the *PCMember* classes should be updated with the multiplicity that appears in partial model C.

In such cases, the user has the option to open the merged model to review and update it. Merged models can be manipulated from visual editors that are integrated in the Eclipse platform, such as the EMF tree editor or the Omondo tool [11], which provides a visual environment to edit UML models based on the Ecore metamodel. Other industrial modeling environments can be used to manipulate resulting models such as Rational Software Architect [12]. These environments provide the added value of code generation and the integration with other IBM software development tools. Furthermore, in the MOMENT framework, the operators that work on mappings can be used to follow the performed merging process in order to automatically detect changes in properties of elements like cardinality, type, addition of attributes to a class, etc.

6. RELATED WORK

In [21], several approaches for model merging are presented. The operator *Merge* is a model management operator that was proposed in [13] and further developed in [14] afterwards. The specification of this operator *Merge* is provided in terms of imperative algorithms so that the operator *Merge* is embodied by a complex algorithm that mixes control logic with the functionality. Although the operator is independent of any metamodel, it depends on an external operator to check the constraints of a specific metamodel. Therefore, it might generate inconsistent models and requires an auxiliary operator to work properly. Moreover, as shown in [14], the algorithm may be changed depending on the metamodel. In MOMENT, the operator *Merge* remains completely reusable for any metamodel. To consider new metamodels, the operators *Equals* and *Resolve* can be customized by simply adding axioms to their respective semantic definition, preserving monotonicity.

Another approach to provide the operator *Merge* from a model management standpoint is presented in [15] by using graph theory. The operator *Merge* is denotationally defined by means of production rules. In both operational and graph-based approaches, the operator *Merge* receives a model of mappings as input. This model indicates the relationships between the elements of the models that are going to be merged. These mappings have to be defined manually or can be inferred by a operator *Match* that uses heuristic functions [16] or historical information [17]. Our operator *Merge* does not depend on mappings since the equivalence relation has been defined axiomatically between elements of the same metamodel in the operator *Equals*, at a higher abstraction level. Another inconvenience of both model management approaches is that they are not integrated in any visual modeling environment. Therefore, they cannot be used in a model-driven development process in the way that the MOMENT framework is able to do through the Eclipse platform.

The Generic Model Weaver AMW [19] is a tool that permits the definition of mapping models (called weaving models) between EMF models in the ATLAS Model Management Architecture. AMW provide a basic weaving metamodel that can be extended to permit the definition of complex mappings. These mappings are usually defined by the user, although they may be inferred by means of heuristics, as in [16]. This tool constitutes a nice solution when the weaving metamodel can change. It also provides the basis for a merge operator on the grounds that a weaving model, which is defined between two models, can be used as input for a model transformation that can obtain the merged model (as mentioned in [19]). In MOMENT, model weavings are generated by model management operators automatically in a traceability model, and can be manipulated by other operators.

An interesting operation-based implementation of the three-way merge is presented in [20]. The union model that permits this kind

of merging is built on top of a difference operator. The difference operator is based on the assumption that all the elements that participate in a model must have a unique identifier. This operator uses the identifiers in order to check if two elements are the same. Our Merge operator is a state-based implementation of the two-way merge so that it does not need a common base model in order to merge two different models. In our approach the operator *Equals* permits the definition of complex equivalence relationships in an easy way. The three-way merge can be specified as a complex operator in the Model Management arena, as described in [14].

More specifically to the problem presented in the case study, UML CASE tools permit the arrangement of Use Cases and their corresponding partial Class Diagram into the same package. Nevertheless, no option is provided to obtain the global Class Diagram from the partial ones. The Rational Rose Model Integration [18] is a tool that provides an ad-hoc solution to merge UML models by basically using the name of the element to determine equivalences, and using the preferred model strategy to obtain the merged model. The equivalence relation and the conflict resolution strategy cannot be customized by the user like in MOMENT. Moreover, once the merged model is generated, there is no way to relate the obtained model to the partial source models in order to keep some degree of traceability.

7. CONCLUSIONS

In this paper, we have presented a state-based automated approach for model merging from a model management standpoint. We have briefly introduced how we deal with algebraic models from a visual modeling environment, and we have described the generic semantics of our operator *Merge*. A customization of the operator has been performed for the Ecore metamodel in order to solve the integration of the partial class diagrams proposed in the case study. The operator takes advantage of the reusability and modularity features of the algebraic specifications. Therefore, it becomes a scalable operator that can be easily specialized to a specific metamodel and that can be intuitively used with other operators. As we have shown in our case study, our approach provides support for maintaining traceability between the Use Case model and the Analysis model. The MOMENT framework offers other operators that enable the synchronization of changes between both models, wherever the changes occur, either in the partial models or in the merged models.

In the current version of the MOMENT framework, the specific semantics of the operator *Merge* is directly introduced using the Maude syntax. In future work, we plan to develop visual interfaces to define the axioms needed to customize the MOMENT operators in order to improve the usability of our tool.

8. REFERENCES

[1] Frankel, D. S.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons OMG Press.

[2] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications.Addisson-Wesley (2000). ISBN 0-201-30977-7, pag. 267-304.

[3] Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.

[4] Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: Two experiencies in software dynamics. *Journal of Universal Science Computer*. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.

[5] Eclipse site: www.eclipse.org

[6] The EMF site: http://download.eclipse.org/tools/emf/scripts/home.php

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.

[8] Kruchten P. The Rational Unified Process: An Introduction. Addison-Wesley Professional. 2003.

[9] Larman C. Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall. 2004.

[10] Mellor, S. J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley (2004). ISBN 0-201-78891-8.

[11] The Omondo site: www.omondo.com

[12] IBM Rational Software Architect: http://www-306.ibm.com/software/awdtools/architect/swarchitect/

[13] Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. MRS Tech. Rep. MSR-TR-2000-53, (in SIGMOD Record 29, 4 (Dec. '00)).

[14] Pottinger, R.A., Bernstein, P. A.: Merging Models Based on Given Correspondences." VLDB 2003.

[15] Song, G., Zhang, K., Kong, J.: Model Management Through Graph Transformation. IEEE VL/HCC'04. Rome, Italy. 2004.

[16] Madhavan, J., P.A. Bernstein, and E. Rahm: Generic Schema Matching using Cupid. VLDB 2001.

[17] Madhavan, J., Bernstein, P. A., Chen, K., Halevy, A.Y., Shenoy, P.: Corpus-based Schema Matching," Workshop on Information Integration on the Web, at IJCAI'2003,pp.59-66.

[18] Rational Suite: http://www-306.ibm.com/software/sw-atoz/indexR.html

[19] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Breton, E, and Gueltas, G : AMW: a generic model weaver. Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). 2005.

[20] Alanen, M, and Porres, I: Difference and Union of Models. In UML 2003 - The Unified Modeling Language, Oct 2003.

[21] Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering, Volume 28 , Issue 5 (May 2002). Pages: 449 – 462.

[22] The MOMENT site: http://moment.dsic.upv.es/

[23] Boronat, A., Ramos, I., Carsí J.A.: Definition of OCL 2.0 Operational Semantics by means of a Parameterized Algebraic Specification. 1st workshop on Algebraic Foundations for OCL and Applications (WAFOCA'06) Valencia, Spain, march 22nd, 2006.

Algebraic Specification of a Model Transformation Engine ♦

Artur Boronat, José Á. Carsí, Isidro Ramos

Department of Information Systems and Computation
Polytechnic University of Valencia
Camí de Vera s/n
46022 Valencia-Spain
{aboronat | pcarsi | iramos}@dsic.upv.es

Abstract. In Model-Driven Engineering, a software development process is a sequence of manipulation tasks that are applied to models, where model transformations play a relevant role. MOMENT (MOdel manageMENT) is a framework that is integrated in the Eclipse platform. MOMENT provides a collection of generic set-oriented operators to manipulate EMF models. In this paper, we present the model transformation mechanism that is embodied by the *ModelGen* operator. This operator uses the term rewriting system Maude as transformation engine and provides support for traceability. *ModelGen* has been defined in an algebraic specification so that we can use formal tools to reason about transformation features, such as termination and confluence. Furthermore, its application to EMF models shows that formal methods can be applied to industrial modeling tools in an efficient way. Finally, we indicate how the *ModelGen* operator provides support for the QVT Relations language in the MOMENT Framework.

Keywords: Model-Driven Engineering, Model Transformation, QVT, Algebraic Specifications, Traceability.

1 Introduction

Nowadays, the Model-Driven Architecture (MDA) [1] and the Software Factories [2] initiatives are leading the Model-Driven Engineering field. Both agree that any software artifact in a software development process can be dealt with as a model. Models provide a more abstract description of a software artifact than the final code of the application. Therefore, working on models increases the productivity in a software development process. It also increases the portability and the quality of the final code by applying generative techniques. In MDA, model transformations have become a relevant issue by means of the forthcoming standard Query/Views/Transformations (QVT) [3]. Since any software artifact can be viewed as a model, model transformation is the basic mechanism that permits the manipulation of software artifacts [4].

♦ This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

Boronat, A., Carsí, J. A., Ramos, I. Algebraic Specification of a Model Transformation Engine. Proc. of the Fundamental Approaches to Software Engineering Workshop (FASE'06) at ETAPS'06. 2006.

Within this arena, the Model Management discipline [5] considers models as first-class citizens and provides a set of generic operators to manipulate them: *Merge*, *Diff*, *ModelGen*, etc. We have developed a framework, called MOMENT (MOdel manageMENT) [6], that is embedded in the Eclipse platform and that provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF). An algebra of model management operators has been specified generically by using the Maude algebraic specification formalism [7] in the MOMENT framework.

In this paper, we focus on the *ModelGen* operator, the model transformation mechanism of MOMENT, which was presented as a proposal in [8]. This operator provides support for the QVT Relations language and also provides support for traceability. *ModelGen* is used by the other model management operators of the Framework when a model manipulation has to be performed. Since the *ModelGen* operator is algebraically specified in Maude, this term rewriting system is used as the underlying runtime environment for model transformations in MOMENT. This fact provides an efficient environment to execute the *ModelGen* operator and a formal environment where algebraic features can be proved, such as the confluence and the termination of a model transformation.

The structure of the paper is as follows: Section 2 presents an overview of the QVT support in the MOMENT Framework and an example; Section 3 presents the algebraic specification of the *ModelGen* operator, how it is related to a QVT transformation, the execution of a model transformation in the Framework, and the analysis of the complexity of the operator; Section 4 compares our approach with other model transformation tools. Finally, Section 5 summarizes the main contributions of this paper.

2 Model Transformations in the MOMENT Framework

The *ModelGen* operator embodies the model transformation mechanism in the MOMENT Framework. This operator has been algebraically specified, although we use it to manipulate graphical models. To deal with models from an industrial standpoint and to manipulate them from a mathematical standpoint, we use two complementary Technical Spaces. A Technical Space (TS) is a working context with a set of concepts, a body of knowledge, tools, required skills, and possibilities [9]. We use the EMF and Maude technical spaces in our Framework. The former is characterized by its interoperability with industrial tools for solving actual Software Engineering problems. The latter constitutes the formal backbone of our model management approach. To achieve the interoperability between them, two kinds of bridges are used in our Framework: two for regular metamodels and one for the QVT Relations metamodel.

For regular metamodels, two bridges are defined between the two technical spaces, at the M2-layer and at the M1-layer (using the Meta-Object Facility [10] terminology). Both of these permit the integration of MOMENT with EMF. The projection mechanism at the M2-layer automatically obtains the algebraic

specification[1] that corresponds to a regular metamodel, by applying generative programming techniques. An algebraic specification that is generated in this way provides the constructors that are needed to define models of the corresponding metamodel as sets. The inverse projection mechanism that obtains an EMF metamodel from an algebraic specification is not relevant in our tool because the algebraic specification must conform to several features in order to be used by our operators and they should be automatically achieved. We also think that visual modeling environments are more suitable for defining these metamodels. At the M1-layer, we have developed a bidirectional projection mechanism that permits us to project an EMF model as a term of an algebra and to project a term of a metamodel algebra as an EMF model. In this case, bidirectionality is needed to apply an operator to an input model since the input model must be serialized as a term and the output term must be deserialized into an EMF model in order to be persisted.

Another bridge is defined for the QVT Relations metamodel. This permits the projection of a QVT transformation as an algebraic specification. This specification constitutes the axiomatic presentation of the *ModelGen* operator, when a transformation is defined among several metamodels.

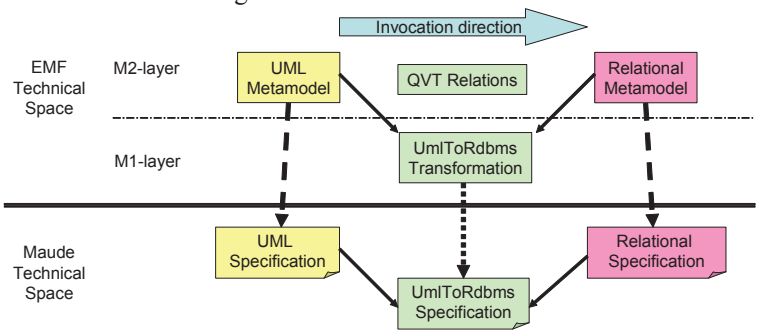


Fig. 1. Overview of the QVT support in the MOMENT Framework

In Fig. 1, the projection bridges between the two Technical Spaces (EMF and Maude) that are used in this work are shown. On the one hand, the discontinuous arrows represent the projection mechanism that obtains an algebraic specification for a regular EMF metamodel. On the other hand, the dotted arrow represents the projection mechanism that obtains the algebraic specification of a model transformation between several metamodels, when it is invoked in one direction. The importation of metamodels in an EMF QVT transformation model is also projected into the Maude TS by using the Maude module importation mechanism (shown as continuous arrows in the figure). Taking into account both kinds of projectors, we can deal with a QVT Relations model as the description of a model transformation or as a regular model. Therefore, it can be transformed by using another QVT Relations

[1] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel; they only permit the representation of information for model management issues.

model. This fact permits us to define the *ModelGen* operator as a mechanism to obtain higher order transformations through the QVT Relations language.

2.1 A Running Example

We have chosen the UmlToRdbms transformation that is presented in the QVT-Merge proposal [3] as an example to illustrate the use of the *ModelGen* operator in the MOMENT Framework. The Ecore metamodel [11] has been used as implementation of the UML metamodel. The Relational metamodel of the QVT-Merge proposal has been specified as an EMF metamodel. Using both metamodels, the UmlToRdbms transformation is applied to the source UML model in Fig. 2 to obtain the target relational schema, which is shown in the figure by using the default EMF graphical modeller.

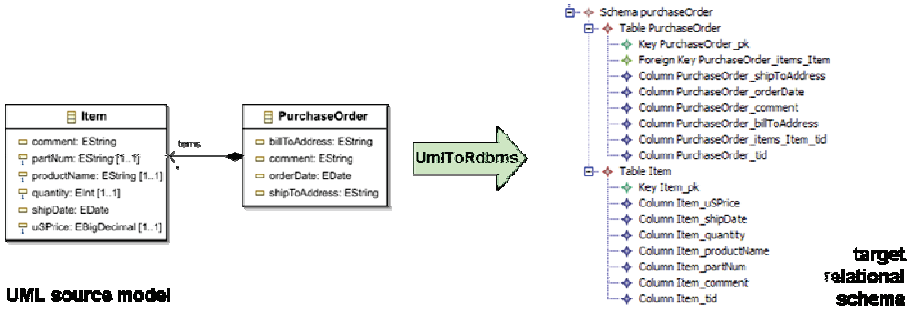


Fig. 2. Example of transformation of a UML model into a relational schema.

3 The Model Transformation Mechanism in MOMENT: the *ModelGen* Operator

In this section, an overview of the QVT Relations language support in MOMENT that is based on the *ModelGen* operator is provided. We indicate how the model transformation in Section 2.1 is specified with the *ModelGen* operator, taking the Relations language as reference point.

3.1 Overview

In MOMENT, a model transformation can be applied to several source models, which may or may not conform to the same metamodel. It generates one target model and a set of traceability models. A traceability model contains a set of traces that relate the elements of the source model to the elements of the target model, indicating which transformation rule has been applied to each source element. To apply a transformation to one or several models in MOMENT, two steps must be performed:

1. The transformation has to be defined as a model of the QVT Relations metamodel by means of a graphical interface or as a program using the Relations language. The transformation model can either be defined by the user or automatically produced by another transformation.
2. The transformation must be invoked by indicating the actual models to be transformed. In MOMENT, this second step is performed automatically. This step can be divided into three phases:
   a) The generation of the Maude modules that contain the algebraic specification of the model transformation operational semantics.
   b) The loading of the corresponding modules into Maude.
   c) The invocation of the *ModelGen* operator, indicating the input models for the transformation. To apply the operator, the input models are projected as terms of the algebra that is obtained from the metamodel of the corresponding input model. The term that is constituted by the operator call and the input terms is thenreduced by Maude, thereby performing the transformation. The resulting models are projected as EMF models. Therefore, any plug-in that provides the suitable graphical editor for these models can be used to show them.

In this paper, we focus on the second step, indicating the structure of the *ModelGen* operator and how it is related to the QVT Relations metamodel.

3.2 Relations: the *ModelGenRule* operator

A QVT relation specifies a relationship that must hold between the model elements of different candidate models. When one or more domains of the relation are enforceable, the relation describes a model transformation. In MOMENT, a QVT relation is projected into several directed transformation rules for the *ModelGen* operator. There is exactly one transformation for each direction in which the relation can be invoked.

The operation *ModelGenRule* that permits the definition of directed transformation rules is declared as follows:

```
op ModelGenRule`(_;_`) : RelationSymbol ParameterList ->
    Tuple{ TTargetMM, TTraceabilityMetamodel,...} .
```

The *ModelGenRule* operator has two parameters: the name of the relation (a term of the sort RelationSymbol) and a polymorphic list of parameters. The membership equational logic [7] is used to define the operational semantics of this operator by means of equations. To use the Maude equational deduction as the runtime engine for our transformation rules, we need them to be confluent and terminating. The first parameter makes the confluence satisfaction easier to achieve. It permits to differentiate two transformation rules, even though they have the same list of parameters. Therefore, we avoid the situation where several equations can be applied to reduce the same term. We discuss the termination issue in more depth in Section 3.2.2. The second parameter permits the definition of a polymorphic list of parameters for the transformation rule. This means that we can define a parameter of any type (either a model or a basic type) as input for the transformation rule.

The result of a transformation rule is a tuple of several elements, where an element can be a model of any of the metamodels involved in the transformation. Among the

models that are produced by a transformation rule, we distinguish the target model and several traceability models. There is one traceability model for each pair *(source model, target model)*.

3.2.1 Structure of a transformation rule in MOMENT

We can only know the direction of a transformation by means of the enforceable property during the invocation process. During this process, the *ModelGenRule* axioms that specify the operational semantics of the relation are generated to be invoked later. Fig. 3 shows the two parts that constitute a *ModelGenRule* axiom: the specification of the model transformation and the specification of the traceability model definition. The Maude code that is generated for each part of a QVT relation is structured as shown in Fig. 3.

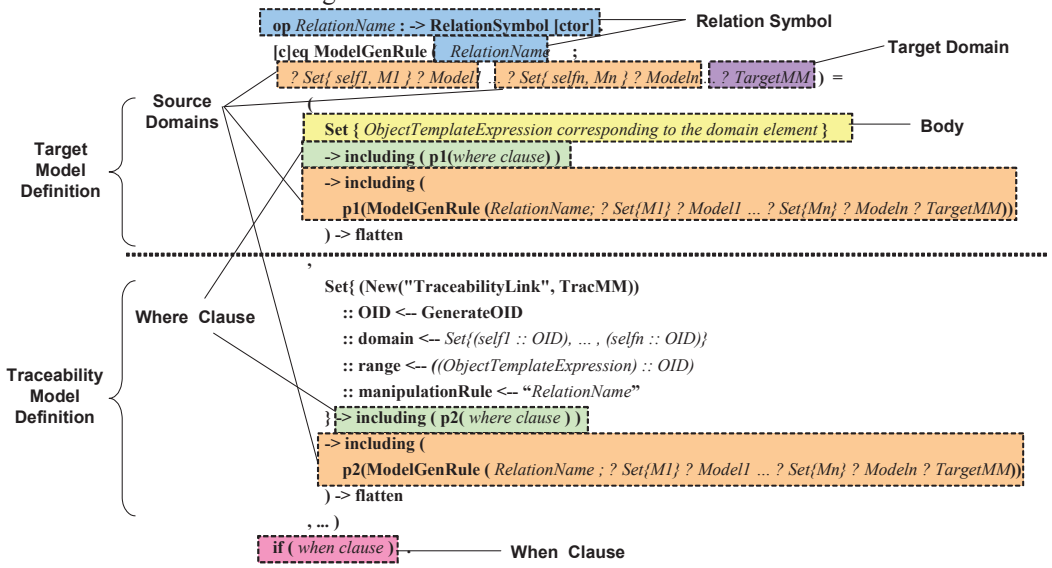


Fig. 3. Axiom for the *ModelGenRule* operator.

- The Relation Symbol. This is the name of the transformation rule. It is used to define a constructor for the sort *RelationSymbol*, which is used in the *ModelGenRule* axioms for the sake of confluence.
- The Domains. They constitute the patterns of the transformation rules and the body. Two kinds of domains can be distinguished in the transformation invocation:
  - The domain that is selected as target when the transformation is invoked is used to generate the body of the transformation rules. In the body of the transformation, object template expressions are used to create new instances in the target model and to obtain the information from the source models. In these object template expressions, OCL is used as the query language and new functions can be declared in a transformation to manipulate data. On the one hand, a parametric algebraic specification has been defined to provide the

operational semantics for OCL 2.0 expressions[2] in MOMENT. On the other hand, Maude itself is suitable to define the operational semantics of QVT functions.

- The source domains of the QVT relation are used to define the pattern of the transformation rule. In the pattern, each input model appears twice. This is needed to achieve two goals at the same time: to use the pattern matching mechanism that Maude provides by means of recursion and to provide support for OCL expressions. The first model is used to search the corresponding element of the pattern that is needed in the transformation rule. The second model is needed because Maude does not provide support for side effects, as in pure functional programming languages. This forces us to keep the whole model throughout the term reduction process in order to navigate it by means of OCL expressions, which can be used in the guard of the pattern. In the following section, we study the pattern matching mechanism in further detail.

- The *When clause*. This clause is used as a precondition for the transformation rule. Therefore, this guard participates in the pattern matching mechanism. It is used to obtain the condition of the equations that constitute the transformation rule. The resulting equations are applied by the conditional pattern matching mechanism of Maude.
- The *Where clause*: this clause is used as postcondition for a transformation rule. In MOMENT, it is used to generate the code that invokes QVT relations that are not defined as *top relations* in the transformation. In this clause, variables can also be initialized with new values to be used in the transformation.

In the QVT proposal, the traceability support is implicit in the QVT Relations language and is explicit in the QVT Core language. The *ModelGen* operator also generates traceability models automatically, but its definition has to be explicitly specified in the *ModelGenRule* axioms. The traceability model produced by the *ModelGen* operator conforms to the traceability metamodel that is presented in [12]. Thus, it can also be manipulated as just another model by the operator *ModelGen*.

In the second part of the axiom shown in Fig. 3, a new trace (instance of the TraceabilityLink class) is added to the traceability metamodel. Its *domain* field is filled with the identifiers (or references) of the source elements that have been matched with the pattern. Its *range* field is filled with the identifiers of all the instances that are created in the body of the transformation rule. Finally, the *manipulationRule* field indicates which transformation rule has been applied. Since the axioms for the *ModelGenRule* transformation rules are automatically generated from a QVT relations model, the traceability support in MOMENT is automatically generated and is kept hidden from the final user.

3.2.2 Pattern Matching

The pattern matching mechanism that is used to apply the transformation rules to source models is provided by Maude through its associative commutative (AC) matching algorithm [13]. To understand how the *ModelGenRule* uses it, we study the

[2] We do not provide further details on OCL 2.0 support because it is out of the scope of this paper.

parts of a transformation rule that are involved in the pattern matching: the *source domains* and the *when clause*.

```
op _,_ : Magma{X} Magma{X} -> Magma{X} [assoc comm ctor] .
op Set{_} : Magma{X} -> Set{X} [ctor] .
```

A source domain is a model used as input in a transformation. It is a set that is constituted by an associative commutative magma of elements. The generic constructors for a Set are the following:

Taking into account that the sort that represents model elements is a subsort of *Magma{X}*, a UML model that is constituted by a package and a class can be defined as a set with the following term: *Set{(UML-Package ...) , (UML-Class ...) }*; where *UML-Package* and *UML-Class* are the constructors that correspond to these concepts in the UML metamodel. By using a recursion mechanism we can use the Maude pattern matching mechanism. We define the following variables: the variable $self_i$ can be bound to a term that represents any element of a $Model_i$ model. The variable M_i can be bound to any magma of elements of a $Model_i$ model. For each QVT relation, we obtain two axioms for the *ModelGenRule* operator by using the following patterns:

- The base case $Set\{ self_i \}$: the right-hand side of this equation is constituted by the tuple of the target model and the traceability models.

```
eq ModelGenRule (RelationName ; ? Set{ self_i } ? Model_i ... ? TargetMM) = (Set{...}, Set{...},...) .
```

- The regular case $Set\{ self_i , M_i \}$: the right-hand side of this equation is constituted by the tuple of the target model and the traceability models, but also contains the application of the recursive transformation rule *RelationName* to the models that are constituted by the residuary magmas of elements. In the first element of the tuple, we apply the recursive operation and we add the first argument of the returning tuple by means of *p1*. In the second element, we use *p2* to get the second element of the returning tuple.

```
eq ModelGenRule (RelationName ; ? Set{ self_i, M_i } ? Model ... ? TargetMM) =
(Set{...}->including(p1(ModelGenRule (PackageToSchema ; ? Set{ M_i } ? Model_i ? TargetMM)))->flatten
,
Set{...}->including(p2(ModelGenRule(PackageToSchema ; ? Set{ M_i } ? Model_i ? TargetMM)) )->flatten) .
```

The *when clause* also provides relevant information to determine whether or not an axiom can be matched to perform a transformation. When the guard described in the *when clause* does not contain an OCL query that searches elements through a model, it can be added to the axiom as a condition, turning the axiom into a conditional equation. When the guard contains an OCL query, the guard is added as an *if…then…else…fi* clause in the body of the equation. Both considerations are needed for the sake of the efficiency of the Maude AC matching algorithm.

When no axiom of the *ModelGenRule* operator can be applied, there is a general axiom for all the transformation rules that is be applied by default. This axiom guarantees the termination of a transformation and is specified as follows:

```
var TR : RelationSymbol .      var PL : ParameterList .
eq ModelGenRule (TR ; PL) = (empty-set, empty-set ) [owise] .
```

As an example of a *ModelGenRule* axiom, we show the axiom that represents the regular case pattern for the following relation *PackageToSchema of the UmlToRdbms transformation* (when the transformation is applied to a UML model):

QVT Relation:

```
top relation PackageToSchema // map each package to a schema
{
    pn: String;
    checkonly domain uml p:Package {name=pn};
    enforce domain rdbms s:Schema {name=pn};
}
```

Maude Axiom:

```
ceq ModelGenRule (PackageToSchema ; ? Set{ self, M } ? Model ? TargetMM) =
(
     Set{ (New("Schema", TargetMM))
          :: OID <-- (self :: OID)
          :: name <-- (self :: name)
     } -> including ( p1(ModelGenRule (PackageToSchema ; ? Set{ M } ? Model ? TargetMM))) -> flatten
,
     Set{ (New("TraceabilityLink", TracMM))
          :: OID <-- generateOID
          :: domain <-- Set{ (self :: OID) }
          :: range <-- Set{ (self :: OID) }
          :: manipulationRule <-- "EPackage-to-Schema"
     } -> including( p2(ModelGenRule (PackageToSchema ; ? Set{ M } ? Model ? TargetMM)) ) -> flatten
)
if (self :: ecore-EPackage) .
```

3.3 Transformations: the *ModelGen* operator

The *ModelGen* operator provides the model transformation mechanism in the MOMENT Framework. This operator corresponds to a *QVT Transformation* when it is invoked in a specific direction. It uses the *ModelGenRule* operator, whose axioms describe the *QVT relations* associated to the transformation. The *ModelGen* operator is declared as follows:

```
op ModelGen`(_;_`) : TransformationSymbol ParameterList
-> Tuple{ TTargetMM, TTraceabilityMetamodel,… } .
```

Similarly to the *ModelGenRule* operator, it has two formal parameters: the symbol that represents the name of the transformation and a polymorphic list of parameters for the transformation. The result of the operator is a tuple that is constituted by the resulting target model and by traceability models. There is one traceability model for each pair *(source model, target model)*.

The code that is generated for a transformation includes the definition of the symbol of the transformation and an axiom that specifies the operational semantics of the transformation. Fig. 4 shows the structure of an axiom for the *ModelGen* operator.

With regard to the definition of a QVT transformation, we identify the following parts:

- The transformation symbol: represents the name of the transformation.
- The parameters of the transformation:
  - Source Domains. *? SourceModel1 ... ? SourceModeln* constitute a list of terms that represent the source models of the transformation.
  - Target Domain. *TargetMM* is a term that represents the target metamodel. It is used to create new instances of the classes that appear in the metamodel by means of the Maude reflection mechanism.
  - List of names of the input and target models. This list is used to define traceability models. This information is needed to indicate the models that are related by means of a traceability model.
- Target model definition. This is the first argument of the resulting tuple, where the target model is generated by applying the *ModelGenRule* axioms (which describe the top QVT relations) to the elements of the source models.
- Traceability model definition. The other arguments of the resulting tuple are the traceability models that relate the elements of the target model to those of each source model. Furthermore, a new instance of the *TraceabilityModel* class is created to relate a source model to the target one.

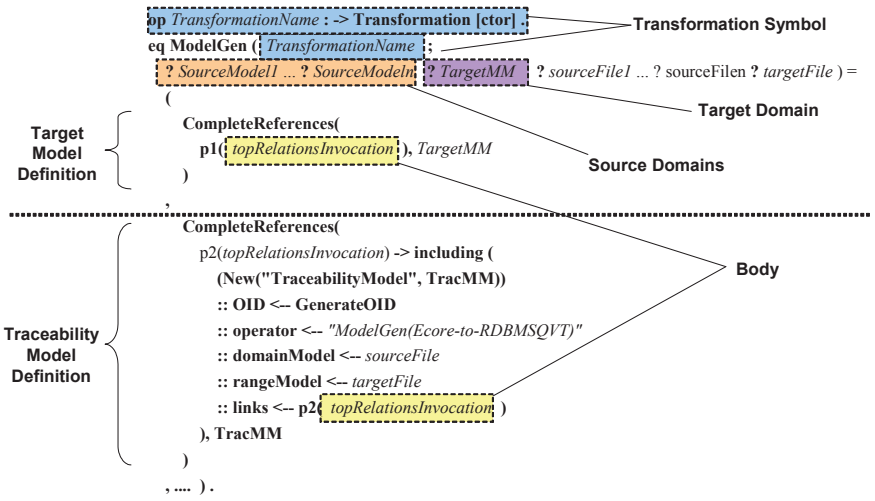


Fig. 4. Axiom for the *ModelGen* operator.

The operator *CompleteReferences* is needed to simplify the definition of transformation rules when the metamodel has opposite references. In EMF, UML associations that may appear in a MOF metamodel are defined by means of opposite references (two instances of the *EReference* class) [11]. If the metamodel has opposite references, both references must be initialized when a model is defined. This fact can make the definition of transformations more complex. Although the EMF editor solves this problem automatically, an independent solution is needed. The *CompleteReferences* operator fulfils this goal. Thus, it permits MOMENT to remain independent of any other technological space that is different from the Maude TS and

to compose several transformations without loss of information. This operator has two input parameters: the model to be completed with references and the corresponding metamodel, which is expressed as a model of the Ecore metamodel. The *CompleteReferences* operator uses the Maude reflection mechanism to traverse the whole input model in order to complete the forgotten references. The output of the operator is the completed model.

Taking into account the example of the UmlToRdbms transformation that is presented in the QVT Merge proposal, we indicate the Maude code that provides the operational semantics of the transformation when it is invoked to transform a UML model into a relational schema. In the *ModelGen* axiom, we have only added the code to obtain the first element of the returning tuple, i.e. the target model.

```
QVT Transformation:
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
{
    top relation ClassToTable {...}
    top relation PackageToSchema {...}
    top relation AssocToFKey {...}
    ...
}
Maude code:
op umlToRdbms : -> Transformation [ctor] .
eq ModelGen (umlToRdbms; ? Model ? TargetMM ? sourceFile ? targetFile ) =
(
    CompleteReferences(
        p1(ModelGenRule (PackageToSchema ; ? Model ? Model ? TargetMM))
        -> including(
            p1(ModelGenRule (ClassToTable ; ? Model ? Model ? TargetMM)))
        -> including(
            p1(ModelGenRule (AssocToFKey ; ? Model ? Model ? TargetMM)))
        -> flatten
    , TargetMM)
, ... ) .
```

3.4 Operator execution

In this section, we present how MOMENT executes the *ModelGen* operator, transforming the UML model of the example in Section 2 into a relational schema. Fig. 5 shows the two MOF layers involved in a model transformation: the M2-layer, where the metamodels and the model transformation are defined; and the M1-layer, where the models are defined and manipulated. The front part of the figure represents the front-end of the MOMENT framework, i.e. EMF and all the plug-ins that are built on it. The back part of the figure represents the formal back-end of the MOMENT Framework, where Maude remains. The traceability support has not been taken into account in the figure.

Fig. 5 represents the transformation of the UML model by using the *ModelGen* operator that is generated from the QVT transformation of the example. The steps that are automatically performed by the MOMENT Framework when the *ModelGen* operator is applied to the source UML model are the following:

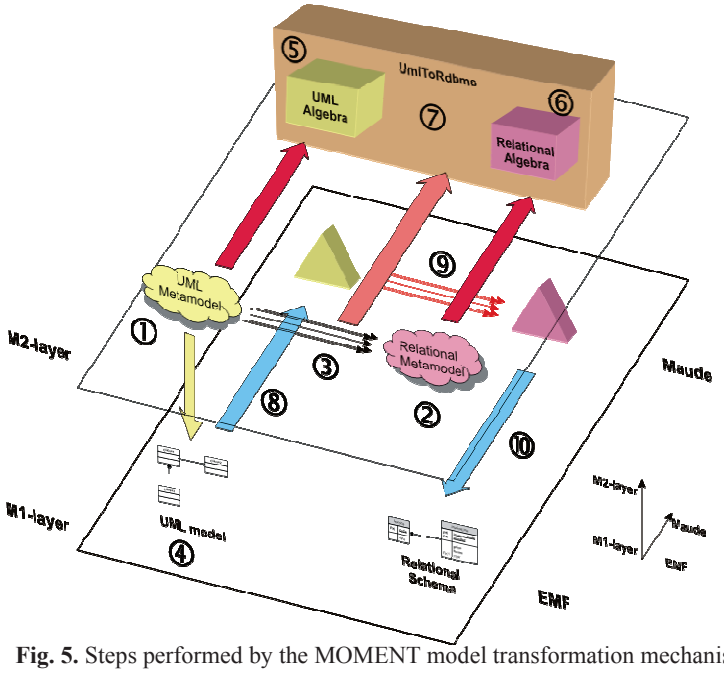


Fig. 5. Steps performed by the MOMENT model transformation mechanism

- *(1) and (2):* We specify both UML and Relational metamodels at the M2-layer by means of the EMF or graphical editors based on this modelling framework. For instance we can also consider XML schemas and Rational Rose models as metamodels.
- *(3)*: The QVT transformation is defined as a model at the M1-layer, but it relates the constructs of the source to the constructs of the target metamodels.
- *(4)*: We define a UML model using a UML graphical editor based on EMF.
- *(5) and (6)*: Both Ecore and UML metamodels, respectively, are projected as algebraic specifications by means of the interoperability bridges that have been implemented in the MOMENT Framework. These algebraic specifications permit each metamodel to be considered as an algebra that provides the constructors to build models as algebraic sets.
- *(7)*: The model that defines the QVT transformation is projected into the Maude TS as the UmlToRdbms algebraic specification, which contains the specification of the *ModelGen* and *ModelGenRule* operators.
- *(8)*: The source UML model, which is defined in step (4) at the M1-layer, is projected into the Maude TS as a term of the UML algebra.
- *(9)*: Maude applies the *ModelGenRule* axioms through its equational deduction mechanism. Thus, Maude constitutes the runtime engine for the MOMENT transformation mechanism.
- *(10)*: This is the last step of the model transformation process. It parses the obtained term in step (9), defining a EMF model in the M1-layer as an instance of the target metamodel defined at the M2-layer.

In the model transformation process, the user only interacts with the MOMENT platform when defining the source and target metamodels (step (1) and (2)), the QVT transformation between both metamodels (step (3)) and the source model (step (4)). The other steps are automatically carried out by the Framework. The output model can also be manually manipulated from a graphical editor.

3.5 Analysis of the Complexity of the *ModelGen* operator

To analyze the complexity of the *ModelGen* operator, we have to take into account the temporal cost of both the *ModelGenRule* operator and the *CompleteReferences* operator.

[13] presents the efficient AC matching algorithm that is used in Maude 2. This algorithm permits an efficient matching of the left-hand side of an equation to a term, when there is a symbol in the term and in the left-hand side of the equation that has been defined as associative and commutative. For instance, the Magma concatenation operation represented by the symbol "," is associative and commutative. The AC matching algorithm permits the substitution of a left-hand side like *Set{self, M}* in a term that represents a model. If we consider that the element *self* must satisfy a guard, the AC matching algorithm determines whether or not an equation can be applied. If we consider *m* as the size of elements of a model, this algorithm can detect the failure of the left-hand side of an equation of this type in $O(log(m))$ time. Furthermore, when the equation can be applied, this algorithm can compute a matching substitution in $O(log(m))$ time.

Taking into account the complexity of this efficient algorithm, we detect the best and the worst case for the *ModelGenRule* operator. Both cases are constituted by a transformation that contains only one transformation rule. In the best case, the transformation rule fails and the default axiom must be applied, finishing the model transformation in $O(log(m))$ time. In the worst case, the transformation rule is applied to all the elements of the model and the computation finishes in $O(m \cdot log(m)) + O(m \cdot R)$ time, where R is the temporal cost associated to the computation of a transformation rule. OCL queries also use the AC matching algorithm to traverse the models. Therefore, the complexity of a transformation rule R is determinant in the analysis of the complexity of the transformation and must be taken into account by the user, since transformation rules are manually coded with the QVT Relations language.

Finally, assuming that the complexity of the *CompleteReferences* is logarithmic, the final temporal cost of a transformation is $O(m \cdot log(m)) + O(m \cdot rule)$ in the worst case. To transform the UML model of the example in Section 2, Maude performed 798620 rewritings in 3328 ms.

4 Related works

We provide a brief comparison of the ModelGen operator to other model transformation mechanisms that are the most current in the Model-Driven Engineering field. The goal of the study is to compare their support for traceability and transformation organization.

MTF [14] is the IBM Model Transformation Framework, which implements some of the QVT concepts and is based on the EMF. It provides a simple declarative language for defining mappings between models. An MTF transformation results in a set of *mappings* that relate objects among different models. The direction of the transformation is defined when the transformation is invoked. Bidirectional

transformations imply constraining the kind of model transformations that can be solved. For instance, transformations that produce loss of information cannot be taken into account.

ATL [15] is a model transformation language that provides declarative and imperative constructs. A transformation is constituted by several transformation rules but there is no mechanism to organize them by means of modules. Traceability support is not provided implicitly in a transformation; traceability models can be generated by a transformation as in the ModelGen operator. The expressions in a transformation rule are defined in OCL expressions, making the ATL language easy to learn and to use. The ATL model transformer is integrated in the NetBeans MDR repository and in the Eclipse platform, allowing the manipulation of EMF models. Tefkat [16] is a model transformation tool that is quite similar to ATL, which is also built on EMF. It incorporates the concept of tracking classes to define traces between the source model and the target model that is generated by the transformation. However, the traces must be defined explicitly in a transformation rule.

XSLT has become a popular alternative for describing model transformations. Tools, such as MTRANS [17] or UMT [18], follow this approach by serializing metamodels and models into XMI documents and then performing the transformations by means of XSLT specifications. A XSLT specification is also an XML document that describes a transformation using both declarative and imperative constructs. Nevertheless, the verbosity of the XML syntax sometimes leads to specifications that are difficult to read and to maintain [19]. XSLT 2 permits the definition of transformations that generate more than one XML document. Therefore, traceability support can be added explicitly to a XSLT transformation. This approach can also be applied to EMF models, which are persisted in XMI. EMF avoids the serialization of certain data, such as derived attributes or default values. Thus, this functionality must be embedded in the XSLT specification. This task can easily become cumbersome.

All these approaches deal with model transformations in a similar way. MTF is the only one that permits the definition of bidirectional relations to perform transformations and that provides implicit traceability support. In the other approaches, the traceability support must be explicitly codified in the transformation function to generate a traceability model. In MOMENT, the set of axioms that describe the ModelGen is automatically generated from a QVT Relations model. Thus, the traceability support is implicit. On the other hand, none of the above tools provide a solution to reuse transformations. In MOMENT, transformations are defined in algebraic modules that can be imported into others. Furthermore, the ModelGen operator can be easily composed with other operators by using functional composition since we are working in an algebraic context. This feature avoids the development of complex frameworks to deal with transformation composition.

5 Conclusions

In this paper, we have presented the ModelGen operator, which permits the definition of directed declarative transformations that can be applied to several source models,

which may or may not conform to the same metamodel. Furthermore, a transformation can be parameterized with additional data that can act as control parameters allowing configuration and tuning. The return value of a transformation is a tuple that is constituted by a model and several traceability models. There is one traceability model for each pair *(source model, target model)*. ModelGen uses the efficient AC matching algorithm that is implemented in Maude 2 as the pattern matching mechanism to apply transformation rules to the source models. The traceability support that is provided in MOMENT [12] permits the definition of an incremental transformation operator in an easy way, similar to the PropagateChanges operator, which is also presented in [12].

Our formal approach for model transformation permits the application of advantageous features to this field, such as transformation composition or modularity. Furthermore, formal features (like confluence and termination) can also be studied. The MOMENT Framework shows that formal methods can be applied to industrial tools not only for proving theoretical aspects, but also for solving actual problems in an efficient manner. Nevertheless, an algebraic setting might not be the most user-friendly environment to work on models. This reason led us to provide support for QVT by using generative programming techniques, as shown in this paper. In this way, Maude remains hidden from the final user although its formal features are used in our framework.

References

1. Frankel, D. S.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons OMG Press. January, 2003.
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons. 2004.
3. QVT-Merge Group, "Revised Submission for MOF 2.0 Query/View/Transformation RFP(ad/2002-04-10)", Version 2.0, ad/2005-03-02, March 2005
4. Sendall, S., Kozaczynski, W. Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software. September/October 2003 (Vol. 20, No. 5), pp. 42-45.
5. Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.
6. The MOMENT web site: http://moment.dsic.upv.es:8080
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.
8. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. IEEE Computer Society Press. 9th European Conference on Software Maintenance and Reengineering. Manchester, UK. 2005.
9. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.
10. OMG, "Meta Object Facility 1.4", http://www.omg.org/technology/documents/formal/mof.htm
11. EMF web site: http://www.eclipse.org/emf/

12. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Support for Traceability in a Generic Model Management Framework. In Proceedings of European Conference on Model Driven Architecture - Foundations and Applications. Nuremberg (Germany). 2005. (To appear)
13. Eker, S.: Associative-Commutative Rewriting on Large Terms. In Proceedings of the 14th International Conference on Rewriting Techniques and Applications. Valencia, Spain, 2003.
14. The MTF web site: http://www.alphaworks.ibm.com/tech/mtf
15. Bézivin, J., Dupé, G., Jouault, F. , Pitette, G., and Rougui, E.J.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: OOPSLA 2003 Workshop, Anaheim, California.
16. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In Model Transformations In Practice Workshop, Montego Bay, Jamaica, October 2005.
17. M. Peltier, Jean Bézevin, and G. Guillaume. MTRANS: A general framework, based on XSLT for model transformations. In WTUML '01, Proceedings of the workshop on Transformations in UML, Genova, Italy, 2001.
18. The UMT web site: umt-qvt.sourceforge.net/
19. A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood. Transformation: The Missing Link of MDA, In A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.): Graph Transformation: First International Conference (ICGT 2002), v1 15 Barcelona, Spain, October 7-12, 2002. Proceedings. LNCS vol. 2505, Springer-Verlag, 2002, pp. 90 - 105

Definition of OCL 2.0 Operational Semantics by means of a Parameterized Algebraic Specification ♦

Artur Boronat, Isidro Ramos, José Á. Carsí

Department of Information Systems and Computation
Technical University of Valencia
C/Camí de Vera, s/n
46022 Valencia-Spain
{aboronat | iramos | pcarsi}@dsic.upv.es

Abstract. This paper presents an algebraic specification of the operational semantics of part of the OCL 2.0 standard. This specification is used in a model management tool that provides support for model-driven tasks within the Eclipse platform. The algebraic specification of OCL has been developed in Maude by using its parameterization mechanism, providing a suitable modularization and making reuse easier. In this work, OCL can be used in the Eclipse Modeling Framework to represent models in an algebraic setting and to perform queries or constraints over software artefacts that can be represented as models.

Keywords: OCL queries, Eclipse Modeling Framework, algebraic specification, Model-Driven Engineering.

1 Introduction

Model-Driven Development is a field in Software Engineering that, for several years, has been representing software artefacts as models in order to improve productivity, quality, and economic incomes. Models provide a more abstract description of a software artefact than the final code of the application. A model can be built by defining concepts and relationships. The set of primitives that permit the definition of these elements constitute what is called the metamodel of the model. The emergence of important model-driven initiatives such as the Model-Driven Architecture [1], supported by OMG, and the Software Factories [2], supported by Microsoft, ensures a model-driven technology stock for the near future.

In the MDA context, the standard Meta-Object Facilities (MOF) [3] provides a way to define metamodels. The standard proposal Query/Views/Transformations (QVT) [4] will provide support for both transformations and queries. While model transformation technology is being developed [5-7], the Object Constraint Language (OCL) remains as the best choice for queries.

♦ This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

Boronat, A., Ramos, I., Carsí, J. A. Definition of OCL 2. 0 Operational Semantics by means of a Parameterized Algebraic Specification. Proc. of the 1st International Workshop on Algebraic Foundations for OCL and Applications. 2006.

OCL [8] is a textual language that is defined as a standard "add-on" to the UML standard. It is used to define constraints and queries on UML models, allowing the definition of more precise and more useful models. It can also be used to provide support for metamodeling (MOF-based and Domain Specific Metamodeling), model transformation, Aspect-Oriented Modeling, extensions for behaviour specifications, support for model testing and simulation, ontology development and validation for the Semantic Web [9]. Despite its many advantages, while there is wide acceptance for UML design in CASE tools, OCL lacks a well-suited technological support.

In this paper, we present an algebraic specification of generic OCL queries, by using Maude [10], that can be used in a MOF-like industrial tool. The OCL algebraic specification permits the study of formal features and their proof, and the execution of OCL expressions as well. This algebraic specification has been developed in the framework MOMENT (MOdel manageMENT) [11], which provides a set of generic operators to deal with models. The MOMENT operators use OCL queries to perform model queries and transformations.

The structure of the paper is as follows: Section 2 provides an example; Section 3 describes the algebraic specification of OCL, indicating the support for basic data types and collection types, and the support for collection operations; Section 4 presents the integration of the algebraic specification of OCL within an industrial modelling framework, indicating the automatic support for representing software artefacts as algebraic specifications that can be queried; Section 5 presents some related works; Section 6 provides some conclusions and ongoing work.

2 The Rdbms Metamodel

The Meta-Object Facility standard (MOF) provides a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata-driven systems. The main achievement of this standard is the definition of a common terminology in the Model-Driven Architecture initiative, which can be used conceptually in other model-driven approaches.

As an example, in this paper, we have taken the Rdbms metamodel that has been provided in the QVT standard as case study. This metamodel provides some primitives to define relational schemas as models. The metamodel can be considered a UML model, as shown in Figure 1. Thus, the OCL-like specification that is presented can also be used for queries over any software artefact that might be defined following the MOF conceptual framework: metamodels, regular models, and instances of models.

OCL invariants permit a more precise definition of a UML model by adding constraints. In this case, we use OCL invariants to make the definition of the Rdbms metamodel more precise. For instance, to define foreign keys, we must ensure that the number of columns that participate in a foreign key must be the same as the number of columns that participate in the corresponding referred primary key. In addition, the type of the columns that participate in a foreign key must be equal to each corresponding column (by order) of the referred primary key. This constraint is expressed by means of the following invariant:

```
context ForeignKey:
inv: if (self.column->size() = self.refersTo.column->size()) then
        self.column->forAll(c:Column |
            self.refersTo.column-> at(self.column->indexOf(c)).type = c.type
        )
     else false
     endif
```

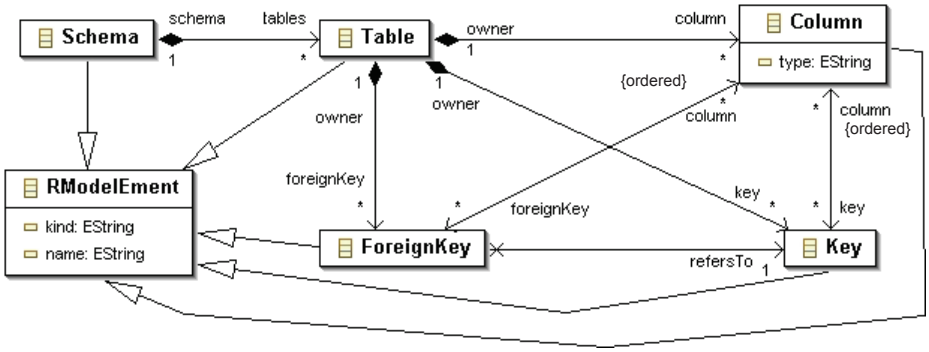


Fig. 1. Rdbms metamodel

3 Algebraic Specification of Generic[1] OCL Queries

In this section, we describe the parameterized algebraic specification of OCL that permits the query of either metamodels or UML models.

3.1 Functional Modules in Maude

In Maude, functional modules describe data types and operations on them by means of membership equational theories. Mathematically, such a theory can be described as a pair (Σ, E ∪A), where: Σ is the signature that specifies the type structure (sorts, subsorts, kinds, and overloaded operators); E is the collection of equations and memberships declared in the functional module; and A is the collection of equational attributes (associativity, commutativity, and so on) declared for the different operators. Computation is the form of equational deduction in which equations are used from left to right as simplification rules, with the rules being Church-Rosser and terminating.

A signature in membership equational logic (MEL) is a triple Σ = (K, S, OP), where: K represents a set of kinds; S= $\{S_k\}_{k\in K}$ represents a pairwise disjoint K-kinded

[1] In this work, OCL genericity refers to the possibility of reusing the OCL specification for any software artifact that can be represented as a model, including metamodels.

family of sets of sorts; and $OP = \{op_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K^+}$ represents a set of many-kinded operations. For a specific operation: op is the operation symbol; $k_i \in K^+$ (i=1..n) are the argument sorts; and k is the range sort.

3.2 Algebraic Specification of OCL Types

Types in OCL are divided in basic data types, collection types, and user-defined types. In this section, the algebraic support for the first two kinds of types is presented.

3.2.1 Basic Data Types

In OCL, there are four basic data types [12] that have a direct correspondence to Maude basic data types. In Table 1, we show the correspondences between OCL 2.0 and the Maude data type system and their corresponding primitives. In the table, when the operations have different symbols in OCL and Maude, we indicate the Maude symbol in brackets.

OCL 2.0	Maude	Common operators
Boolean	Bool	or, and, xor, not, = (==), <> (=/=), implies, if-then-else-endif (if-then-else-fi)
Integer	Int	= (==), <> (=/=), <, <=, >, >=, +, -, *, / (quo), mod (rem), abs (abs), max, min
Real	Float	/, round (ceiling), floor
String	String	concat (+), size (length), substring(substr), = (==), <> (=/=)

Table 1. OCL and Maude data type correspondences

There are also differences in invoking the operations in OCL and in Maude. For example, to invoke the *size* operator over a string *a,* we use dot notation: *a.size()*; while in Maude, we use the prefix notation: *length(a)*.

The Maude data type system is richer than the OCL data type system, and provides more types such as support for identifiers and rational numbers, and related operations. Thus, Maude operations can also be used in OCL queries.

3.2.2 Collection Types

OCL provides four specific collection types that are defined as follows:

- A Set is a collection that contains instances of a valid OCL type, where order is not relevant and duplicate elements are not allowed.
- An OrderedSet is a set whose elements are ordered.
- A Bag is a collection that may contain duplicate elements. Elements in a bag are not ordered.
- A Sequence is a bag whose elements are ordered.

The OCL support has been specified generically in the parameterized module *OCL-SUPPORT(X :: TRIV)*, where the formal parameter *X* has the trivial theory as type. The trivial theory only contains a sort *Elt* (referred to as *X$Elt* in the OCL specification) that represents the sort of elements that can be contained in an OCL collection. This sort represents the *OCLAny* type of the standard OCL specification. The *OCL-SUPPORT(X::TRIV)* module imports the basic data types and provides the

constructors that are needed to define collections of elements. It provides collection operators as well.

To take into account the uniqueness and order features of an OCL collection, we introduce two intermediate sorts and their constructors (shown in Table 2): *Magma{X}* and *OrderedMagma{X}*. Basically, we define the sort *Magma{X}* as the sort of the term that represents a set of elements that are not ordered by means of the association and the commutativity attributes. Instead, the constructor of the sort *OrderedMagma{X}* does not have the commutativity property, producing terms that represent ordered concatenations of elements.

```
1. sorts Magma{X} OrderedMagma{X} .
2. subsort X$Elt < Magma{X} .
3. subsort X$Elt < OrderedMagma{X} .
4. op _,_ : Magma{X} Magma{X} -> Magma{X} [assoc comm ctor] .
5. op _::_ : OrderedMagma{X} OrderedMagma{X} -> OrderedMagma{X} [assoc ctor] .
```

Table 2. Specification of groups of elements

Terms of the sort *Magma{X}* are used to define sets (line 4) and bags (line 8), while terms of the sort *OrderedMagma{X}* are used in ordered sets (line 6) and sequences (line 10). In Table 3, we show the Maude code that specifies the four types of collections. In our specification, collections of collections are allowed by indicating that one collection can be an element of another collection (line 2). The sort *Collection{X}* can be considered as an abstract concept on the grounds that there is no specific constructor for it. Each collection has a constant constructor that defines an empty collection (lines 5,7, 9, 11).

```
1. sorts Collection{X} Set{X} OrderedSet{X} Bag{X} Sequence{X} .
2. subsort Collection{X} < X$Elt .
3. subsorts Set{X} OrderedSet{X} Bag{X} Sequence{X} < Collection{X} .
4. op Set{_} : Magma{X} -> Set{X} [ctor] .
5. op empty-set : -> Set{X} [ctor] .
6. op OrderedSet{_} : OrderedMagma{X} -> OrderedSet{X} [ctor] .
7. op empty-orderedset : -> OrderedSet{X} [ctor] .
8. op Bag{_} : Magma{X} -> Bag{X} [ctor] .
9. op empty-bag : -> Bag{X} [ctor] .
10. op Sequence{_} : OrderedMagma{X} -> Sequence{X} [ctor] .
11. op empty-sequence : -> Sequence{X} [ctor] .
```

Table 3. Specification of OCL collection types

In this specification, the uniqueness property of both the collection Set and the collection OrderedSet is checked in the operations that join two collections: *union*, *intersection* and *including* for Set, and *union*, *append*, *prepend*, *insertAt* and *including* for OrderedSet.

A view has been defined for each Maude simple data type in order to deal with collections of simple data types. For instance, to deal with collections of integers, the following view is defined: *view INT from TRIV to INT is sort Elt to Int . endv*

This view is used to instantiate the OCL-SUPPORT(X) module as OCL-SUPPORT(INT). This way, the following example is a valid collection of integers:
OrderedSet{ Set{1, 2, 3} :: Bag{1, 2, 3, 3} :: Sequence{3 :: 3 :: 2 :: 1}}

3.3 OCL Collection Operations

Two kinds of operations on collection types can be distinguished in OCL 2.0: regular operations and loop operations or iterators. Regular operations provide common functionality over collections. Loop operations or iterators permit looping over the elements in a collection performing a specific action.

3.3.1 Regular Collection Operations

To specify the common operations, a classification of the operations has been performed taking into account the return type of each operation (shown in Table 4). Each kind of operation is identified by a sort. We define the OCL operations as constant constructors of one of these sorts. For instance, the *size* operation, which returns an integer, is defined as follows: *op size : -> IntFun{X} [ctor]* .

	Return type	Collection operator symbols					Iterator symbols
		Collection	Set	OrderedSet	Bag	Sequence	Collection
Fun{X}	Collection	union, flatten, including, excluding, iterate	--, inter-secti-on	--, insertAt, append, prepend	intersection	insertAt, append, prepend	select, reject, any, sortedBy, collect, collectNested, iterate
EltFun{X}	Element			first, last, at		first, last, at	
BoolFun{X}	Boolean value	includes, includesAll, excludes, excludesAll, isEmtpy, notEmpty					one, forAll, forAll2[2], exists, isUnique
IntFun{X}	Integer value	count, size, sum, product		indexOf		indexOf	

Table 4. OCL collection operations that have been specified

To make these operations work, two more elements are needed. In the signature, a skeleton operator is needed to use operations in an OCL-like way. In the axiomatic presentation, axioms are needed to define the operational semantics of each operation.

The skeleton of an operation defines its syntactical structure and can be reused for several operations. For instance, to invoke the size operator in an OCL-like way, we provide the following operator: `op _->_ : Collection{X} IntFun{X} -> Int .` ; where the first argument is a collection, and the second argument is the corresponding operator symbol. To invoke the *size* operator over a set A, we proceed as follows: *A -> size* . The axiomatic definition of the operations defines the use of operation terms in the skeleton constructor. For instance, the operational semantics of the operation *size* is axiomatically defined for the sort *Set{X}* by using recursion as follows:

```
var N : X$Elt . var M : Magma{X} .
eq Set{ N , M } -> size = (Set{ M } -> size) + 1 .
eq Set{ N } -> size = 1 .
eq empty-set -> size = 0 .
```

[2] The forAll2 operation has been included to provide support when two iterators are being used in the forAll operation.

However, the same skeleton operator is used to define the size operator for the sort *Sequence{X}*:

```
eq Sequence{ N :: OM } -> size = (Sequence{ OM } -> size) + 1 .
eq Sequence{ N } -> size = 1 .
eq empty-sequence -> size = 0 .
```

Nonetheless, we cannot use the *count* symbol (representing the count operator) as a term in the above skeleton constructor because it is not defined in the OCL axiomatic presentation. This is due to the fact that the count operator has an argument.

3.4.2 Loop Operations or Iterators

An iterator permits the loop over the elements of a collection. Every loop operation has an OCL expression as parameter. This is called the body, or body parameter, of the operation. As a guiding example, we use a standard OCL expression that permits obtaining the odd numbers from a set of integers:

```
Set{1,2,3,4,5,6} -> select(i | i.mod(2) <> 0)
```

In this expression, *select* is the iterator operation while the expression *(i | i.mod(2) <> 0)* is the body. A body expression may have arguments to process external information. In the OCL algebraic specification, these arguments are considered by means of the sorts *Parameter* and *ParameterList*. *Parameter* has a polymorphic constructor that allows the use of any Maude data type:

```
op ?_ : Universal -> Parameter [ctor poly (1)] .
```

This constructor allows defining the following examples as valid parameters: *? 1, ? 1.0, ? 'a, ? "a", ? true*. To obtain the actual value that is enclosed in a parameter, the following casting operators can be used: *asCollection, asElt, asInt, asFloat, asString, asBool, asQid*. For example, the following expression obtains the corresponding integer value: *red (? 1 :: asInt)* .

ParameterList is a list of *Parameter* terms. The *Parameter* sort is subsort of the *ParameterList* sort indicating that a parameter list can be constituted by a single parameter. There are three constructors for lists (shown in Table 5): the *Parameter* constructor; the *empty-params* is the constant that constitutes an empty list; and the concatenation operator, which builds a list by means of blank spaces, in which the *empty-params* constant has the identity property. Thus, *? 1 ? "a"* is a valid parameter list.

```
subsort Parameter < ParameterList .
op empty-params : -> ParameterList [ctor] .
op      : Parameter ParameterList -> ParameterList [ctor id: empty-params] .
```

Table 5. Constructors for the the ParameterList sort.

Both iterator operations and body expressions are considered in the algebraic specification separately. This separation is needed to simulate higher-order functions in Maude by considering body functions as terms that can be passed as arguments to iterator operations. More precisely, the body of an iterator is associated to a symbol, which is defined as a term of a Body{X}. Thus, the iterator specification is defined independently of the specification of the body. To specify iterators and body operations, the following elements are included in the specification:

– Body sorts. Several sorts of body functions can be defined depending on their resulting type, as in collection operations. The sorts that can be used are the

following: Body{X} (for functions that return a collection or an element); BoolBody{X} (for boolean functions); RatBody{X} (for functions that return a rational or an integer value); FloatBody{X} (for functions that return a float value); StringBody{X} (for functions that return a string value); and IdBody{X} (for functions that return a Qid value).

- Body symbols. These are the names of the functions. They are not included in the OCL specification since they must be defined by the user depending on the desired functionality. For instance, to define a body function, called *isOdd,* that returns an boolean value, we use the following constructor:
  op isOdd : -> BoolBody{RAT} [ctor] .
- Body skeleton constructor. To provide the way body functions must be defined, skeleton constructors are provided for each body sort in the following way:
  op _::_`(_;_`) : BodyElt{X} BoolBody{X} ParameterList Collection{X} -> Bool .
  In all of them, the first argument is a term that represents either a magma of elements or an ordered magma, the second is the corresponding body symbol, the third one is a variant list of parameters that can be empty, and the last one is the whole initial collection to which the first argument belongs.
- Body axiomatic definition. This constitutes the specification of the body function out of the iterator operation. This makes a difference between the standard OCL specification and our OCL algebraic specification. To define a body function, the axioms must be provided by the user in Maude notation. For instance, to define a boolean function that determines whether an integer number is odd or not using the previously defined *isOdd* symbol, we define the following axiom:

  var intN : Int . var intCol : Collection{RAT} . var PL : ParameterList .
  eq intN :: isOdd (PL ; intCol) = ((intN rem 2) =/= 0) .

  Where intN is an integer variable, intCol is a collection of rational numbers (also integer numbers), and PL is the list of parameters, which is ignored in this case.
- Sorts of iterators. These are the same sorts of the operations that classify collection operations by their return type.
- Iterator symbols. The symbol of the iterators corresponds to the name of the iterators that are defined in the specification. The iterators that are considered in our specification are presented in Table 4, indicating their corresponding sorts.
- Iterator skeleton constructors. They permit both the invocation of the iterator in an OCL-like way and the definition of iterators as higher-order functions that have a body operation as first argument. For instance, the skeleton constructor for the *select* iterator is defined as follows:

  op _->_`(_;_;_`) : Collection{X} Fun{X} BoolBody{X} ParameterList Collection{X} -> Collection{X} .

  Where the first argument is the collection to be looped, the second argument is an iterator symbol, the third argument is the body operation, the fourth argument is a list of arguments for the body operation, and the fifth argument is the proper collection that is looped. The fifth argument is needed due to the free side-effect that characterizes Maude and is useful when the collection must be navigated in the

body operation. When the iterator is processed, if this argument is not added, the recursion mechanism consumes the elements of the collection, and queries over the whole collection would not be complete.

To invoke the select iterator over a set of integers with the body isOdd we use:
Set{1, 2, 3, 4, 5, 6} -> select(isOdd ; empty-params ; empty-set) .

– Iterator axiomatic specification. It defines the operational semantics of iterator operations independently of body operations. This fact permits the reuse of the algebraic specification of iterator operations simulating them as higher-order functions. Three axioms constitute the algebraic specification of the *select* operator for sets (as shown in Maude notation in Table 6). These are the arguments of select: BB is a variable that contains the boolean body, PL is a parameter list for the body operator, and Col is the original set. The first axiom considers the recursion case where there is more than one element in the set. If the body function validates to a true value, the element is added to the resulting set. Finally, the recursion over the rest of the elements continues. The second axiom considers the recursion case when only one element remains in the set so that the recursive trail ends. The third axiom considers the case where the set is empty.

When the select operation is invoked, Maude returns the following result:
reduce in SOUP : Set{1,2,3,4,5,6}->select(isOdd ; empty-params ; empty-set) .
rewrites: 33 in -285521343301ms cpu (0ms real) (~ rewrites/second)
result Set{RAT}: Set{1,3,5}

```
eq Set{ N , M } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } -> including ( ( Set{ M } -> select ( BB ; PL ; Col )) ) -> flatten
    else Set{ M } -> select ( BB ; PL ; Col ) fi .
eq Set{ N } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } else empty-set fi .
eq empty-set -> select ( BB ; PL ; Col ) = empty-set .
```

Table 6. Axiomatic specification of the select operation for sets.

4 Algebraic specification of metamodels and models

The advantage of OCL is that user-defined types can be used in expressions to perform queries on software artefacts (namely models). User-defined types are the types that can be used in a model: classes, associations, enumerations, and so on. One of the keys to success in the use of the OCL algebraic specification is the integration with an industrial modelling environment. In this way, OCL expressions can be evaluated in a graphical model without having to prepare the information in a specific format manually.

In our case, we have chosen the Eclipse Modeling Framework (EMF). EMF is a modeling environment that is plugged into the Eclipse platform and provides a sort of implementation of the MOF. It brings code generation capabilities: for UML-like models, it generates the final structural code of the application; for metamodels, it generates a default tree-like editor that permits the definition of domain-specific models and the validation of the corresponding metamodel. EMF enables the automatic importation of software artefacts from heterogeneous data sources: UML

models (by means of visual modeling environments), relational schemas of any relational database management system (through the Rational Rose tool), and XML schemas. Therefore, EMF has become an industrial framework for MDA.

To perform OCL queries over EMF software artefacts, two types of projection mechanisms have been specified. On the one hand, we have defined a projection mechanism that obtains the algebraic specification[3] that corresponds to a specific Ecore model automatically, by applying generative programming techniques. On the other hand, another projection mechanism permits us to project an Ecore model instance as a term of the algebra that corresponds to the Ecore model.

As shown in Fig. 2, an Ecore model may represent either a metamodel at the M2-layer (for instance, the UML metamodel) or a model at the M1-layer (for instance, a UML model). Similarly, an Ecore model instance may represent either a model that conforms to a metamodel at the M1-layer (for instance, a UML model) or a model instance at the M0-layer (for instance, the instances of a UML model). From now on, the ocl support is explained by using the example of the UML model, although it is exactly the same to define OCL queries over Ecore metamodels.

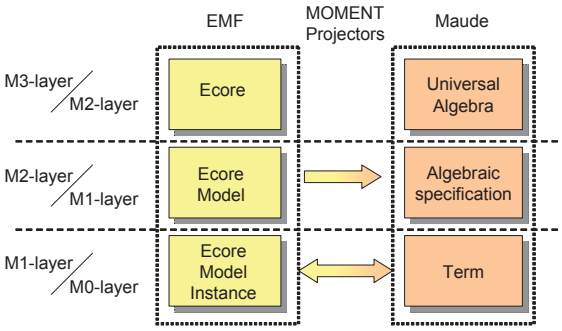


Fig. 2. EMF and Maude conceptual integration

4.1. A metamodel as an algebraic specification

To query a metamodel, it must be represented as an algebraic specification in Maude. This specification is split in two parts: the generic OCL specification that is defined in the parameterized module OCL-SUPPORT(X::TRIV) and the user-defined types in the model.

[3] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel, they only permit the representation of information for model management issues.

4.1.1 The algebraic signature of a metamodel

A metamodel in EMF is an Ecore model, which is mainly constituted by EClass instances (informally called classes) that are related to each other by means of inheritance relationships and EReference instances (informally called references in Ecore and associations in UML). They are used to generate an algebraic specification that is used as an actual parameter for the *OCL-SUPPORT(X::TRIV)* module. Informally, the following tasks are involved in the generation of Maude code from a model:

- Signature generation: An Ecore class is constituted by attributes and references. This information is used to generate a sort that represents the collection of instances of this class and a constructor, whose arguments are: an internal identifier, a group of arguments that represent the attributes (basic data types) and a group of identifier collections (representing references). For instance, the *Table* class in Fig. 1 is used to generate the following Maude code:
  sort Table .
  op *`(Table_____`) : Qid String String OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> rdbms-Table [**ctor**] .*
  where the first argument is the internal identifier of the instance, the second argument is the inherited *name* attribute, the third argument is the inherited *kind* attribute, the fourth argument is the *key* reference (UML role), the fifth argument is the *foreignKey* reference, the sixth argument is the *column* reference and the seventh argument is the *schema* reference. This template is only applied to classes that are not abstract. When a class is defined as abstract, the code only contains the declaration of the code and no constructor is generated, indicating that this class cannot be instantiated.
- Kind structure generation: class inheritance is represented by means of subsort relationships in Maude. In the example, the *Table* class is a specialization of the RModelElement class. In the Maude specification, the sort *Table* is defined as subsort of the RModelElement sort.
- Kind unification: all the sorts that are generated in an algebraic specification are encapsulated within the same kind, by adding a root supersort, called *rdbmsNode* (where *rdbms* is the name of the package that contains the model definition). This sort permits the definition of the view between the model algebraic specification and the formal parameter of the *OCL-SUPPORT(X::TRIV)* module. All the sorts that have no supersort must be related to root supersort by means of a subsort relationship. This allows all the terms that represent instances of model classes to belong to this sort, thereby becoming elements of a collection.
- Signature generation: the generated elements applying the previous templates are encapsulated within a module. The automatically generated module for the example model is called *sigrdbms*.
- View generation: to use the signature that is generated for a model as actual parameter for the module *OCL-SUPPORT(X::TRIV)*, a view is defined containing the mapping between the *Elt* sort of the *TRIV* theory and the generated root sort for the model. The generated view for the model in the example is as follows:
  view rdbms from TRIV to sigrdbms is sort Elt to rdbmsNode . endv

4.1.2 The algebraic specification of a metamodel

A model that conforms a metamodel is represented as a set of instances of the classes that constitute the metamodel. To define models as sets, the signature that is obtained from the definition of the metamodel is used as actual parameter for the parameterized module that contains the generic algebraic specification of OCL by means of the generated view.

The instantiation of the *OCL-SUPPORT(X::TRIV)* module is contained in another module, called *sprdbms* in the example. This module also contains the operations that are needed to navigate a model by means of either the attributes or the roles of a class.

The operation to navigate a reference (UML role) has the following skeleton:

op _::_`(_`) : X$Elt Fun{X} Collection{X} -> Collection{X} [ctor] .

where *X$Elt* represents an element of the model (a class instance), *Fun{X}* represents the symbol of the role, the Collection{X} argument represents the model to be navigated, and the Collection{X} return parameter represents the collection of class instances that is obtained by means of the navigation operation. For example, to navigate through the reference *column* of a *Table* class instance, called *t*, of the a relational schema, called *RdbmsSchema*, we use the following expression:

t :: column (RdbmsSchema)

The operation to obtain the value of an attribute has the following skeletons, depending on the attribute type:

*op _::_ : X$Elt YFun{X} -> Y [ctor] . (w*here Y = {Int, Float, Bool, String, Qid})

For example, to obtain the value of the *name* attribute of a *Table* instance, called *t*, we use the following expression:

t :: name

Finally, the module *sprdbms* contains all the operations that are needed to define a relational schema (constructors to define collections, to define basic data type values and user-defined types) and to apply OCL queries to any relational model (collection operations, iterators, and user-defined navigation operations). Therefore, the membership equational theory that permits the query of the instances of a specific model is defined by the tuple (S_{OCL}, $OP_{OCL} \cup OP_{QUERY}$, E_{OCL}), where:

$S_{OCL} = S_{SimpleDataType} \cup S_{Collections} \cup S_{CollectionOperations} \cup S_{Iterators} \cup S_{UserDefinedClasses}$

$OP_{OCL} = OP_{SimpleDataTypes} \cup OP_{Collections} \cup OP_{UserDefinedTypes}$

$OP_{QUERY} = OP_{CollectionOperations} \cup OP_{Iterators} \cup OP_{UserDefinedNavigationOperations}$

$E_{OCL} = E_{CollectionOperators} \cup E_{Iterators} \cup E_{UserDefinedNavigationOperations}$

4.2. A relational model as an algebraic term

In this paper, OCL queries are performed over models. The definition of a model can be graphically performed by means of EMF. Then, it can be automatically serialized to a term that represents a set of instances of the classes that constitute the model. The serialization uses the algebraic specification that is generated for a model as mentioned above.

For example, a relational schema that conforms to the Rdbms metamodel (shown in Fig. 3) is serialized as a set (shown in Fig. 4), where all the elements are instances of the classes of the metamodel.

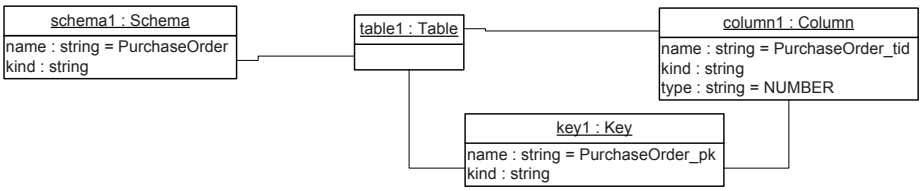


Fig. 3. Object diagram that represents a relational schema that conforms to the Rdbms metamodel defined as a UML model in Fig. 1

```
Set {
    (rdbms-Schema 'schema1 "purchaseOrder" "" OrderedSet { 'table1 } ),
    (rdbms-Table 'table1 "PurchaseOrder" "" OrderedSet { 'key1 } empty-orderedset OrderedSet {
    'column1 } OrderedSet {'schema1 } ),
    (rdbms-Key 'key1 "PurchaseOrder_pk" "" OrderedSet {'table1 } OrderedSet { 'column1 } ),
    (rdbms-Column 'column1 "PurchaseOrder_tid" "" "NUMBER" empty-orderedset OrderedSet { 'key1 }
    OrderedSet {'table1 } )
}
```

Fig. 4. Algebraic representation of the relational schema shown in Fig. 3.

The internal structure of a term is transparent to the user of the algebraic specification due to the navigation operations, which permit the user to navigate in an OCL-like way throughout the roles and attributes of the objects of the model instance.

Therefore, the invariant in Section 2 can be written by defining the body expression of the *forAll* iterator as a body operation. The body operation *inv::body0* defines the body expression of the *forAll* iterator, indicating that the types of the columns of a specific foreign key must be equal to the types of the corresponding columns of the referred primary key. The body operation *inv1* defines the body of the invariant, checking the size of the column collections that belong to the foreign key and to the referred primary key and invoking the forAll operation. The code generated for both body expressions is as follows:

```
var column1 : rdbms-Column . var self : rdbmsNode .

op inv::body0 : -> BoolBody{rdbms} [ctor].
ceq column1 :: inv::body0 ( ? self ; rdbmsModel ) =
    (((((self :: refersTo ( rdbmsModel ) :: column ( rdbmsModel )) -> asSequence) -> at(
        ((self :: column ( rdbmsModel )) -> indexOf(column1))
    )) :: type) == (column1 :: type))
if column1 :: rdbms-Column .
eq column1 :: inv::body0 ( PL ; rdbmsModel ) = false [owise].

op inv1 : -> BoolBody{rdbms} [ctor] .
ceq self :: inv1 ( PL ; rdbmsModel ) =
    (if (((self :: column ( rdbmsModel )) -> size) == ((self :: refersTo ( rdbmsModel ) :: column (
    rdbmsModel )) -> size)) then
        ((self :: column ( rdbmsModel )) -> forAll (inv::body0 ; ? self ; rdbmsModel ))
    else
        false
    fi)
if self :: rdbms-ForeignKey .
eq self :: inv1 ( PL ; rdbmsModel ) = false [owise] .
```

The invariant is checked by selecting the instances of the context class and by applying the invariant body by means of a forAll iterator as follows:
red *rdbms-instance ->* ***select*** *(oclIsTypeOf ; ? "ForeignKey" ; rdbms-instance) ->* ***forAll****(inv1 ; empty-params ; rdbms-instance) .*

5 Related works

Although OCL is not as well supported as UML in some CASE tools, there is a growing interest in providing support for OCL in order to achieve different goals. In [13], several tools that support OCL are studied. Taking them and others into account, some technological examples, which are classified by their main goal, are provided:

- Model transformation: MOMENT, ATL, YATL.
- Model verification: the KeY System.
- Requirements validation: ITP/OCL, the USE tool, the Dresden OCL Toolkit, the Kent OCL tool.
- Code generation (also for requirements validation): Octopus, OCLE.
- OCL Testing: HOL/OCL.

Nevertheless, only a few of them rely on formal methods to provide support for the operational semantics of OCL, and even fewer tools are integrated in (commercial) CASE tools. We focus on some tools that rely on formal methods in this section.

The USE tool [14] provides interactive validation of OCL constraints over a model. This tool reads the input model and the OCL constraints from textual resources, supporting class diagrams, object diagrams and sequence diagrams. Afterwards, objects and links can be graphically created to define a snapshot of a running system. This tool has been extended for the automatic generation of test cases and validation cases [15].

The KeY system [16] provides functionality for formal specification and deductive verification within a commercial CASE tool (Together Control Center). In this approach, the user defines a software artefact in UML that can be annotated with OCL constraints. The OCL constraints are translated into formulas of JavaDL (a dynamic logic for Java [17] that can be reduced by means of an interactive theorem prover.

The ITP/OCL tool [18] provides automatic validation of UML static class diagrams with respect to OCL constraints. It provides an algebraic OCL specification using Maude, where UML class diagrams and object diagrams are formalized by means of algebraic specifications in membership equational logic and where OCL constraints are defined as formulas in membership equational logic theories. A graphical front-end is being developed for the ITP/OCL tool, which permits the definition of class diagrams and the definition of correct object diagrams.

In these last three discussed approaches, only UML diagrams are considered for validating OCL expressions. In the MOMENT-OCL specification, OCL queries can be automatically applied either to metamodels or to models that may be defined in EMF by making use of the Maude parameterization mechanism, following a more automated model-driven oriented approach. In our approach, while Maude is used to execute OCL expressions and to guarantee formal features of the expressions, the

OCL expressions can be applied to graphical model-based software artefacts through the EMF.

6 Conclusions and Further Work

OCL is becoming a de-facto standard for defining constraints and queries in the Model-Driven Engineering field. The number of tools that provide support for this language is growing, and although the operational semantics of OCL is said to be formal, only a few tools rely on formal methods to define its operational semantics.

In this paper, we present an algebraic specification of part of the operational semantics of OCL 2.0 from an implementation point of view. This specification takes advantage of the parameterization mechanism and the attribute theory of Maude and provides a way to simulate collection operations as higher-order functions for the sake of reuse.

This specification is used to perform model queries in the EMF and to represent EMF software artefacts as algebraic specifications or as terms that can be manipulated by means of model management operators [19]. The OCL specification has been developed generically so that it can be used for any kind of metamodels, models or model instances. These operators use OCL queries to specify model transformations that can be run in Maude as detailed in [5]. Therefore, not only can OCL be studied in an algebraic setting, it can also be applied in the well-known modelling environment EMF.

As the syntax that was obtained for OCL in Maude is not complete standard-compliant, we are developing an OCL compiler to Maude code. In this way, the gap between OCL-practitioners and the algebraic specification will be transparent. Further work consists in exploiting the formal features of the OCL specification from a more-theoretical point of view and its application to real case studies.

References

1. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture--Practice and Promise. (2003)
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons (2004)
3. OMG: Meta Object Facility (MOF) 2.0 Core Specification, ptc/04-10-15. (2004)
4. OMG: MOF 2.0 QVT final adopted specification (ptc/05-11-01). (2005)
5. Boronat, A., Carsí, J.A., Ramos, I.: Algebraic Specification of a Model Transformation Engine. In: Proceedings of Fundamental Approaches to Software Engineering, FASE'06, LNCS, Vol. 3922. Springer Vienna, Austria (2006)
6. Bézivin, J., Dupe, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: Proceedings of OOPSLA 2003 Workshop, Anaheim, California (2003)
7. The Model Transformation Framework: http://www.alphaworks.ibm.com/tech/mtf
8. Warmer, J., Kleppe, A.: The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA. Addison-Wesley (2004)

9. Chiorean, D., Bortes, M., Corutiu, D.: Proposals for a Widespread Use of OCL. In: Proceedings of Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Technical Report LGL-REPORT-2005-001, Vol. EPFL, Montego Bay, Jamaica (2005) 68--82
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285** (2002) 187-243
11. The MOMENT Project: http://moment.dsic.upv.es
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.2 manual and examples. (2005) http://maude.cs.uiuc.edu/maude2-manual/
13. Toval, A., Requena, V., Fernández, J.L.: Emerging OCL tools. Software and System Modeling **2** (2003) 248-261
14. Richters, M.: The USE tool: A UML-based Specification Environment. (2001) http://www.db.informatik.uni-bremen.de/projects/USE/
15. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. In: Proceedings of UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference. Springer, San Francisco, CA, USA (2003) 265-279
16. Ahrendt, W., Baar, T., Beckert, B., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Schmitt, P.H.: The KeY System: Integrating Object-Oriented Design and Formal Methods. In: Proceedings of Fundamental Approaches to Software Engineering. 5th International Conference, FASE 20022306. Springer, Grenoble, France (2002) 327-330
17. Beckert, B.: A Dynamic Logic for Java Card. In: Proceedings of Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France (2000)
18. Egea, M., Clavel, M.: The ITP/OCL tool. (2006) http://maude.sip.ucm.es/itp/ocl/
19. Boronat, A., Carsí, J.A., Ramos, I.: Automatic Support for Traceability in a Generic Model Management Framework. In: Proceedings of Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, LNCS, Vol. 3748. Springer Nuremberg, Germany (2005) 316-330

An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework ♦

Artur Boronat, Joaquín Oriente, Abel Gómez, Isidro Ramos, José Á. Carsí

Department of Information Systems and Computation
Technical University of Valencia
C/Camí de Vera s/n
46022 Valencia-Spain
{aboronat | joriente | agomez | iramos | pcarsi}@dsic.upv.es

Abstract. In the Model-Driven Architecture initiative, software artefacts are represented by means of models that can be manipulated. Such manipulations can be performed by means of transformations and queries. The standard Query/Views/Transformations and the standard language OCL are becoming suitable languages for these purposes. This paper presents an algebraic specification of the operational semantics of part of the OCL 2.0 standard, focusing on queries. This algebraic specification of OCL can be used within the Eclipse Modeling Framework to represent models in an algebraic setting and to perform queries or transformations over software artefacts that can be represented as models: model instances, models, metamodels, etc. In addition, a prototype for executing such OCL queries and invariants over EMF models is presented. This prototype provides a compiler of the OCL standard language that targets an algebraic specification of OCL, which is run on the term rewriting system Maude.

Keywords: MDA, OCL queries and invariants, metamodeling, algebraic specification.

1 Introduction

Model-Driven Development is a field in Software Engineering that, for several years, has represented software artefacts as models in order to improve productivity, quality, and economic incomes. Models provide a more abstract description of a software artefact than the final code of the application. A model can be built by defining concepts and relationships. The set of primitives that permit the definition of these elements constitutes what is called the metamodel of the model.

Interest in this field has grown in software development companies due to several factors. Previous experiences with Model Integrated Computing [1] (where embedded systems are designed and tested by means of models before generating them automatically) have shown that costs decrease in the development process. The consolidation of UML as a design language for software engineers has contributed to

♦ This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J. A. An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework. Proc. of the Second European Conference on Model Driven Architecture. 2006.

software Model-Driven Development by means of several CASE tools that permit the definition of UML models and automated code generation. The emergence of important model-driven initiatives such as the Model-Driven Architecture [2], which is supported by OMG, and the Software Factories [3], which is supported by Microsoft, ensures a model-driven technology stock for the near future.

Model-Driven Development has evolved into the Model-Driven Engineering field, where not only design and code generation tasks are involved, but also traceability, model management, metamodeling issues, model interchange and persistence, etc. To fulfil these tasks, model transformations and model queries are relevant tasks that must be solved. In the MDA context, they are dealt with from an open-standard point of view. The standard Meta-Object Facility (MOF) [4] provides a way to define metamodels. The standard proposal Query/Views/Transformations (QVT) [5] will provide support for both transformations and queries. While model transformation technology is being developed [6-8], the Object Constraint Language (OCL) remains as the best choice for queries.

OCL [9] is a textual language that is defined as a standard "add-on" to the UML standard. It is used to define constraints and queries on UML models, allowing the definition of more precise and more useful models. It can also be used to provide support for metamodeling (MOF-based and Domain Specific Metamodeling), model transformation, Aspect-Oriented Modeling, support for model testing and simulation, ontology development and validation for the Semantic Web, among others. Despite its many advantages, while there is wide acceptance for UML design in CASE tools, OCL lacks a well-suited technological support.

In this paper, we present an algebraic specification of generic OCL queries, by using Maude [10], that can be used in a MOF-like industrial tool. The OCL algebraic specification permits the study of formal features and their proofs, and the execution of OCL expressions as well. This algebraic specification has been developed in the MOMENT framework (MOdel manageMENT) [11], which provides a set of generic operators to deal with models. The MOMENT operators use OCL queries to perform model queries and transformations.

The structure of the paper is as follows: Section 2 provides an example; Section 3 describes the algebraic specification of OCL, indicating the support for basic data types and collection types, and the support for collection operations; Section 4 presents the integration of the algebraic specification of OCL within an industrial modelling framework; Section 5 provides the architecture of the prototype; Section 6 presents some related works; Section 7 provides some conclusions and ongoing work.

2 The Coach Company Example

The Meta-Object Facility standard (MOF) [4] provides a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata-driven systems. The main achievement of this standard is the definition of a common terminology in the Model-Driven Architecture initiative, which can be used conceptually in other model-driven approaches.

As an example, in this paper, we have modelled a simple coach company in UML. In this design, a coach has a specific number of seats and can be used for regular trips or for private trips. In regular trips, the tickets are bought individually. In private trips, the whole coach is rented for a trip. The model is shown in UML notation in Fig. 1. The example provides a specific UML model, and the queries are applied to its instances. The OCL-like specification that is presented can also be used for queries over any software artefact that might be defined following the MOF conceptual framework: metamodels, regular models, and instances of models.

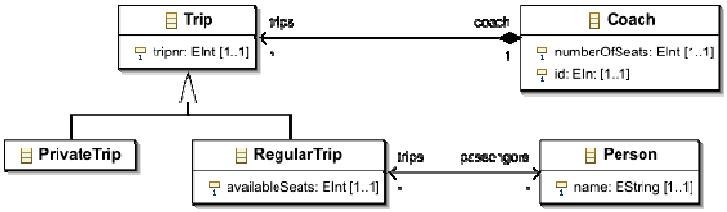


Fig. 1. Coach company model

OCL queries[1] permit a more precise definition of the above model by adding constraints. For instance, we can indicate that overbooking is not allowed in a regular trip by means of the following invariant:

```
context Coach:
inv: self.trips -> select( t:Trip | t.oclIsType(RegularTrip))
        -> forAll(r:Trip | r.oclAsType(RegularTrip).passengers -> size()
                <= r.coach.numberOfSeats -> sum())
```

3 Algebraic specification of generic[2] OCL queries

In this section, we describe the parameterized algebraic specification of OCL that permits the query of either metamodels or UML models. The Maude term rewriting system [10] has been used for this purpose. Maude provides an algebraic specification language that belongs to the OBJ family[3]. Its equational deduction mechanism animates the OCL algebraic specification over a specific model instance, providing the operational semantics for OCL expressions. We have developed a plug-in that embeds the Maude environment into the Eclipse framework so that we can use it for our purposes.

[1] We consider that an invariant is built on an OCL query that returns a Boolean value. Thus, although we talk about invariants, we are also using OCL queries.

[2] In this work, OCL genericity refers to the possibility of reusing the OCL specification for any software artefact that can be represented as a model, including metamodels.

[3] In this paper, we assume some basic knowledge about algebraic specifications and OBJ-like notation. We refer to [12] for more details.

3.1 Overview of the parameterized OCL algebraic specification

In Maude, functional modules describe data types and operations on them by means of membership equational theories. Mathematically, such a theory can be described as a pair (Σ, E ∪A), where: Σ is the signature that specifies the type structure (sorts, subsorts, kinds, and overloaded operators); E is the collection of equations and memberships declared in the functional module; and A is the collection of equational attributes (associativity, commutativity, and so on) that are declared for the different operators. Computation is the form of equational deduction in which equations are used from left to right as simplification rules, with the rules being Church-Rosser and terminating.

OCL collection types and their operations have been defined in a parameterized algebraic specification, called *OCL-SUPPORT{X :: TRIV}*. Fig. 2 shows the elements involved in the parameter passing mechanism diagram. *TRIV* is the algebraic specification of the formal parameter, which is called theory in Maude.

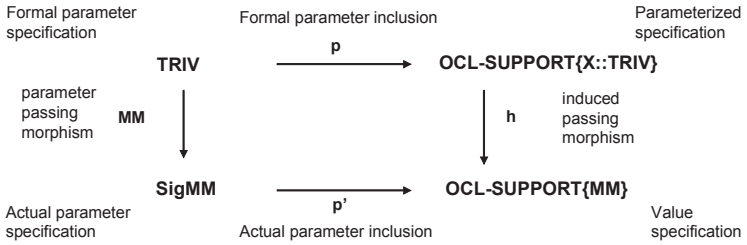


Fig. 2. The parameter passing diagram for the *OCL-SUPPORT{X :: TRIV}* parameterized module.

SigMM is an algebraic specification that is automatically obtained from a specific metamodel automatically. The *SigMM* specification constitutes the actual parameter for the *OCL-SUPPORT{X :: TRIV}* module and provides a constructor for each type that is defined in the metamodel and an inheritance hierarchy among the types that appear in the metamodel. The *MM* view is the morphism that relates the elements of the *TRIV* formal parameter to the elements of the *SigMM* actual parameter.

OCL collection types and related operations have been generically specified in the parameterized module *OCL-SUPPORT{X :: TRIV}*, where the formal parameter *X* has the trivial theory as type. The trivial theory only contains a sort *Elt* (referred to as *X$Elt* in the OCL specification) that represents the sort of elements that can be contained in an OCL collection. This sort represents the *OCLAny* type of the standard OCL specification. The *OCL-SUPPORT{X::TRIV}* module imports the basic data types and provides the constructors that are needed to define collections of elements. It provides collection operations as well.

In Fig. 2, p and p' are inclusion morphisms that indicate that the formal parameter specification is included in the parameterized specification, and that the actual parameter specification is included in the value specification, respectively. The h morphism is the induced passing morphism that relates the elements of the parameterized module to the elements of the *OCL-SUPPORT{MM}* value specification by using the *MM* parameter passing morphism.

3.2 Algebraic Specification of OCL Types

Types in OCL are divided into basic data types, collection types, and user-defined types. In this section, the algebraic support for the first two kinds of types is presented.

3.2.1 Basic Data Types

In OCL, there are four basic data types that have a direct correspondence to Maude basic data types. In Table 1, we show the correspondences between OCL 2.0 and the Maude data-type system and their corresponding primitives. In the table, when the operations have different symbols in OCL and Maude, we indicate the Maude symbol in brackets.

OCL 2.0	Maude	Common operators
Boolean	Bool	or, and, xor, not, = (==), <> (=/=), implies, if-then-else-endif (if-then-else-fi)
Integer	Int	= (==), <> (=/=), <, <=, >, >=, +, -, *, / (quo), mod (rem), abs (abs), max, min
Real	Float	/, round (ceiling), floor
String	String	concat (+), size (length), substring(substr), = (==), <> (=/=)

Table 1. OCL and Maude data-type correspondences

3.2.2 Collection Types

OCL provides four specific collection types that are defined as follows:

- A Set is a collection that contains instances of a valid OCL type, where order is not relevant and duplicate elements are not allowed.
- An OrderedSet is a set whose elements are ordered.
- A Bag is a collection that may contain duplicate elements. Elements in a bag are not ordered.
- A Sequence is a bag whose elements are ordered.

To take into account the uniqueness and order features of an OCL collection, we introduce two intermediate sorts and their constructors (shown in Table 2): *Magma{X}* and *OrderedMagma{X}*. Basically, we define the sort *Magma{X}* as the sort of the term that represents a group of elements that are not ordered by means of the association and the commutativity attributes. The constructor for this sort has the symbol "," and is associative and commutative. Thus, working with integers, "*1, 2, 3*" is the term that represents a valid *Magma{Int}*. In addition, we can state that "*1,2,3*" and "*3,2,1*" represent the same group of elements modulo the commutative and associative attributes.

Instead, the constructor of the sort *OrderedMagma{X}* does not have the commutativity property, producing terms that represent ordered concatenations of elements. The constructor for this sort has "::" as symbol and permits building ordered groups of elements by using the common syntax for lists in functional programming. Thus, the term "*1 :: 2 :: 3*" represents a valid ordered magma of integers, and "*1 :: 2 :: 3*" is different from "*3 :: 2 :: 1*" because the constructor "::" is not commutative.

```
1.  sort Magma{X} OrderedMagma{X} .
2.  subsort X$Elt < Magma{X} OrderedMagma{X} .
3.  sorts Collection{X} Set{X} OrderedSet{X} .
4.  subsort Collection{X} < X$Elt .
5.  subsorts Set{X} OrderedSet{X} < Collection{X} .
6.  op _,_ : Magma{X} Magma{X} -> Magma{X} [assoc comm ctor] .
7. op _::_ : Magma{X} Magma{X} -> Magma{X} [assoc ctor] .
8.  op Set{_} : Magma{X} -> Set{X} [ctor] .
9. op empty-set : -> Set{X} [ctor] .
10.  op OrderedSet{_} : OrderedMagma{X} -> OrderedSet{X} [ctor] .
11. op empty-orderedset : -> OrderedSet{X} [ctor] .
```

Table 2. Specification of groups of elements

Terms of the sort *Magma{X}* are used to define sets (line 8), while terms of the sort *OrderedMagma{X}* are used in ordered sets (line 10). In Table 2, we show the Maude code that specifies the Set and OrderedSet types. In our specification, collections of collections are allowed by indicating that one collection can be an element of another collection (line 4). The sort *Collection{X}* can be considered as an abstract concept on the grounds that there is no specific constructor for it. Each collection has a constant constructor that defines an empty collection (lines 9, 11). The types Bag and Sequence have also been specified, similarly to the Set and OrderedSet types, respectively. In this specification, the uniqueness property of both the collection Set and the collection OrderedSet is checked in the operations that join two collections: *union*, *intersection* and *including* for Set, and *union*, *append*, *prepend*, *insertAt* and *including* for OrderedSet.

A view has been defined for each Maude simple data type in order to deal with collections of simple data types. For instance, to deal with collections of integers, the following view is defined: *view Int from TRIV to INT is sort Elt to Int . endv*

This view is used to instantiate the OCL-SUPPORT{X} module as OCL-SUPPORT{Int}. This way, the following example is a valid collection of integers:
OrderedSet{ Set{1, 2, 3} :: Bag{1, 2, 3, 3} :: Sequence{3 :: 3 :: 2 :: 1}}

3.3 Loop Operations or Iterators

Two kinds of operations on collection types can be distinguished in OCL 2.0: regular operations and loop operations or iterators. Regular operations provide common functionality over collections. Loop operations or iterators permit looping over the elements in a collection while performing a specific action. In this paper, we focus on the second type of operations.

Every loop operation has an OCL expression as parameter. This is called the body, or body parameter, of the operation. As a guiding example, we use a standard OCL expression that permits obtaining the odd numbers from a set of integers:

Set{1,2,3,4,5,6} -> select(i | i.mod(2) <> 0)

In this expression, *select* is the iterator operation and the expression *(i | i.mod(2) <> 0)* is the body. Both iterator operations and body expressions are considered in the algebraic specification separately. This separation is needed to simulate higher-order functions in Maude by considering body functions as terms that can be passed as arguments to iterator operations.

Using the example of the selection of odd numbers from an integer set, we study first how to specify the body of the select expression *i | i.mod(2) <> 0.* Expression bodies can be evaluated to several types depending on the kind of operator in which they are used. For instance, the body expression of a select evaluates to a boolean value. Depending on the return type of the body expression, a symbol is associated to it indicating the name of the body expression. For the example, we obtain:

op isOdd : -> BoolBody{Int} [ctor] .

The body expression is built by using the following operation:

op _::_`(_;_`) : Magma{X} BoolBody{X} ParameterList Collection{X} -> Bool .

where the first argument is a term that represents a magma of elements, the second argument is the corresponding body symbol, the third argument is a variant list of parameters that can be empty, and the fourth argument is the whole initial collection to which the first argument belongs. To define a body function, the axioms must be provided by the user in Maude notation. For the example, we define the following equation:

var intN : Int . var intCol : Collection{Int} . var PL : ParameterList .
eq intN :: isOdd (PL ; intCol) = ((intN rem 2) =/= 0) .

Once the body expression has been defined, we provide an algebraic specification of the operational semantics of the select operation for sets. The different collection operations have been defined as function symbols (terms of the sorts that are shown in Table 3), depending on the return type of each operation. For instance, the select operation, which returns a collection of elements, is defined as follows:

op select : -> Fun{X} [ctor] .

	Return type	Collection operator symbols					Iterator symbols
		Collection	Set	OrderedSet	Bag	Sequence	Collection
Fun{X}	Collection	union, flatten, including, excluding, iterate	--, inter-secti-on	--, insertAt, append, prepend	intersection	insertAt, append, prepend	select, reject, any, sortedBy, collect, collectNested, iterate
EltFun{X}	Element			first, last, at		first, last, at	
BoolFun{X}	Boolean value	includes, includesAll, excludes, excludesAll, isEmtpy, notEmpty					one, forAll, forAll2[4], exists, isUnique
IntFun{X}	Integer value	count, size, sum, product		indexOf		indexOf	

Table 3. OCL collection operations that have been specified

The operational semantics of iterator operations is defined independently of body operations. This fact permits the reuse of the algebraic specification of iterator operations simulating them as higher-order functions. Three axioms constitute the algebraic specification of the *select* operator for sets (as shown in Maude notation in Table 4). These are the arguments of select: BB is a variable that contains the boolean body expression, PL is a parameter list for the body operator, and Col is the original set. The first axiom considers the recursion case where there is more than one element in the set. If the body function validates to a true value, the element is added to the

[4] The forAll2 operation has been included to provide support when two iterators are being used in the forAll operation.

resulting set. Finally, the recursion over the rest of the elements continues. The second axiom considers the recursion case when only one element remains in the set so that the recursive trail ends. The third axiom considers the case where the set is empty.

To invoke an iterator in an OCL-like way, the following operation is used:

op _->_`(_;_;_`) : Collection{X} Fun{X} BoolBody{X} ParameterList Collection{X} -> Collection{X} .

where the first argument is the collection to be looped, the second argument is an iterator symbol, the third argument is the body operation, the fourth argument is a list of arguments for the body operation, and the fifth argument is the proper collection that is looped. The fifth argument is useful when the collection must be navigated in the body operation. When the iterator is processed, if this argument is not added, the recursion mechanism consumes the elements of the collection, and queries over the whole collection would not be complete. To invoke the select iterator over a set of integers with the body isOdd we use: *Set{1, 2, 3, 4, 5, 6} -> select(isOdd ; empty-params ; empty-set)* .

```
eq Set{ N , M } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } -> including ( ( Set{ M } -> select ( BB ; PL ; Col )) ) -> flatten
    else Set{ M } -> select ( BB ; PL ; Col ) fi .
eq Set{ N } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } else empty-set fi .
eq empty-set -> select ( BB ; PL ; Col ) = empty-set .
```

Table 4. Axiomatic specification of the select operation for sets.

4 Algebraic specification of metamodels and models

The advantage of OCL is that user-defined types can be used in expressions to perform queries on software artefacts (namely models). User-defined types are the types that can be used in a model: classes, associations, enumerations, and so on. One of the keys to success in the use of the OCL algebraic specification is the integration with an industrial modelling environment. In this way, OCL expressions can be evaluated in a graphical model without having to prepare the information in a specific format manually.

In our case, we have chosen the Eclipse Modeling Framework (EMF) [13]. EMF is a modeling environment that is plugged into the Eclipse platform and provides a sort of implementation of the MOF. It brings code generation capabilities: for UML-like models, it generates the final structural code of the application; for metamodels, it generates a default tree-like editor that permits the definition of domain-specific models and the validation of the corresponding metamodel. EMF enables the automatic importation of software artefacts from heterogeneous data sources: UML models (by means of visual modeling environments), relational schemas of any relational database management system (through the Rational Rose tool), and XML schemas. Moreover, third-party researchers and developers are bringing new tools to work on ontologies through EMF and graphical Domain Specific Languages. Therefore, EMF has become an industrial framework for MDA.

To perform OCL queries over EMF software artefacts, two types of projection mechanisms have been specified. On the one hand, we have defined a projection mechanism that obtains the algebraic specification[5] that corresponds to a specific Ecore model automatically by applying generative programming techniques. On the other hand, another projection mechanism permits us to project an Ecore model instance as a term of the algebra that corresponds to the Ecore model.

As shown in Fig. 3, an Ecore model may represent either a metamodel at the M2-layer (for instance, the UML metamodel) or a model at the M1-layer (for instance, a UML model). Similarly, an Ecore model instance may represent either a model that conforms to a metamodel at the M1-layer (for instance, a UML model) or a model instance at the M0-layer (for instance, the instances of a UML model). From now on, the OCL support is explained by using the example of the UML model, although it would be exactly the same as defining OCL queries over Ecore metamodels.

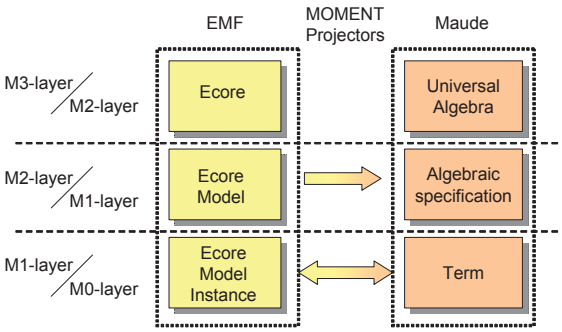


Fig. 3. EMF and Maude conceptual integration

4.1. A model as an algebraic specification

To query a model, it must be represented as an algebraic specification in Maude. This specification is split in two parts: the generic OCL specification that is defined in the parameterized module OCL-SUPPORT{X::TRIV} and the user-defined types in the model.

4.1.1 The algebraic signature of a model (SigMM)

An Ecore model is mainly constituted by EClass instances (informally called classes) that are related to each other by means of inheritance relationships and EReference instances (informally called references in Ecore and associations in UML). They are

[5] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel; they only permit the representation of information for model management issues.

used to generate an algebraic specification that is used as an actual parameter for the *OCL-SUPPORT{X::TRIV}* module. Informally, the following tasks are involved in the generation of Maude code from a model:

- Signature generation: An Ecore class is constituted by attributes and references. This information is used to generate a sort that represents the collection of instances of this class and a constructor, whose arguments are: an internal identifier, a group of arguments that represent the attributes (basic data types) and a group of identifier collections (representing references). For instance, the *RegularTrip* class in Fig. 1 is used to generate the following Maude code:
  sorts RegularTrip .
  op `(RegularTrip_____`) : Qid Int Int OrderedSet {QID} OrderedSet {QID} -> RegularTrip [ctor] .
  where the first argument is the internal identifier of the instance, the second argument is the inherited *tripnr* attribute, the third argument is the *availableSeats* argument, the fourth argument is the inherited *coach* reference (UML role), and the fifth argument is the *passengers* reference (UML role). This template is only applied to specific classes. When a class is defined as abstract, the code only contains the declaration of the code and no constructor is generated, indicating that this class cannot be instantiated.
- Kind structure generation: class inheritance is represented by means of subsort relationships in Maude. In the example, the *RegularTrip* class is a specialization of the Trip class. In the Maude specification, the sort *RegularTrip* is defined as a subsort of the *Trip* sort.
- Kind unification: all the sorts that are generated in an algebraic specification are encapsulated within the same kind, by adding a root supersort, called *tripNode* (where trip is the name of the package that contains the model definition). This sort permits the definition of the view between the model algebraic specification and the formal parameter of the *OCL-SUPPORT{X::TRIV}* module. All the sorts that have no supersort must be related to root supersort by means of a subsort relationship. This allows all the terms that represent instances of model classes to belong to this sort, thereby becoming elements of a collection.
- Signature generation: the generated elements applying the previous templates are encapsulated within a module. The automatically generated module for the example model is called *sigtrip*.
- View generation: to use the signature that is generated for a model as actual parameter for the module *OCL-SUPPORT{X::TRIV}*, a view is defined containing the mapping between the *Elt* sort of the *TRIV* theory and the generated root sort for the model. The generated view for the model in the example is as follows:
  view vtrip from TRIV to sigtrip is sort Elt to tripNode . endv

4.1.2 The algebraic specification of a model

The instance of a model is represented as a set of instances of the classes that constitute the model *MM*. To define model instances as sets, the signature that is obtained from the definition of the model is used as actual parameter for the parameterized module that contains the generic algebraic specification of OCL by means of the generated view.

The instantiated module *OCL-SUPPORT{MM}* is contained in another module, called *sptrip* in the example. This module also contains the operations that are needed to navigate the model by means of either the attributes or the roles of a class.

The operation to navigate a reference (UML role) has the following skeleton:

op _::_`(_`) : X$Elt Fun{X} Collection{X} -> Collection{X} [ctor] .

where *X$Elt* represents an element of the model instance (a class instance), *Fun{X}* represents the symbol of the role, the Collection{X} argument represents the model instance (which is needed to navigate the model), and the Collection{X} return parameter represents the collection of class instances that is obtained by means of the navigation operation. For example, to navigate through the reference coach of a RegularTrip class instance, called rt, of the a model instance, called *ModelInstance*, we use the following expression:

rt :: coach (ModelInstance)

The operation to obtain the value of an attribute has the following skeletons, depending on the attribute type:

*op _::_ : X$Elt YFun{X} -> Y [ctor] . (w*here Y = {Int, Float, Bool, String, Qid})

For example, to obtain the value of the *availableSeats* of a *TripRegular* class instance, called *rt*, we use the following expression:

rt :: availableSeats

Finally, the module *sptrip* contains all the operations that are needed to define an instance of a model (constructors to define collections, to define basic data type values and user-defined types) and to apply OCL queries to instances of any model (collection operations, iterators, and user-defined navigation operations). Therefore, the membership equational theory that permits the query of the instances of a specific model is defined by the tuple (S_{OCL}, $OP_{OCL} \cup OP_{QUERY}$, E_{OCL}), where:

$S_{OCL} = S_{SimpleDataType} \cup S_{Collections} \cup S_{CollectionOperations} \cup S_{Iterators} \cup S_{UserDefinedClasses}$

$OP_{OCL} = OP_{SimpleDataTypes} \cup OP_{Collections} \cup OP_{UserDefinedTypes}$

$OP_{QUERY} = OP_{CollectionOperations} \cup OP_{Iterators} \cup OP_{UserDefinedNavigationOperations}$

$E_{OCL} = E_{CollectionOperators} \cup E_{Iterators} \cup E_{UserDefinedNavigationOperations}$

4.2. An instance of a model as an algebraic term

OCL queries are performed over instances of models. The definition of a model can be graphically performed by means of EMF. Then, it can be automatically serialized to a term that represents a set of instances of the classes that constitute the model. The serialization uses the algebraic specification that is generated for a model as mentioned above.

For example, the instance of the model Trip (shown in Fig. 4) is serialized as a set, where all the elements are instances of the classes of the model, as follows:

Set { (Coach 'coach1 1 10 OrderedSet { 'person1 }),
(RegularTrip 'trip1 1 9 OrderedSet {'coach1 } OrderedSet { 'person1}),
(Person 'person1 "Peter" OrderedSet { 'trip1 }) }

The internal structure of a term is transparent to the user of the algebraic specification due to the navigation operations, which permit the user to navigate in an OCL-like way throughout the roles and attributes of the objects of the model instance.

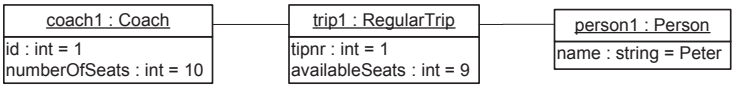


Fig. 4. Object diagram defined as an instance of the model defined in Fig. 1

Therefore, the query that is used in the invariant in Section 2 can be written by defining the body expression of the *forAll* iterator as a body operation. This body operation checks that all the regular trips have a lower number of passengers than the number established by the numberOfSeats attribute of the corresponding Coach instance. The Maude code that is automatically obtained for the body expression of the forAll operator in the example is as follows:

```
var self : trip-Trip . var tripModel : Set{trip} .
op notOverbooked : -> BoolBody{trip} [ctor].
ceq self :: notOverbooked ( PL ; tripModel ) =
    ((((self :: oclAsType ( ? "RegularTrip" ; tripModel )) :: passengers ( tripModel))
    -> size) <= ((self :: coach ( tripModel ) :: numberOfSeats) -> sum))
if self :: trip-Trip .
eq self :: notOverbooked ( PL ; tripModel ) = false [owise].
```

where self is a variable of type Trip and tripModel is a set that represents the model instance to be queried. The query that is used in the body of the invariant can be coded as follows:

```
red self :: trips ( tripModel ) -> select ( oclIsTypeOf ; "RegularTrip" ; tripModel ) -> forAll
(notOverbooked ; empty-params ; tripModel) .
```

where self is a variable of type Coach, tripModel is a variable that contains the model instance to be checked, and the select and the forAll operation provide the body expression of the invariant in Maude code by using the oclIsTypeOf operator and the above body expression notOverbooked.

5 MOMENT-OCL: a prototype for executing algebraic OCL expressions within the Eclipse Modeling Framework

The OCL algebraic specification that has been presented in the paper permits both the representation of models as sets and the use of queries and invariants over them. This permits the use of OCL expressions in algebraic model transformations, such as those presented in [6]. In addition, we have developed a simple OCL editor that permits the evaluation of OCL queries and invariants over EMF models or model instances. In this section, we provide a brief description of the architecture of this prototype, which is called MOMENT-OCL.

Fig. 6 shows the components of the MOMENT-OCL prototype that permit the execution of algebraic OCL expressions over EMF models:

– The *OCL Projector* component is the module that projects the OCL expression to Maude code. It makes use of the *Kent OCL* library [14] to validate the syntax and the semantics of the expression.

  The process of compilation from OCL to Maude follows the typical structure of a language processor. The process is divided in two phases: an initial analysis phase and a second synthesis phase.

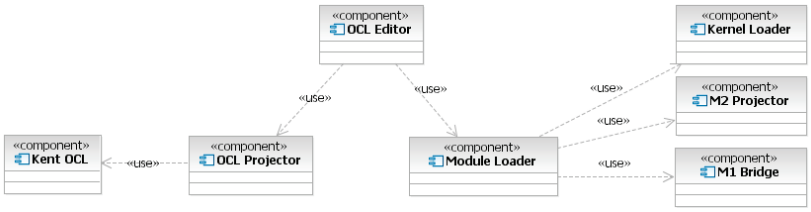


Fig. 5. MOMENT-OCL Architecture

In the first phase, we have reused the OCL support of the Kent Modelling Framework (KMF) [15], which provides lexical, syntactical and semantical analysis of OCL expressions over an EMF model. KMF analyzes an OCL expression, taking into account the semantics of the model, and produces an Abstract Syntax Tree (AST) to represent the data that is needed in the synthesis phase.

In the second phase, once an OCL expression has been analyzed by KMF correctly, the AST is parsed and Maude code for body expressions, queries and invariants are produced in order to evaluate OCL expressions over EMF models in Maude.

- The *Module Loader* component obtains the algebraic specification from a metamodel, by instantiating the OCL-SUPPORT{X::TRIV} module with the signature obtained for a specific metamodel. This algebraic specification is extended with the Maude code obtained from the compilation of OCL expressions by means of the OCL Projector component. The *Module Loader* uses three other components: the M2 Projector, which projects a metamodel *MM* (the Coach model in the example) as the signature *SigMM*; the M1 Bridge, which projects a model (model instance in the example) as a term of the corresponding algebraic specification OCL-SUPPORT{MM}; and the Kernel Loader, which instantiates the parameterized algebraic specification of OCL with the signature SigMM, providing the formal environment where OCL expressions for the model MM can be evaluated.
- The OCL Editor permits the definition of OCL queries and invariants over EMF models and provides syntactical and semantical analyses of the expressions by reusing this functionality from the KMF. It permits the evaluation of queries and invariants. Finally, there is an option to show the generated Maude code which specifies the operational semantics for the OCL expressions for debugging purposes.

  The editor (shown in Fig. 7) consists of a tree viewer, a properties view, and a console to show results. The tree viewer is the main window of the application. The information is shown in a hierarchical structure. The first step is to consider a model and its instance specifying the paths of the respective XMI files, where they are defined. The second step consists in specifying different OCL contexts for the model selected in the first step. Finally, we can define groups of queries or invariants for each context. It is possible to evaluate all the invariants defined in a model or only those defined in a specific context. The queries and invariants have two attributes, a name to identify them, which is useful to visualize the result of a group of invariants, and the expression, which can be written in an OCL editor.

If we consider an invariant or query, we can analyze the expression syntactically and semantically, evaluate it by showing the result, or parse it to Maude code, as indicated in Fig. 7.

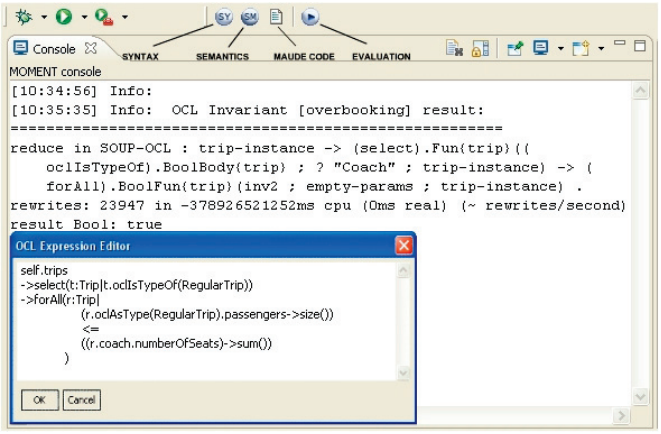


Fig. 6. MOMENT-OCL screenshot

6 Related works

Although OCL is not as well supported as UML in some CASE tools, there is a growing interest in providing support for OCL in order to achieve different goals. In [16], several tools that support OCL are studied. Taking them and others into account, some technological examples, which are classified by their main goal, are provided:

- Model transformation: MOMENT, ATL, YATL.
- Model verification: the KeY System.
- Requirements validation: ITP/OCL, the USE tool, the Dresden OCL Toolkit, the Kent OCL tool.
- Code generation (also for requirements validation): Octopus, OCLE.
- OCL Testing: HOL/OCL.

Nevertheless, only a few of them rely on formal methods to provide support for the operational semantics of OCL, and even fewer tools are integrated in (commercial) CASE tools. We focus on some tools that rely on formal methods in this section.

The KeY system [17] provides functionality for formal specification and deductive verification within a commercial CASE tool (Together Control Center). In this approach, the user defines a software artefact in UML that can be annotated with OCL constraints. The OCL constraints are translated into formulas of JavaDL (a dynamic logic for Java) that can be reduced by means of an interactive theorem prover.

The USE tool [18] provides interactive validation of OCL constraints over a model. This tool reads the input model and the OCL constraints from textual resources, supporting class diagrams, object diagrams and sequence diagrams. Afterwards, objects and links can be graphically created to define a snapshot of a

running system. This tool has been extended for the automatic generation of test cases and validation cases.

The ITP/OCL tool [19] provides automatic validation of UML static class diagrams with respect to OCL constraints. It provides an algebraic OCL specification using Maude, where UML class diagrams and object diagrams are formalized by means of algebraic specifications in membership equational logic and where OCL constraints are defined as formulas in membership equational logic theories. A graphical front-end is being developed for the ITP/OCL tool, which permits the definition of class diagrams and the definition of correct object diagrams.

In these last approaches, only UML diagrams are considered for validating OCL expressions. In the MOMENT-OCL specification, OCL queries can be automatically applied either to metamodels or to models that may be defined in EMF by making use of the Maude parameterization mechanism, following a more automated model-driven oriented approach. In our approach, while Maude is used to execute OCL expressions, the OCL expressions can be applied to graphical model-based software artefacts through the EMF. Whenever EMF, and related support, is used to develop a (domain specific or UML) modelling environment, we can use MOMENT-OCL to provide invariant checking and query evaluation. Thus, our philosophy does not consist in developing a new modelling environment to provide OCL support, we provide it for other existing modelling approaches. Other java-based approaches that integrate OCL within the EMF are [14, 20, 21], from which we took the Kent library to reuse the analysis phase for the ocl compilation.

Several instances of a model can be easily queried with our tool. Nevertheless, as the main goal of the algebraic specification of OCL in MOMENT-OCL was its use as query and data manipulation language for model transformation, the part of OCL that considers method validation has not been addressed yet.

7 Conclusions and Further Work

OCL is becoming a de-facto standard for defining constraints and queries in the Model-Driven Engineering field. The number of tools that provide support for this language is growing, and although the operational semantics of OCL is said to be formal, only a few tools rely on formal methods to define its operational semantics.

In this paper, we present an algebraic specification of part of the operational semantics of OCL 2.0 from an implementation point of view. This specification takes advantage of the parameterization mechanism for the sake of reuse.

This specification is used to perform model queries in the EMF and to represent EMF software artefacts as algebraic specifications or as terms that can be manipulated by means of model management operators [11]. The OCL specification has been developed generically so that it can be used for any kind of metamodel, model or model instance. These operators use OCL queries to specify model transformations that can be run in Maude as detailed in [6]. Thus, not only can OCL be studied in an algebraic setting, it can also be used in the well-known modelling environment EMF.

Further work consists in exploiting the formal features of the OCL specification from a more-theoretical point of view and its application to real case studies. By now,

MOMENT-OCL permits the analysis, compilation and evaluation of ocl expressions over EMF models in Maude, but the output that is returned by Maude is not still processed. Further work consists in providing such processing, allowing the user to check the invariants that have failed in an evaluation.

References

1. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer Society Press 30 (1997) 110-111
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture--Practice and Promise. (2003)
3. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons (2004)
4. OMG: Meta Object Facility (MOF) 2.0 Core Specification, ptc/04-10-15. (2004)
5. OMG: MOF 2.0 QVT final adopted specification (ptc/05-11-01). (2005)
6. Boronat, A., Carsí, J.A., Ramos, I.: Algebraic Specification of a Model Transformation Engine. In: Proceedings of Fundamental Approaches to Software Engineering, FASE'06, LNCS, Vol. 3922. Springer Vienna, Austria (2006)
7. Bézivin, J., Dupe, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: Proceedings of OOPSLA 2003 Workshop, Anaheim, California (2003)
8. The Model Transformation Framework: http://www.alphaworks.ibm.com/tech/mtf
9. Warmer, J., Kleppe, A.: The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA. Addison-Wesley (2004)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. 285 (2002) 187-243
11. Boronat, A., Carsí, J.A., Ramos, I.: Automatic Support for Traceability in a Generic Model Management Framework. In: Proceedings of Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, LNCS, Vol. 3748. Springer Nuremberg, Germany (2005) 316-330
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.2 manual and examples. (2005) http://maude.cs.uiuc.edu/maude2-manual/
13. The Eclipse Modeling Framework: http://www.eclipse.org/emf/
14. Kent Object Constraint Language Library: http://www.cs.kent.ac.uk/projects/ocl/index.html
15. Kent Modeling Framework: http://www.cs.kent.ac.uk/projects/kmf/
16. Toval, A., Requena, V., Fernández, J.L.: Emerging OCL tools. Software and System Modeling 2 (2003) 248-261
17. Ahrendt, W., Baar, T., Beckert, B., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Schmitt, P.H.: The KeY System: Integrating Object-Oriented Design and Formal Methods. In: Proceedings of Fundamental Approaches to Software Engineering. 5th International Conference, FASE 20022306. Springer, Grenoble, France (2002) 327-330
18. Richters, M.: The USE tool: A UML-based Specification Environment. (2001) http://www.db.informatik.uni-bremen.de/projects/USE/
19. Egea, M., Clavel, M.: The ITP/OCL tool. (2006) http://maude.sip.ucm.es/itp/ocl/
20. Vanwormhoudt, G.: EMF OCL Plugin. (2006) http://www.enic.fr/people/Vanwormhoudt/siteEMFOCL/maven-reports.html
21. Eclipse Modeling Framework Technologies. (2006) http://www.eclipse.org/emft/projects/

XI Jornadas de Ingeniería del Software y Bases de Datos
JISBD 2006
José Riquelme - Pere Botella (Eds)
© CIMNE, Barcelona, 2006

DEFINICIÓN DE OPERACIONES COMPLEJAS CON UN LENGUAJE ESPECÍFICO DE DOMINIO EN GESTIÓN DE MODELOS[1]

Abel Gómez, Artur Boronat, Luis Hoyos, José Á. Carsí e Isidro Ramos

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camí de Vera, s/n
46022 València
e-mail: {agomez,aboronat,lhoyos,pcarsi,iramos}@dsic.upv.es, web: http://issi.dsic.upv.es

Palabras clave: Ingeniería dirigida por modelos, Gestión de modelos, Lenguajes específicos de dominio, *Maude*.

Resumen. *La Ingeniería dirigida por Modelos permite incrementar la productividad en el proceso de desarrollo software, obteniendo herramientas más interoperables y sencillas de mantener mediante técnicas que elevan el nivel de abstracción. En esta dirección ha aparecido la disciplina «Gestión de Modelos», que proporciona un conjunto de operadores genéricos basados en teoría de conjuntos para tratar con modelos. Esta aproximación muestra su potencia en las capacidades de composicionalidad de los operadores que proporciona. Este artículo describe cómo proporciona soporte a la definición de operadores complejos una herramienta del marco de la Gestión de Modelos mediante un lenguaje específico de dominio.*

1. INTRODUCCIÓN.

En la iniciativa MDA, un artefacto software es considerado como un modelo. Un modelo en este contexto es la especificación de la funcionalidad, estructura y/o comportamiento de un sistema o aplicación [12], de forma que permite el desarrollo de éstas de forma automática mediante técnicas de programación generativas [7]. El proceso de desarrollo software en esta filosofía se basa en el refinamiento de los artefactos software desde el espacio del problema (donde se capturan los requisitos de la aplicación), al espacio de la solución (donde se especifica el diseño y desarrollo del producto software final). Durante este proceso de refinamiento se aplican diversas operaciones a los modelos, como por ejemplo, transformaciones o integraciones de distintos modelos. Tradicionalmente, este tipo de tareas

[1] Este artículo ha sido financiado por el Proyecto Nacional de Investigación, Desarrollo e Innovación DYNAMICA TIC 2003-07804-C05-01.

Gómez, A., Boronat, A., Hoyos, L., Carsí, J. A., Ramos, I. Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos. Actas de las XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006). 2006.

se han resuelto de manera *ad-hoc* para un contexto o metamodelo específico.
Para proporcionar un marco de trabajo genérico se propuso una nueva disciplina llamada Gestión de Modelos en [2]. Ésta considera los modelos y las correspondencias entre ellos como entidades de primer orden, proporcionando un conjunto de operadores independientes de metamodelo y basados en teoría de conjuntos para tratar con ellos (*Merge*, *Cross*, *Diff*, *ModelGen*, etc.). Estos operadores proporcionan una solución reutilizable y componible para las tareas descritas anteriormente.
Empleando nuestra experiencia en la aplicación de la lógica ecuacional de pertenencia para resolver problemas actuales de la ingeniería del software [5], se ha desarrollado una herramienta (llamada MOMENT) que da soporte algebraico a estos operadores mediante un eficiente sistema de reescritura de términos —*Maude* [6]— desde un entorno de modelado industrial.
El entorno elegido para integrar MOMENT ha sido *Eclipse Modeling Framework* (EMF) [15]. La integración de estas tecnologías permite aprovechar las capacidades de modelado de EMF, y la creación de interfaces de usuario amigables, obteniendo interfaces sencillas de emplear ocultando las peculiaridades de *Maude* al usuario.
Para ocultar estas peculiaridades en la declaración de operadores complejos en *Maude*, se ha diseñado en MOMENT un Lenguaje Específico de Dominio (*Domain Specific Language*). En la definición de este lenguaje se ha seguido la propia filosofía de desarrollo software dirigido por modelos, de forma que la declaración de un operador se representa mediante un modelo EMF. El modelo correspondiente a una operación compleja de Gestión de Modelos puede construirse mediante las interfaces gráficas que proporciona Eclipse o mediante una representación textual. La especificación final del operador en *Maude* se obtiene de forma automática mediante técnicas de generación automática de código.
Este artículo muestra cómo se ha resuelto en MOMENT la definición de operadores complejos mediante la aplicación a un caso de estudio no trivial. La estructura del artículo es la siguiente: el apartado 2 proporciona un ejemplo que ilustra la utilidad de la definición de operadores complejos; el apartado 3 describe brevemente MOMENT y algunos de los operadores simples disponibles que son empleados en la solución propuesta al ejemplo de motivación. El apartado 4 describe la arquitectura de la herramienta. El apartado 5 describe la solución propuesta para el problema presentado en el apartado 2, y finalmente, los apartados 6 y 7 describen algunos trabajos relacionados y las conclusiones respectivamente.

2. EJEMPLO DE MOTIVACIÓN: PROPAGACIÓN DE CAMBIOS.

Como ejemplo de motivación, utilizaremos un escenario de propagación de cambios semejante al introducido en [13], donde en primer lugar se define un diagrama de clases UML que modela un sistema de información para una aplicación.
Para construir la aplicación que almacene la información en una base de datos relacional, se utiliza la información descrita en el diagrama *Uml*. Aplicando un mecanismo de transformación (paso 1), obtenemos el nuevo esquema relacional *(Rdb)*. El mecanismo de transformación también genera un conjunto de enlaces entre el nuevo esquema relacional generado (*Rdb*) y el diagrama de clases origen (*Uml*) para proporcionar soporte a la

trazabilidad (*MapUml2Rdb*).

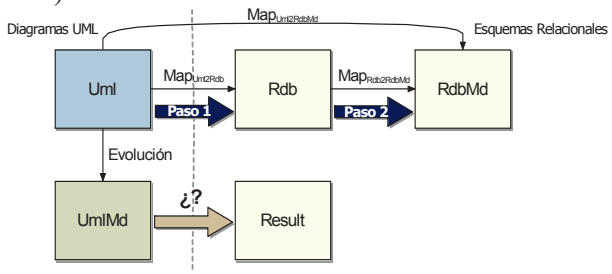


Figura 1. Ejemplo de propagación de cambios.

Tras obtener un esquema relacional a partir del diagrama UML original, se continúa con el desarrollo del nuevo sistema. Esto puede implicar cambios en la aplicación y en el esquema de la base de datos (paso 2), obteniendo el esquema relacional (*RdbMd*[2]). Estos cambios son trazados y almacenados por la herramienta que gestiona la manipulación del modelo, o introducidos directamente por el usuario (*MapRdb2RdbMd*).

Una vez se ha desarrollado el nuevo sistema, pueden producirse cambios en los requisitos del sistema, requiriendo modificaciones sobre el modelo original *Uml*. Es más sencillo modificar el diagrama *Uml* que modificar el esquema de la base de datos *RdbMd* de forma que se mantenga la consistencia entre el diseño de la aplicación y la capa de persistencia. En este punto, la aplicación de nuevo del mecanismo de transformación aplicado en el paso 1 descartaría los cambios aplicados de *Rdb* a *RdbMd*.

Una solución a este ejemplo de propagación de cambios puede proporcionarse mediante operadores de Gestión de Modelos.

3. MOMENT: UN FRAMEWORK PARA LA GESTIÓN DE MODELOS.

MOMENT es la herramienta de Gestión de Modelos que nos permitirá abordar la solución del problema del caso de estudio mediante la definición de una operación compleja de Gestión de Modelos. El siguiente apartado presenta brevemente los fundamentos en los que se apoya esta herramienta, así como los operadores simples sobre los que construiremos la solución en el apartado 5.

3.1. Espacios tecnológicos en MOMENT: EMF y *Maude*.

En MOMENT se emplean los espacios tecnológicos [11] *Maude* y EMF. El álgebra de operadores propuesta por Bernstein para tratar con modelos se ha especificado directamente como un álgebra genérica usando el formalismo de especificaciones algebraicas *Maude*. Mediante un plug-in que integra *Maude* dentro del entorno de desarrollo Eclipse podemos usar este sistema para nuestros propósitos.

[2] Los sufijos «–Md» y «–Unmd» son contracción de «Modificado» y «No modificado» respectivamente. Se ha adoptado esta convención en todo el texto para evitar el uso de nombres de variables excesivamente largos en el ejemplo de aplicación al caso de estudio.

3.2. Operadores en MOMENT.

En MOMENT cada operador se encuentra definido de forma genérica en un módulo paramétrico. Para aplicar estos operadores sobre unos modelos concretos, estos módulos deben ser instanciados con los metamodelos implicados en la operación. Toda esta tarea la realiza automáticamente los mecanismos de control de ejecución de MOMENT.

3.2.1. Operadores comunes.

Los operadores comunes son aquellos aplicables a cualquier tipo de modelo y se basan en teoría de conjuntos[3]. Los operadores de interés para nuestro caso de estudio son los siguientes:

1. *Cross* y *Merge*. Estos operadores corresponden a operaciones de conjuntos bien definidas: intersección y unión disjunta respectivamente. Ambos operadores reciben dos modelos (A y B) como entradas, y producen un tercer modelo (C). También devuelven sendos modelos de enlaces (*mapAC* y *mapBC*) que relacionan los elementos de cada modelo de entrada con los elementos del modelo resultante.
2. *Diff*. Este operador realiza la diferencia entre dos modelos de entrada (A y B) devolviendo un modelo resultado (C) y un único modelo de enlaces (*mapAC*), —un modelo *mapBC* no es interesante porque en C no habrá elementos de B—.
3. *ModelGen*. *ModelGen* realiza la traducción de un modelo (A) (o un conjunto de modelos), definidos mediante sus respectivos metamodelos[4], a un metamodelo MMB destino, obteniendo el modelo B, según una transformación determinada especificada en el lenguaje de transformaciones QVT [14]. Este operador produce a su vez un modelo de correspondencias relacionando los elementos de cada uno de los modelos de entrada con los elementos del modelo generado.

3.2.2. Operadores de soporte para la navegación:

Son operadores más específicos que los anteriores, diseñados para tratar con modelos de trazabilidad (modelos de enlaces que permiten relacionar elementos entre dos modelos distintos). En las siguientes definiciones encontramos estos elementos: dos modelos, uno denominado *dominio*¸ y otro modelo *rango* (A y B); un modelo de trazabilidad (mapAB) que relaciona los elementos de dos modelos de entrada; un modelo (A') que es un submodelo de A; y un modelo (B') que es un submodelo de B. Los operadores de trazabilidad considerados son:

1. *Range*. Dados un modelo de trazabilidad (mapAB), un modelo dominio (A'), y un modelo rango (B), obtiene un submodelo del modelo rango B (B') cuyos elementos se relacionan con los de A'.
2. *RestrictDomain*. Dado un modelo dominio (A') y un modelo de trazabilidad (mapAB), devuelve un modelo con los enlaces de trazabilidad de «mapAB» que tienen elementos

[3] La semántica de los operadores se asemeja a los operadores de teoría de conjuntos. No obstante, como se muestra en [4], esta semántica puede refinarse en función del metamodelo sobre el que se aplican.
[4] O como se expresaría según la terminología inglesa, «un modelo conforma a su respectivo metamodelo».

del modelo A' como elementos dominio.

3. *Compose.* Dados tres modelos A, B y C, y los modelos de trazabilidad «mapAB» y «mapBC» que relacionan respectivamente A con B y B con C; el operador *Compose* obtiene el modelo «mapAC» haciendo explícita la relación entre los elementos de A y C por la propiedad transitiva.

4. EJECUCIÓN DE OPERADORES COMPLEJOS EN MOMENT.

El desarrollo de MOMENT ha sido realizado siguiendo la propia filosofía de desarrollo de software dirigido por modelos. Se ha desarrollado un modelo de la aplicación empleando EMF que describe todos los elementos involucrados en la ejecución de operadores. Posteriormente, mediante las capacidades de generación de código que proporciona EMF se ha obtenido el código Java de la aplicación.

4.1. Soporte para la definición de operadores complejos.

Esta filosofía, además de incrementar la mantenibilidad del código, permite manipular todos los elementos representados en el modelo de la aplicación a un mayor nivel de abstracción. Además proporciona un mecanismo automático de persistencia y recuperación de datos (por defecto el formato de persistencia es XMI, lo que también proporciona interoperabilidad con otras aplicaciones).

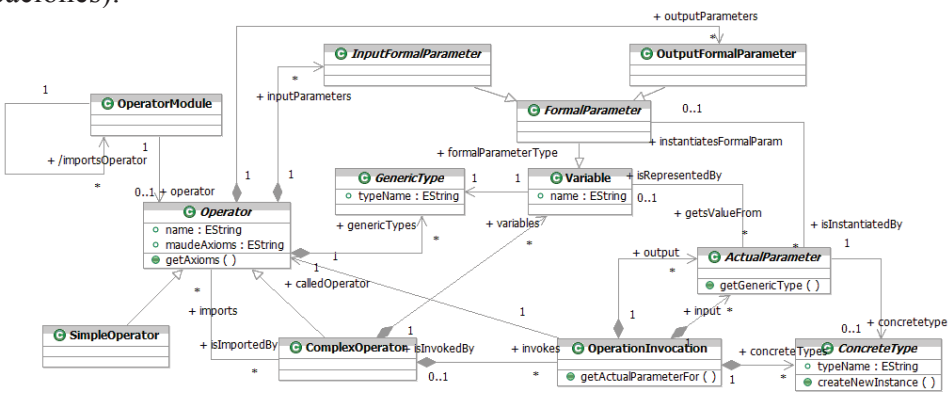


Figura 2. Modelo simplificado para la especificación de operadores complejos en MOMENT.

El diagrama de clases de la Figura 2 muestra la parte del modelo de la herramienta que permite la declaración y ejecución de operadores de gestión de modelos. La clase *Operator* captura la información de la declaración de un operador. Esta clase se especializa en operadores simples *(SimpleOperator)* y complejos *(ComplexOperator)*. Los parámetros formales tanto de entrada *(InputFormalParameter)* como de salida *(OutputFormalParameter)* son especializaciones de la clase *Variable*; disponiendo todos ellos tanto de un nombre como de un tipo *GenericType*. Los tipos en la declaración de un operador se denominan genéricos puesto que, como se comentó en el apartado 3.2, la declaración de un operador se realiza de forma genérica independiente del metamodelo concreto.

La clase *OperationInvocation* captura la información sobre la ejecución de un determinado

operador (referenciado mediante el rol *calledOperator*) con unos datos concretos, representados en la figura mediante la clase *ActualParameter* (parámetro actual). Cada parámetro actual es el dato concreto con el que se invoca un operador e instancia un parámetro formal de la declaración de un operador (rol *instantiatesFormalParameter*). De igual forma que un parámetro actual, también dispone de un tipo *(ConcreteType)*. Por ejemplo, un parámetro actual podría ser un diagrama de clases UML concreto (por ejemplo el modelo *Uml* del caso de estudio), y su tipo concreto sería el metamodelo UML.
Esta representación de los operadores complejos como instancias del modelo de la aplicación permite tratar con ellos a un mayor nivel de abstracción, ya que las manipulaciones y consultas sobre un operador se realizan sobre los conceptos modelados en la figura, y no sobre un árbol de sintaxis abstracta construido en tiempo de compilación a partir de una gramática. Finalmente, dada esta representación del operador y mediante técnicas generativas, se obtiene el código *Maude* en la forma de un módulo paramétrico independiente de metamodelo.

4.2. Arquitectura de la herramienta.

MOMENT emplea un proceso *Maude* desde un programa Java. Para ello se ha hecho uso de *Maude Development Tools* [8], un conjunto de herramientas que extienden Eclipse y proporcionan una API para el uso de *Maude*. El diagrama de componentes UML de la Figura 3 muestra los elementos de MOMENT relacionados con la ejecución de operaciones de Gestión de Modelos. Los principales módulos son los denominados «Lanzador de operaciones» *(Operator Launcher)* y «Cargador de módulos» *(Module Loader)*.

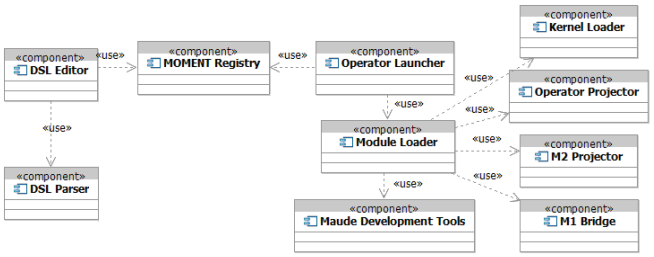


Figura 3. Componentes relacionados con la ejecución de operaciones de gestión de modelos.

El primero de ellos es el que proporciona la interfaz de MOMENT al usuario. Permite, dada la declaración de un operador (sea simple o complejo), especificar sus parámetros actuales así como dónde se guardarán los resultados devueltos. El segundo componente, el cargador de módulos, se encarga de controlar de forma transparente al usuario el proceso *Maude* sobre el que se ejecutarán las operaciones.
MOMENT se sustenta sobre un conjunto básico de módulos *Maude* (denominado *kernel*) que proporcionan la funcionalidad básica del *framework*. Entre estos módulos encontramos aquellos que implementan los operadores simples comentados en el apartado 3.2. El *Module Loader*, se ocupa de la preparación del contexto de ejecución (carga de los módulos del kernel por parte del *Kernel Loader*) y de la proyección a código *Maude* de todos los elementos que intervendrán en la ejecución. Esto incluye la especificación algebraica de los metamodelos

implicados *(M2 Projector)*, el código del operador a ejecutar *(Operator Projector)*, y los términos correspondientes a los parámetros de entrada en el momento de la invocación de la operación *(M1 Bridge)*. Finalmente, el puente a nivel M1 *(M1 Bridge)* es también el componente encargado de procesar los términos resultantes de una ejecución y recuperarlos en el espacio tecnológico de EMF. Es de destacar en este punto la expresividad de *Maude* puesto que a diferencia de otros lenguajes (como Java), la composicionalidad funcional de operaciones es inherente al formalismo de la lógica ecuacional de pertenencia subyacente.

4.3. Definición textual de operadores complejos.

La representación de los operadores complejos como instancias de un modelo aporta numerosas ventajas. No obstante, las interfaces gráficas proporcionadas por defecto por Eclipse resultan poco intuitivas para la definición de nuevos operadores. En este caso, la opción más natural para el usuario es ofrecer una representación textual. Por ello se ha definido una gramática [8] para soportar la definición de operadores complejos de esta forma. Tal como se observa en la Figura 3, existen en MOMENT dos componentes adicionales: *DSL Editor* y *DSL Parser*. El primero corresponde a un editor de texto con coloreado de sintaxis que permite la declaración textual de operadores complejos. Mediante el *DSL Parser*, obtiene automáticamente la representación EMF equivalente a la declaración textual de un operador, que finalmente, se incluye en el repositorio (*MOMENT Registry*). Es el *Operator Launcher* quien recupera los operadores de este repositorio para ejecutarlos sobre *Maude*.

5. APLICACIÓN DE MOMENT AL CASO DE ESTUDIO.

Dado el problema del caso de estudio (apartado 2), el modelo *MapUml2RdbMd* puede ser fácilmente obtenido de los modelos *MapUml2Rdb* y *MapRdb2RdbMd* mediante el operador *Compose*. Por tanto, el problema puede enunciarse de la siguiente manera:
«Dados los siguiente modelos: un diagrama de clases UML original (Uml); un diagrama de clases UML (UmlMd) evolucionado de Uml; un esquema de base de datos relacional RdbMd, generado a partir del diagrama de clases UML y modificado posteriormente; y un modelo de trazabilidad entre UML y RDB' (MapUml2RdbMd); deberemos obtener el esquema relacional del diagrama de clases UmlMd que conserve los cambios realizados en RdbMd.»
Este problema puede ser resuelto por el siguiente operador complejo:

```
operator PropagateChanges(MM1 Uml, MM1 UmlMd, MM2 RdbMd,
                TraceabilityMetamodel MapUmlIni2RdbMd, Transformation Uml2Rdbms)
        : <MM2, TraceabilityMetamodel>
{   // Obtención del modelo resultado
    <UmlUnmd, MapUml2UmlUnmd, MapUmlMd2UmlUnmd> = Cross(Uml, UmlMd);                (1)
    <RdbUnmd> = Range(mapUml2RdbMd , UmlUnmd, RdbMd);                               (2)
    <UmlNew, MapUmlMd2UmlNew, MapUmlUnmd2UmlNew> = Diff(UmlMd, UmlUnmd);            (3)
    <RdbNew, MapUmlNew2RdbNew> = ModelGen(Uml2Rdbms, UmlNew);                       (4)
    <Result, MapRdbUnmd2Result, MapRdbNew2Result> = Merge(RdbUnmd, RdbNew);         (5)
    // Generación del modelo de trazabilidad
    <MapUmlUnmd2RdbUnmd> = RestrictDomain(UmlUnmd , MapUml2RdbMd);                  (6)
    <MapUmlUnmd2Result> = Compose(MapUmlUnmd2RdbUnmd, MapRdbUnmd2Result);           (7)
    <MapUmlNew2Result> = Compose(MapUmlNew2RdbNew, MapRdbNew2Result);               (8)
    <MapUmlMd2Result, Map1, Map2> = Merge(MapUmlUnmd2Result, MapUmlNew2Result);     (9)
    return <Result, MapUmlMd2Result>;
}
```

Este operador está construido a partir de operadores simples del álgebra de MOMENT y realiza las siguientes operaciones (ver Figura 4): (1) «UmlUnmd» es la parte del modelo UML que permanece sin modificar en el modelo *UmlMd*. (2) «RdbUnmd» es el submodelo de *RdbMd* que corresponde a la parte no modificada de *UmlMd*. (3) «UmlNew» es la parte de *UmlMd* que ha sido añadida al modelo *Uml*. (4) «RdbNew» es el esquema relacional obtenido de la traducción de *UmlNew* al metamodelo relacional. (5) «Result» es el modelo final obtenido de la integración de las bases de datos obtenidas en los pasos 2 y 4.

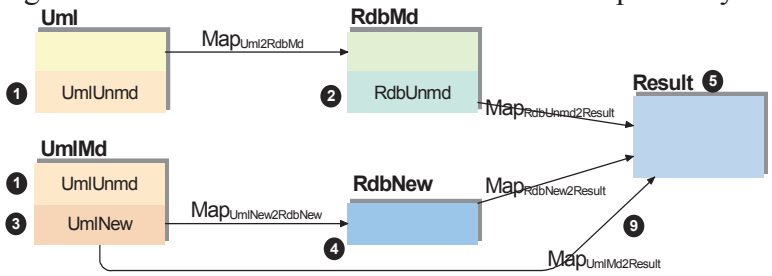


Figura 4: Solución al problema del caso de estudio.

«MapUmlUnmd2RdbUnmd» (6) es el modelo de trazabilidad que enlaza los elementos de *Uml* con elementos de *RdbMd* que han permanecido sin cambios. (7) *MapUmlUnmd2Result* es el modelo de trazabilidad que relaciona la parte no modificada de *Uml (UmlUnmd)* y *Result* combinando *MapUmlUnmd2RdbUnmd* y *MapRdbUnmd2Result*. (8) Obtiene el modelo de trazabilidad entre la parte nueva de *UmlMd* y *Result*. Finalmente, (9) une los modelos obtenidos en 7 y 8 mediante el operador *Merge*, obteniendo *MapUmlMd2Result*.

Este operador resuelve el problema de la propagación de cambios de forma independiente de los metamodelos implicados, por lo que puede ser aplicado a cualquier combinación de metamodelos (en lugar de UML y relacional). Tras obtenerse su representación como modelo EMF mediante el *parser* implementado, se obtiene el código *Maude* que encontramos en [8].

6. TRABAJOS RELACIONADOS.

En el campo de la ingeniería dirigida por modelos pocas son las aproximaciones que proporcionan composicionalidad en sus operaciones, especialmente en la disciplina de Gestión de Modelos, dado el reducido número de herramientas que proporcionan un conjunto de operadores tal y como la Gestión de Modelos los define.

RONDO [13] es la plataforma de Gestión de Modelos basada en los trabajos de P. Bernstein [13]. Ésta proporciona un lenguaje de *script* similar al que proporciona MOMENT mediante el cual se pueden definir operadores complejos. No obstante, en esta herramienta, el operador *Merge* por ejemplo, recibe como entradas dos modelos (A y B) y un modelo de *mappings* entre ellos (*mapAB*); produciendo el modelo combinado C y dos nuevos modelos de correspondencias (*mapAC* y *mapBC*): *<C, mapAC, mapBC> = Merge(A, B, mapAB)*.

En MOMENT, sin embargo, las relaciones de equivalencia entre los elementos de dos modelos (necesarias para aplicar un operador), se definen entre los elementos de sus metamodelos de forma axiomática. Esta colección de relaciones de equivalencia entre dos

metamodelos constituye un morfismo que puede ser reusado por todos los operadores del álgebra de MOMENT permitiendo una especificación más clara de los operadores complejos.
Existen también diversas plataformas para la manipulación de modelos que se han implementado sobre Eclipse y su *framework* de modelado *(EMF)* como por ejemplo *ATL* [3] (*Atlas Transformation Languaje*). Esta plataforma proporciona un lenguaje que permite especificar transformaciones de modelos de forma declarativa o imperativa, de forma dependiente del metamodelo. Este lenguaje carece de la potencia para la composición de operadores de MOMENT. En este sentido ATL únicamente proporciona la capacidad para definir *librerías* de funciones auxiliares que no son ejecutables de forma independiente [1]. En MOMENT, al contrario, un operador siempre es una unidad ejecutable de forma independiente y susceptible de ser incluida en la definición de un operador complejo mayor.
En el marco de EMF también se encuentre *Epsilon Object Language* [10]. Esta aproximación proporciona un lenguaje de bajo nivel para la manipulación de modelos, con el objetivo de construir lenguajes de fines más específicos sobre éste. EOL está inspirado en los mecanismos de navegación de OCL [17]. Esta forma de navegación, basada en la estructura de los modelos, impide la definición de operaciones genéricas fácilmente componibles.

7. CONCLUSIONES.

La Gestión de Modelos es un campo de investigación emergente en la ingeniería dirigida por modelos que destaca por su potencia en la composición de operaciones de una forma natural, intuitiva, genérica y reutilizable.
En este artículo hemos presentado la herramienta MOMENT desde el prisma de la definición de operadores complejos.
Maude proporciona una implementación de la lógica ecuacional de pertenencia, que ha sido utilizada para definir las operaciones de gestión de modelos mediante módulos funcionales. En estos módulos funcionales, las operaciones son descritas como funciones y dependiendo de las propiedades algebraicas que son añadidas a cada operación (asociatividad, conmutatividad, etc) se pueden componer fácilmente. Estas facilidades de composición de funciones han sido reflejadas en el lenguaje de definición de operadores complejos.
Por otra parte, el uso de una herramienta como Eclipse y EMF proporciona a esta aproximación una gran interoperabilidad y extensibilidad, a la vez que permite aplicar la propia filosofía de ingeniería dirigida por modelos al diseño del lenguaje de definición de operadores complejos. La representación de un operador a un mayor nivel de abstracción permite el aprovechamiento de técnicas de programación generativas para la obtención del código ejecutable final y facilita el diseño de diversas interfaces de usuario para la definición y modificación de operadores complejos con independencia del lenguaje empleado por el usuario. Ejemplo de esta independencia es que un operador complejo puede ser definido mediante un editor gráfico proporcionado por defecto por Eclipse, o mediante el editor textual implementado en MOMENT que proporciona una sintaxis más intuitiva.
Pero también cabe destacar que esta representación como instancia de un modelo, permite aprovechar el esfuerzo de terceros en el campo de la ingeniería dirigida por modelos en, por ejemplo, la creación semi-automática de lenguajes visuales específicos de dominio.

De esta manera, como trabajo futuro encontramos el desarrollo de un editor visual para la definición de operadores complejos aprovechando los esfuerzos del proyecto *Graphical Modeling Framework* (GMF) [16], que permite la generación de editores gráficos para la edición de modelos EMF mediante una metáfora gráfica específica de dominio.

REFERENCIAS.

[1] ATLAS group, LINA & INRIA, Nantes. *ATL User Manual. Version 0.7*. http://www.eclipse.org/gmt/atl/doc/ATL_User_Manual[v0.7].pdf

[2] Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: *A Vision for Management of Complex Models*. Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00)).

[3] Bézivin, J., Valduriez. P., Jouault, F. *The ATL home page*. http://www.sciences.univ-nantes.fr/lina/atl/

[4] Boronat, A., Carsí, J.Á., Ramos, I., Letelier, P. *Formal Model Merging Applied to Class Diagram Integration*. Int. ERCIM Workshop on Soft. Evolution. 2006. Lille.

[5] Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: *Two experiencies in software dynamics*. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.

[6] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: *Maude: specification and programming in rewriting logic*. Theoretical Computer Science, 285(2):187-243, 2002.

[7] Czarnecki, K., Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*. Addisson-Wesley (2000). ISBN 0-201-30977-7, pag. 267-304.

[8] Gómez, A., Boronat, A., Carsí, J.Á., Ramos, I. *Integración de un sistema de reescritura de términos en una herramienta de desarrollo software industrial*. IV Jornadas de trabajo DYNAMICA. Archena (Murcia), 2005.

[9] Gramática para la definición de operadores complejos y código Maude proyectado para el caso de estudio. http://moment.dsic.upv.es/Default.aspx?tabid=86

[10] Kolovos D. *Website de EOL*. http://www-users.cs.york.ac.uk/~dkolovos/epsilon/

[11] Kurtev, I., Bézivin, J., Aksit, M. *Technological Spaces: An Initial Appraisal*. Int. Federated Conf. (DOA, ODBASE, CoopIS). Irvine, 2002.

[12] *MDA Guide Version 1.0.1*. http://www.omg.org/docs/omg/03-06-01.pdf

[13] Melnik, S., Rahm E., Bernstein P. A., *Rondo: A Programming Platform for Generic Model Management*. SIGMOD 2003. Extended ver. in Web Semantics, vol. 1, Num. 1.

[14] OMG: MOF 2.0 QVT final adopted specification (ptc/05-11-01). (2005).

[15] Sitio web de EMF: http://www.eclipse.org/emf/

[16] Sitio web de GMF: http://www.eclipse.org/gmf/

[17] Warmer, J., Kleppe, A.: *The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA*. Addison-Wesley (2004).

Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos.*

Abel Gómez, Artur Boronat,
Jose Á. Carsí, Isidro Ramos

Departament de Sistemes Informàtics i Computació.
Universitat Politècnica de València.
Camino de Vera, s/n.
46022 València. España.
{ agomez | aboronat | pcarsi | iramos } @ dsic.upv.es

Claudia Täubner,
Silke Eckstein

Institut für Informationssysteme.
Technische Universität Braunschweig.
Mühlenpfordtstraße 23, 2.OG.
D-38106 Braunschweig. Deutschland.
{ c.taeubner | s.eckstein } @ tu-bs.de

Resumen

Este artículo muestra cómo el proceso de desarrollo de software dirigido por modelos (DSDM) es aplicable al campo de la bioinformática ya que la estructura de los datos biológicos se puede expresar mediante modelos de forma muy natural. En el contexto de la bioinformática es común la existencia de fuentes de datos (rellenadas de forma manual) heterogéneas. Con el objetivo de validar la información de estas fuentes de datos, se han adaptado diversos formalismos y herramientas de simulación. El proceso de introducción de datos —obtenidos de estas bases de datos— en las herramientas de validación se realiza tradicionalmente de forma manual. Este trabajo describe cómo se ha resuelto este problema siguiendo una metodología de DSDM empleando transformaciones de modelos. Esto permite automatizar el proceso de migración de datos, obtener herramientas modulares, aislar el proceso de transformación de datos de los formatos de persistencia de estos, y disponer de información de trazabilidad.

Palabras clave: Desarrollo de Software Dirigido por Modelos (DSDM), bioinformática, migración de datos.

*Este artículo ha sido financiado por el Proyecto Nacional de Investigación Científica, Desarrollo e Innovación META TIN2006-15175-C05-01.

1. Motivación.

La práctica tradicional en los estudios científicos de «experimentación → análisis → publicación» está cambiando a «experimentación → organización de datos → análisis → publicación». Esto se debe a que, hoy en día, los datos no se obtienen únicamente de experimentos, sino que también se obtienen de simulaciones. Esta gran cantidad de datos generados no siempre tiene la misma estructura y puede estar almacecenada en bases de datos heterogéneas. Además, el elevado número de datos requiere el desarrollo de herramientas informáticas que nos permitan representarlos para analizarlos, y a su vez, realizar otras simulaciones con ellos.

En el campo de la bioinformática encontramos esta problemática en el análisis y simulación de los mecanismos de señales de las células (*Pathways*). Un *pathway* es un conjunto de reacciones químicas que se producen en el interior de una célula tras la recepción de un estímulo. En estudios de este tipo se observa tanto el rellenado independiente de fuentes de datos como el desarrollo independiente de herramientas de modelado. Por ello, estos datos deben ser reconvertidos de forma manual para que puedan ser utilizados en las herramientas de simulación. En un escenario así, disponer de herramientas interoperables es deseable.

El Desarrollo de Software Dirigido por Mo-

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I., Täubner, C., Eckstein, S. Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007.

delos —DSDM— es una aproximación que pretende dar solución a problemas como éste. En el DSDM, un modelo es una estructura de datos que puede ser definida mediante un lenguaje de modelado (generalmente llamado metamodelo). Un modelo permite definir la funcionalidad, estructura y/o comportamiento de un sistema [8] dependiendo del metamodelo utilizado. La utilización de modelos en un proceso de DSDM permite automatizar el desarrollo de aplicaciones y su evolución mediante técnicas de programación generativa [2], como las transformaciones de modelos y la generación de código.

Este artículo muestra cómo la filosofía de DSDM permite resolver los problemas surgidos en el estudio de los caminos de señales en el campo de la bioinformática. Los problemas de interoperabilidad entre aplicaciones pueden abordarse de una manera sistemática, donde (a) el formato de datos puede definirse como un modelo y (b) los datos se definen como una colección de objectos que son instancia de las clases del este modelo. El tratamiento de los datos desde la perspectiva del DSDM permite (c) el desarrollo de herramientas donde el tratamiento de éstos sea independiente de los formatos de persistencia, obteniendo herramientas más modulares. Esto a su vez facilita (d) la automatización en la migración de los datos mediante el uso de técnicas de transformaciones de modelos. Todos estos factores reducen el coste de producción de las herramientas afectando de forma directa y positiva en la productividad de los usuarios/biólogos.

El documento se organiza de la siguiente manera: en el apartado 2 se expone el contexto biológico, introduciendo las razones por las que es interesante el estudio de los caminos de señales (*pathways*), y se describe la aproximación actual, en la que esto es un proceso poco eficiente. El apartado 3 introduce las bases del desarrollo dirigido por modelos y la tecnología empleada para llevar a cabo la migración de datos. En el apartado 4 se describe la solución diseñada para resolver las deficiencias de la aproximación actual, y por último en el apartado 5 se exponen las conclusiones finales y trabajos futuros.

2. Caso de estudio.

Numerosos procesos biológicos son controlados por los denominados *mecanismos de señales*. Un camino de señales —*signaling pathway*— es una colección de reacciones químicas que tienen lugar en una célula como reacción a un estímulo[1] (generalmente externo) disparando diversos eventos en el interior de ésta. Estos eventos son los que controlan la progresión del ciclo de vida (muerte, crecimiento, especialización), activación o desactivación de genes para el momento de la reproducción, el movimiento de la célula, etc. El estudio de estos caminos de señales se realiza mediante el ensayo empírico de los estímulos externos que recibe una célula y recogiendo los datos sobre las reacciones químicas desencadenadas en su interior. Estos datos obtenidos de forma experimental por los biólogos son almacenados en diferentes bases de datos para su posterior estudio y consulta por otros expertos. Existen diversas fuentes donde es almacenada la información como por ejemplo TRANSPATH®, BioCarta, KEGG o Reactome.

Las reacciones químicas involucradas en un *pathway* suelen ser un proceso en cascada, produciéndose entonces reacciones de forma concurrente. Por ello, para el modelado y análisis de estos procesos biológicos se han adaptado y aplicado algunos de los formalismos desarrollados para el análisis de sistemas concurrentes. Ejemplo de ello son aproximaciones en π-cálculo [12], ambient calculus [11], *Life Sequence Charts* (LSCs) [3] o redes de Petri [7].

El caso de estudio parte de los trabajos realizados por Taübner et al. [13], donde se ha modelado, simulado y validado el *pathway* del *receptor de tipo toll 4* (TLR4) —que se introduce en el siguiente punto— mediante redes de Petri coloreadas. A continuación se expone en primer lugar el contexto biológico describiendo el problema abordado en [13] y en segundo lugar se describen los detalles tecnológicos de éste: fuente de datos y herramienta de simu-

[1] Un estímulo puede ser, por ejemplo, la presencia de una determinada molécula en el entorno de la célula.

lación seleccionadas y proceso de extracción y tratamiento de los datos.

2.1. Los receptores de tipo *Toll*. El *pathway* TLR4.

Los receptores de tipo *Toll* —*Toll-like receptors (TLR)*— son una familia de proteinas que forman parte del sistema inmunitario. Permiten la adaptación del sistema inmune de diversos seres vivos siendo responsables del reconocimiento de diversos patrones (secuencias de moléculas) que identifican a diferentes patógenos. Una vez reconocido el patógeno (cualquier entidad biológica capaz de producir daño) estimulan la respuesta de defensa contra él por parte del sistema inmunitario.

Actualmente se han identificado trece receptores de tipo *Toll* en humanos (TLR1-TLR13). Estos receptores reconocen moléculas que están constantemente asociadas a amenazas contra el sistema inmune y que son muy específicas a estas amenazas, no pudiendo confundirlas con moléculas propias. Algunas de estas moléculas específicas pueden ser los *lipopolisacáridos* (LPS) presentes en la superficie de células bacterianas, proteínas presentes en los flagelos de bacterias, ARN de doble cadena presente en virus, etc.

Para el caso de estudio, por simplicidad, se toma como ejemplo una pequeña parte del *pathway* en el que interviene el receptor de tipo *Toll* 4 —TLR4—, por ser el primero reconocido en mamíferos y el mejor estudiado hasta el momento. A continuación se indican las reacciones que serán transformadas en el caso de estudio.

1. `LPS + LBP` $\leftrightharpoons$ `LPS:LBP`
2. `LPS:LBP + CD14` $\leftrightharpoons$ `LPS:LBP:CD14`
3. `ST2 + TIRAP` $\leftrightharpoons$ `ST2:TIRAP`
4. `ST2 + MyD88` $\leftrightharpoons$ `ST2:MyD88`

La notación expresa a la izquierda los reactivos y a la derecha los productos. El símbolo «$\leftrightharpoons$» expresa que esta reacción es bidireccional, ya que se trata de una reacción de *equilibrio químico* —informalmente se puede describir como una reacción que se mantiene hasta que los reactivos y los productos alcanzan un estado de equilibrio, no llegándose a consumir ninguno de los dos—. Para `(1)`, por ejemplo, podríamos expresar de forma sencilla lo siguiente: *la mólecula LPS se une a la molécula LBP, dando lugar a la molécula compuesta LPS:LBP.*

2.2. Aproximación actual en el estudio del pathway del TLR4 en bioinformática.

Son numerosas las fuentes de datos de las que se puede extraer información sobre el *pathway* del *TLR4*. Algunas de ellas pueden ser TRANSPATH®, KEGG, Reactome o BioCarta. Estas bases de datos han sido rellenadas a partir de estudios empíricos y pueden contener información incompleta e incluso contradictoria. Los actuales trabajos para la representación y simulación de pathways se dirigen en la representación y simulación de estos *pathways* para poder validar la información obtenida en los estudios empíricos.

Con ese objetivo se han adaptado diversos formalismos gráficos para la simulación de *pathways* biológicos. Ejemplos de ello son los diagramas de estados [14], los *Life Sequence Charts* [14, 3], o las redes de Petri coloreadas [13]. El trabajo en el que se fundamenta la solución aquí presentada describe un proceso mediante el cual es posible extraer los datos del pathway TLR4 de la base de datos TRANSPATH® con el objetivo de simularlo mediante la herramienta de simulación y validación de redes de Petri coloreadas *CPN Tools*.

TRANSPATH® es una base de datos que almacena información acerca de caminos de señales. Permite la exportación de datos en XML, y en su versión 7.4 contiene información sobre 62.549 moléculas, 23.076 genes y 104.362 reacciones. Una red de Petri, por su parte, es una representación formal para un sistema distribuido discreto, que permite representar eventos concurrentes. La red se forma por nodos (llamados lugares), transiciones y arcos dirigidos. Estos arcos conectan siempre un lugar con una transición o una transición con un lugar. En los lugares puede haber un determinado número de tokens, que pueden moverse de un lugar a otro cuando una transición se dispara (una transición se habilita cuando todas sus entradas contienen to-

kens). La figura 1 muestra un ejemplo de red de Petri, donde los círculos blancos son los lugares, el rectángulo relleno de color negro es una transición, las flechas son los arcos dirigidos, y los puntos negros, los tokens.

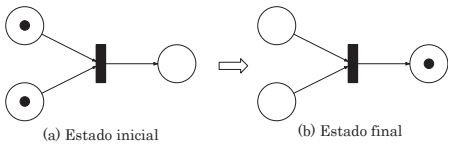


Figura 1: Ejemplo de red de Petri.

Una red de Petri coloreada es aquella en la que los tokens pueden tener un determinado color (esto es, algún atributo distintivo), pudiendo caracterizar tokens de diferentes tipos. *CPN Tools* es una herramienta que permite la construcción, modificación y validación sintáctica de redes de Petri coloreadas de forma gráfica. Dispone además de un simulador (tanto interactivo como automático) para poder inspeccionar la evolución de estos modelos.

En [13] el método empleado para poder realizar la simulación a partir de los datos almacenados en TRANSPATH® es manual. Esto supone que el usuario/biólogo que va a realizar la simulación debe consultar de forma manual la base de datos obteniendo el listado de reacciones que intervienen en el *pathway* que desea estudiar. Con los datos obtenidos debe crear y editar de forma manual la correspondiente red de Petri coloreada en *CPN Tools*, creando uno a uno cada uno de los lugares, transiciones, arcos, tokens, etc. En el trabajo al que se hace referencia, esto supuso crear de forma manual —y únicamente para el ejemplo del *pathway* del TLR4— 75 lugares, 47 transiciones y casi un centenar de colores.

3. Ingeniería Dirigida por Modelos.

La Ingeniería Dirigida por Modelos es un campo en la Ingeniería del Software que, durante años, ha representado los artefactos software como modelos con el objetivo de incrementar la productividad, calidad, y reducir los gastos en el proceso de desarrollo de software. Recientemente, existe un interés creciente en este campo. Prueba de ello es la aproximación de Model Driven Architecture [8], apoyada por la OMG.

El Desarrollo Dirigido por Modelos ha evolucionado del campo de la Ingeniería Dirigida por Modelos. En él, no sólo las tareas de diseño y generación de código están involucradas, sino que también se incluyen las capacidades de trazabilidad, tareas de meta-modelado, intercambio y persistencia de modelos, etc. Para poder abordar estas tareas, las operaciones entre modelos, transformaciones, y consultas sobre ellos son problemas relevantes que deben ser resueltos. En el contexto de MDA se abordan desde el punto de vista de los estándares abiertos. En este caso, el estándar Meta Object Facility (MOF) [10], proporciona un mecanismo para definir metamodelos. El estándar Query/Views/Transformations (QVT) [9] indica cómo proporcionar soporte tanto para transformaciones como para consultas. A diferencia de otros lenguajes nuevos, QVT se apoya en el ya existente lenguaje *Object Constraint Language* (OCL) para realizar las consultas sobre los artefactos software.

3.1. MOMENT. Un framework para la Gestión de Modelos.

MOMENT [1] es una herramienta que da soporte a los estándares propuestos por el OMG para dar soporte a transformaciones. La herramienta proporciona un soporte algebraico para las tareas de transformación y consulta de modelos mediante un eficiente sistema de reescritura de términos —Maude— y desde un entorno de modelado industrial —Eclipse Modeling Framework (EMF)—. EMF [5] puede ser visto como una implementación del estándar MOF, y permite la importación automática de artefactos software desde orígenes de datos heterogéneos: modelos UML, esquemas XML, etc. Respecto a Maude, MOMENT aprovecha las capacidades de modularidad y parametrización de este sistema para proporcionar un entorno de transformación y consulta de modelos de forma genérica e independiente de metamodelo.

Como lenguaje para transformaciones MOMENT se apoya sobre el estándar abier-

to QVT, proporcionando una implementación tanto del lenguaje QVT-Relations como de OCL. QVT-Relations es un lenguaje de transformaciones declarativo que proporciona de forma implícita capacidades de trazabilidad. Para este lenguaje, MOMENT ofrece un amplio soporte para transformaciones unidireccionales 1-a-1. Además, esta herramienta proporciona una completa implementación de los operadores de consulta del lenguaje *OCL*.

4. Un enfoque de DSDM en la migración de datos biológicos.

En el trabajo inicial sobre el estudio del *pathway* del TLR4 la migración de datos desde la base de datos origen hasta la herramienta de simulación para su representación mediante una red de Petri coloreada debe realizarse de forma manual. A continuación se muestra una solución al problema de migración de datos del caso de estudio empleando transformaciones, según la filosofía de desarrollo de software dirigido por modelos. Esto supone las siguientes tareas: (a) desarrollo del modelo de datos del dominio origen (TRANSPATH®) y (b) desarrollo del modelo de datos del dominio destino (*CPN Tools*). (c) Definición mediante un lenguaje de transformaciones las reglas de transformación entre el dominio origen y el dominio destino; (d) implementación del mecanismo de preprocesado de datos permitiendo reconstruir los datos originales como instancias del modelo origen, y (e) definición del postprocesado de los datos, que implementa el mecanismo de persistencia al formato final.

A continuación se presenta la solución proporcionada. En primer lugar, se describe el proceso de transformación y sus etapas, en segundo lugar se describen los modelos del dominio origen y el modelo destino, y por último, se describe en mayor detalle el proceso de transformación.

4.1. Arquitectura e implementación de la herramienta.

El proceso de migración de datos, tal y como se deriva de las tareas anteriores, se realiza en tres pasos: (1) recuperación y pre-procesado de los datos, (2) ejecución de la transformación mediante un motor de transformaciones y (3) post-procesado y persistencia de los datos. En una aproximación de DSDM, el uso de un motor de transformaciones implica que se deben definir en primer lugar los modelos origen y destino de la transformación para poder establecer las correspondencias entre uno y otro posteriormente.

La solución aquí presentada hace uso de MOMENT. MOMENT es una herramienta integrada en el entorno Eclipse que proporciona soporte para transformaciones. Como entorno de modelado, MOMENT emplea el *Eclipse Modeling Framework* (EMF). Este *framework* proporciona como lenguaje de modelado Ecore. Además, emplea como formato de persistencia XMI. EMF permite –entre otros— la creación de modelos Ecore a partir de un esquema XSD. No obstante, los modelos Ecore obtenidos a partir de esquemas XSD son complejos y no siempre representan la semántica de los datos de forma clara. Por ello, se ha decidido definir de forma manual los modelos origen y destino, teniendo sólo en cuenta aquella información relevante para el caso de estudio.

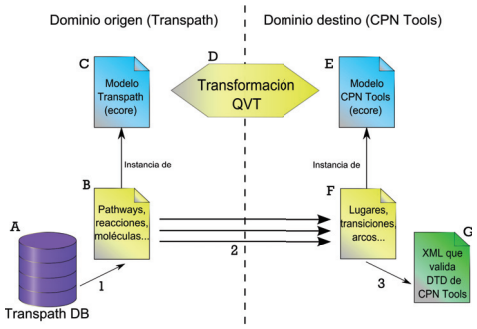


Figura 2: Arquitectura de la herramienta.

La figura 2 muestra la arquitectura de la herramienta. En ella se reflejan de forma clara los 3 pasos necesarios para realizar la migración de los datos. En primer lugar se extraen los datos de la base de datos TRANSPATH® (A), y se reconstruyen como una instancia en XMI (B) del modelo Ecore de TRANSPATH® (C). Este primer paso (1) se realiza de forma

sencilla en Java ya que la correspondencia de los datos origen con el modelo en EMF es directa. La implementación de este primer paso de pre-procesado de los datos es una adaptación del trabajo realizado en [15].

El segundo paso de la migración (2) es el principal y más complejo. Se realiza mediante MOMENT y su motor de transformaciones, ejecutando la transformación del dominio de TRANSPATH® (reacciones, moléculas, etc.) al dominio de *CPN Tools* (lugares, transiciones, arcos, etc.). Tras la definición de las reglas de transformación (D) entre los dominios origen (C) y destino (E), se aplica la transformación sobre los datos recuperados de la base de datos (B) obteniendo una instancia en el dominio de *CPN Tools* (F).

Por último, el tercer paso de la migración (3) es de nuevo un proceso trivial en el que se persisten los datos desde EMF (F) a un fichero XML comprensible por la herramienta *CPN Tools* (G). En este, paso además, pueden realizarse otras tareas de post-procesado, como por ejemplo, la inclusión de algún algoritmo de redibujado (*layout*) sobre los elementos de la red de Petri.

4.2. Desarrollo de los modelos.

En primer lugar se ha definido un modelo únicamente con las partes que resultan de interés para la simulación de un pathway eliminando los conceptos innecesarios de la compleja base de datos TRANSPATH®. La figura 3 muestra —mediante una metáfora visual similar al diagrama de clases de UML— el modelo Ecore desarrollado. En el modelo encontramos la clase *Network*, elemento raíz del modelo. Una Red —*Network*— contiene un conjunto de *Pathways*, *Reacciones* y *Moléculas*. A su vez, un *Pathway* se compone de diversas *Cadenas* —*Chains*— de reacciones, y una reacción puede estar involucrada en diferentes *cadenas*. Por último, las reacciones se relacionan con las moléculas. Una molécula puede ser un reactante de una reacción, puede ser un producto, o puede intervenir indirectamente como catalizador o inhibidor. Las clases *ReactantsCoefficient*, *ProductsCoefficient*, *EnzymesCoefficient* e *InhibitorsCoefficient* heredan todas de la clase *Coefficient*, omitida por motivos de claridad, que contiene un atributo *coefficient* de tipo entero. Estas clases permiten expresar que una molécula interviene con un cierto coeficiente[2] en una reacción, e intentan representar, según la expresividad de Ecore, la semántica de una clase asociación.

La figura 4 muestra el modelo creado para la herramienta CPN Tools. En este caso, se ha decidido un diseño más alejado del diseño conceptual de una red de Petri Coloreada, incluyendo conceptos específicos de la propia herramienta *CPN Tools*. Esto se debe a que un diseño así que permite tratar con todos los conceptos interesantes de la herramienta *CPN Tools* desde EMF (color de los elementos gráficos y arcos, posiciones, etc.). Además, de esta manera el proceso de persistencia al fichero XML final es trivial, siendo una correspondencia 1-a-1.

El modelo —donde la clase raíz es *Cpnet*— se compone de dos grande bloques de clases: aquellas que cuelgan de *Globbox* y aquellas que lo hacen de *Page*. La figura 4 muestra estos dos grande grupos separados mediante una línea discontínua. El primer grupo de clases permite representar las declaraciones de la red (colores —*Color sets*—, *variables*, *bloques*, etc.). La herramienta permite definir distintos tipos de colores (*Color Sets*). De todos los tipos que proporciona, por simplicidad (ya que son los únicos necesarios) sólo se han incluido el *Color Set* simple «*Enumerated*» y el *Color Set* compuesto «*Product*».

El segundo grupo de clases (aquellas contenidas en el elemento *Page*) representan a todos los elementos visuales de la red de Petri coloreada. Estos elementos visuales heredan todos de la clase *DiagramElement*, y además, pueden estar contenidos en distintos grupos —*Group*—. Así, dentro de una *página* podemos encontrar lugares —*Places*—, transiciones —*Trans*—, arcos —*Arcs*—, anotaciones —*Annot*—, etc. Por su parte, se dice que los *lugares* que se han definido son de un tipo (co-

[2] El coeficiente representa el número de moléculas que aparecen en la ecuación de una reacción. Por ejemplo, en la reacción $2H_2 + O_2 = 2\ H_2O$, los coeficientes son los números que aparecen a la izquierda de las móleculas H_2 y H_2O.

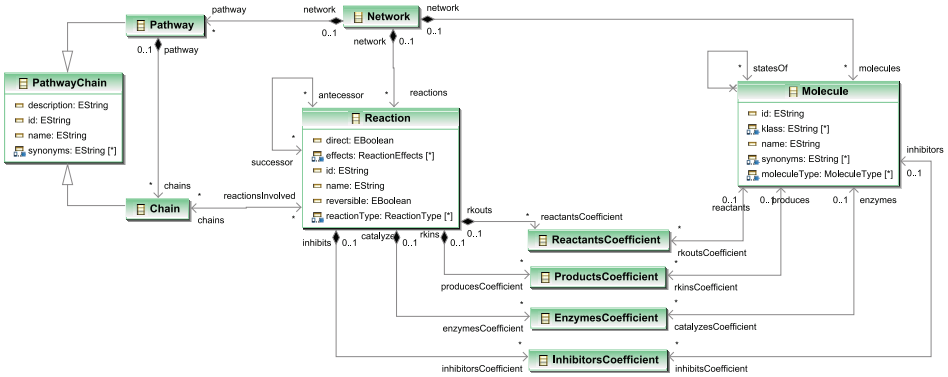


Figura 3: Modelo de la base de datos Transpath®.

lor) que debe haber sido definido en las declaraciones (mediante el rol *type* desde la clase *Place* a la clase *ColorSet*). Las clases *InitMark* y *Mark* permiten representar el estado de un determinado *lugar*, indicando los tokens que se encuentran en éste. El tipo de estos tokens viene dado por el rol *colorSetElement* entre las clases *Mark* y *ColorSetElement*.

4.3. Proceso de transformación.

Finalmente, se han definido las reglas de transformación que permiten convertir los datos del dominio origen al dominio destino. Estas reglas, expresan las correspondencias establecidas por los biólogos —cuando construyen una red de Petri coloreada de forma manual— entre los datos de TRANSPATH® y las primitivas de la herramienta *CPN Tools*. La tabla 1 muestra de forma simplificada estas correspondencias entre el dominio origen y el dominio destino.

El lenguaje con el que se expresan las relaciones entre ambos dominios es QVT-Relations, que se describe de forma breve a continuación. En QVT-Relations, una transformación son un conjunto de relaciones establecidas entre los dominios participantes en la transformación que deben cumplirse para que ésta sea satisfactoria [9]. Un dominio es una variable tipada que puede corresponderse con algún elemento del modelo que va a transformarse. Éste puede tener un *patrón*, que puede considerarse como un conjunto de restricciones que deben cumplir los elementos del modelo candidato —modelo sobre el que se aplica la transformación— para que se trate de una correspondencia válida.

Transpath	CPN Tools
Network	Cpnet
Pathway	Globbox Page
Molecule (complex)	Product
Molecule (simple)	Enumerated
Reaction	Trans
Molecule (reactant)	Place Arc (de Place a Trans)
Molecule (product)	Place Arc (de Trans a Place)

Tabla 1: Correspondencias entre el dominio origen y el dominio destino.

Los dominios además, pueden caracterizarse mediante el uso de las palabras clave `checkonly` y `enforce`. Para un dominio `checkonly`, la transformación comprobará que existe una correspondencia válida —que satisfaga el patrón del dominio— en el modelo candidato. En caso de que el dominio destino sea `enforce`, si al ejecutar la transformación no existe ninguna correspondencia posible, se creará un elemento que cumpla con el patrón del dominio.

La *relación ReactantsToPlaces*, muestra un ejemplo de la sintaxis para la definición de *relaciones* y *dominios*. Esta *relación* establece la correspondencia entre una *molécula* y un *lugar*. Encontramos dos dominios: *tpDo-*

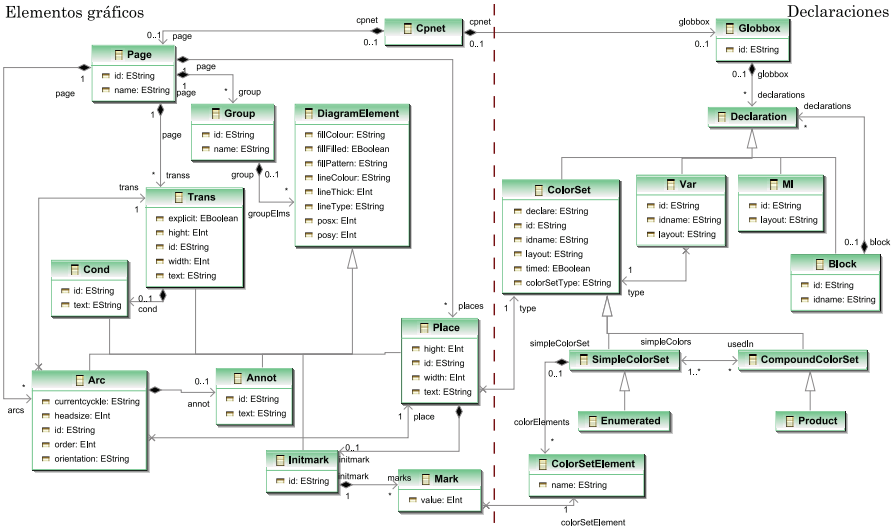


Figura 4: Modelo de la herramienta CPNTools.

main se corresponde con el dominio origen (TRANSPATH®) y *cpnDomain* con el dominio destino (*CPN Tools*). Para el dominio *tpDomain* se comprueba que exista una molécula con un atributo, `name`, de tipo *String*. Para cada molécula que cumpla este patrón deberá existir (y si no existe, se creará) un *Place* en el dominio destino (*cpnDomain*), cuyo atributo `id` contenga el nombre de dicha molécula.

```
relation ReactantsToPlaces {
  molecName1 : String;
  checkonly domain tpDomain molecule1 : Molecule
    {  name = molecName1  };

  enforce domain cpnDomain place1 : Place
    {  id = molecName1  };
  //...
  when
    {  IsSimpleMolecule(molecule1);  }
  where
    {  ReactantsToArcs(molecule1,place1,...);  }
}
```

Regla 1: Regla *ReactantsToPlaces*.

A la aplicación de una regla pueden añadírsele pre- y post-condiciones mediante el uso de las cláusulas `when` y `where`. Por ejemplo, la regla *ReactantsToPlaces* sólo se aplicará en caso de que la molécula sea de tipo simple —IsSimpleMolecule(...) es una función que mediante una expresión OCL comprueba si la molécula es simple o compuesta—. La cláusula `where` establece que tras la aplicación de la *relation* se procederá a la ejecución de la regla *ReactantsToArcs*.

El proceso de transformación completo se realiza de arriba a abajo, navegando el modelo origen a través de las relaciones de contención, definidas como asociaciones de composición. Esto es, se parte del elemento raíz del modelo origen (*Network*), y se navega hacia abajo (*Network* → *Pathways* → *Chains* → *Reactions* → *Coefficients* → *Molecules*) creando en el modelo destino los elementos correspondientes, tal y como expresa de forma resumida la tabla 1.

Para las declaraciones, la transformación creará un *ColorSet* de tipo *Enumerated* a partir de cada molécula simple. A partir de las moléculas complejas se crearán los *ColorSet* de tipo *Product*, y que estarán formados por los *Enumerated ColorSets* correspondientes a las moléculas simples que los forman.

Para los elementos visuales se parte de un objeto de tipo *Reaction*. Para cada reacción, se crea un objeto de tipo *Trans*. Navegando el rol *reactantsCoefficient* de la clase *Reaction* se obtienen los reactantes. Para cada uno de ellos, se crea un objeto *Place*. Por último ya es posible enlazar cada *place* recién creado con el

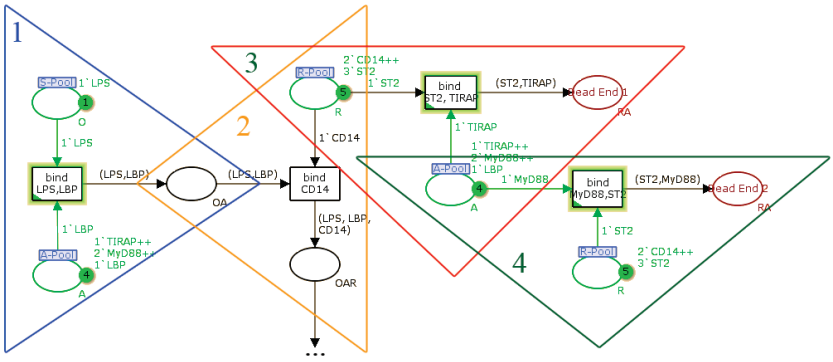


Figura 5: Representación parcial del Pathway del TLR4 en *CPN Tools*.

elemento *Trans* correspondiente mediante un arco —*Arc*— de tipo *PtoT* (*Place to trans*, según la terminología de *CPN Tools*). Para los productos de la reacción se procede de forma similar, en este caso navegando el rol *productsCoefficient*. Como resultado del proceso de transformación podemos observar la figura 5, que muestra la representación de las reacciones del caso de estudio en *CPN Tools*. En la figura se pueden observar cuatro triángulos numerados. Cada triángulo encierra los elementos generados para la reacción identificada con el mismo número. De esta manera, para la reacción (1) —LPS + LBP $\leftrightharpoons$ LPS:LBP—, se han generado los elementos dentro del triángulo izquierdo (1). De igual manera ocurre con los triángulos 2, 3 y 4, que se corresponden con las consiguientes reacciones.

El código completo de la transformación y los ficheros de ejemplo se encuentran en [6].

5. Conclusiones y trabajos futuros.

En este artículo se ha presentado un caso de estudio en el cual se aborda un problema de interoperabilidad entre aplicaciones en el campo de la bioinformática mediante un enfoque dirigido por modelos. En este entorno es común la existencia de fuentes de datos y herramientas de simulación heterogéneas, y la natural representación de los datos biológicos como modelos permite abordar estos problemas desde un punto de vista más eficiente (acortando el ciclo de vida del desarrollo de software) y más elegante (puesto que las operaciones se realizan con un lenguaje más expresivo dada su naturaleza declarativa).

El trabajo pesenta como ventajas frente a las aproximaciones tradicionales (1) la automatización de tareas que anteriormente se realizaban de forma manual; (2) el desarrollo de herramientas modulares, independizando el mecanismo de transformación del formato de persistencia de los datos y facilitando la extensibilidad y mantenibilidad. (3) Aprovecha las ventajas de las transformaciones de modelos. El uso de modelos para representar los datos a ser transformados permite representar de forma más clara la estructura de éstos, siendo más intuitiva su manipulación ya que se trabaja con conceptos de alto nivel. (4) Se proporcionan capacidades de trazabilidad de forma implícita, ayudando a la localización de información errónea en los orígenes de datos. Por último, (5) el uso de un lenguaje como QVT-Relations ofrece como ventaja (frente a las aproximaciones imperativas tradicionales) que permite expresar de forma declarativa —y por tanto más descriptiva— las correspondencias entre los dominios origen y destino.

Futuras líneas de investigación de esta aproximación dirigida por modelos para el desarrollo de aplicaciones en bioinformática van dirigidas en dos direcciones. En primer lugar, la representación de los datos biológicos como modelos permiten el aprovechamiento de los nuevos *frameworks* para la generación de metáforas visuales a partir de éstos, como es

el caso de *GMF*[4] o *MS DSL Tools*. Por otra parte las investigaciones en ingeniería dirigida por modelos pueden proporcionar un rico *background* tecnológico no únicamente para la transformación de datos de un dominio a otro, sino para tareas como la integración de estos obtenidos desde orígenes heterogéneos.

6. Agradecimientos

Agradecemos al Prof. Dr. H.D. Ehrich y M. Ziegler del *Institut für Informationssyeteme* de la *Technische Universität Carolo-Wilhelmina zu Braunschweig*, su apoyo, colaboración y experiencia aportados en el desarrollo de este trabajo. Igualmente agradecemos su duro trabajo a Pascual Queralt como soporte en el uso de MOMENT-QVT.

Referencias

[1] A. Boronat, J. Iborra, J. Ángel Carsí, I. Ramos, and A. Gómez. Del método formal a la aplicación industrial en gestión de modelos: Maude aplicado a eclipse modeling framework. *Revista IEEE América Latina*, September 2005.

[2] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[3] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[4] Eclipse Organization. The graphical modeling framework, 2006. `http://www.eclipse.org/gmf/`.

[5] EMF. `http://download.eclipse.org/tools/emf/scripts/home.php`.

[6] A. Gómez. Ficheros de ejemplo para el caso de estudio, 2007. `http://moment.dsic.upv.es/jisbd07/files/`.

[7] M. Heiner, I. Koch, and K. Voss. Analysis and simulation of steady states in metabolic pathways with petri nets, Aug. 02 2001.

[8] Object Management Group. MDA Guide Version 1.0.1. 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[9] Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005. `http://www.omg.org/cgi-bin/doc?ptc/2005-11-01`.

[10] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[11] Regev, Panina, Silverman, Cardelli, and Shapiro. Bioambients: An abstraction for biological compartments. *TCS: Theoretical Computer Science*, 325, 2004.

[12] A. Regev, W. Silverman, and E. Y. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.

[13] C. Taubner, B. Mathiak, A. Kupfer, N. Fleischer, and S. Eckstein. Modelling and simulation of the TLR4 pathway with coloured petri nets. *2006. EMBS '06. 28th Annual International Conference of the IEEE*, pages 2009–2012, August 2006.

[14] C. Taubner and T. Merker. Discrete modelling of the ethylene-pathway. In *ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops*, page 1152, Washington, DC, USA, 2005. IEEE Computer Society.

[15] M. Ziegler. Implementierung eines Transformationsmoduls in Java zur Abbildung von Signaltransduktions-Instanzen auf CPN-Instanzen. Master's thesis, Technische Universität Braunschweig, Braunschweig, Januar 2007.

Biological Data Transformation in Pathway Simulation*

Abel Gómez,[1] Artur Boronat,[1] Jose Á. Carsí,[1] and Isidro Ramos[1]

[1]*Departament de Sistemes Informàtics i Computació. Universitat Politècnica de València. Camino de Vera, s/n. 46022 València. España.*

This work shows how Model-Driven Software Development (MDSD) can be applied in the bioinformatics field since biological data structures can be easily expressed by means of models. The existence of several heterogeneous data sources is usual in the bioinformatics context. In order to validate the information stored in these data sources, several formalisms and simulation tools have been adopted. The process of importing data from the source databases and introducing it in the simulation tools is usually done by hand. This work describes how to overcome this drawback by applying MDSD techniques (e.g. model transformations). Such techniques allow us to automate the data migration process between source databases and simulation tools, making the transformation process independent of the data persistence format, obtaining more modular tools and generating traceability information automatically.

MOTIVATION

The traditional sequence of "experiment → analysis → publication" is changing to "experiment → data organization → analysis → publication" [6]. This is because, nowadays, data is not only obtained from experiments, but also from simulations. The great amount of new data that can be generated from these experiments is not always homogeneous and may be stored in different databases.

This scenario is found especially when analyzing and simulating cell-signaling mechanisms (*Signal Transduction Pathways*). In studies of this type, it is very common to find both independent databases and modeling tools. Thus, the data of the databases must be converted from the source databases to the simulation tools in order to be used.

Model-Driven Software Development (MDSD) is an approach that attempts to solve problems of this kind. In MDSD, a model is a data structure that can be defined by means of a modeling language (usually called *metamodel*). Using models in a MDSD process allows the automation of the development and evolution of the software applications thanks to generative programming techniques [5] such as model transformations and code generation.

Dealing with data from the MDSD perspective helps to develop tools where the data processing mechanisms are independent of the final persistence format, obtaining more modular tools. This also helps to automate the data migration process by means of model transformation techniques. All these factors reduce the costs of the software development process, directly increasing the productivity of the users/biologists.

BACKGROUND

In organisms, proteins have a wide variety of functions and they interact with each other in similar multifaceted ways. These interactions of proteins are described by means of signal transduction pathways or networks, which are typically represented as certain kinds of maps.

* This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation META TIN2006-15175-C05-01 and the *Generalitat Valenciana* ref. ACOMP07/216

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008.

These signal transduction pathways are composed by experts, who study the relevant literature that is produced by various groups worldwide doing research on very small parts of signal transduction pathways in different kind of organisms. This information is then composed bottom-up to a signal transduction pathway and introduced into databases to provide an integrated view on the entire pathway.

Examples for such signal transduction pathway databases are TRANSPATH® [11], KEGG [10] or Reactome [9]. They usually provide a web interface for interactive searches and also make their data available as text files in flat file or XML format.

Understanding the flow of information inside a cell is fundamental for an in-depth understanding of the functioning of a cell as a whole. Therefore, modeling and simulating this information flow is beneficial because it helps to understand the flow of signals in a complex network, to test hypotheses in silico before validating them with experiments, and to validate the data collected about a certain signal transduction pathway.

We are currently working on one of the major signal transduction pathways databases, TRANSPATH® [11], and we are using Colored Petri Nets [7] (among others) as the specification language. The corresponding simulation tool is CPN Tools [8]. TRANSPATH® is a database that is accessible by means of the usual methods, i.e., web interface, text files, XML (using its own XML format), etc. Coloured Petri nets are a formal representation for distributed discrete systems that allow concurrent events to be represented.

In [15], data is extracted from the TRANSPATH® database and introduced in the *CPN Tools* application manually. This implies that the user/biologist who is going to perform the simulation must manually query the database to extract the list of reactions involved in the signal transduction *pathway* to be studied. With the extracted data the corresponding Petri net must be built in the simulation tool manually.

TECHNIQUES AND TOOLS

Model-Driven Engineering (MDE) is a Software Engineering field that over the years has represented software artifacts as models in order to increase productivity, quality and to reduce costs in the software development process. Nowadays, there is increasing interest in this field, as demonstrated by the OMG guidelines that support this trend with the Model-Driven Architecture (MDA [12]) approach. Model-Driven Software Development (MDSD) has evolved from Model-Driven Engineering. MDSD not only involves design and code generation tasks, but also traceability capabilities, meta-modeling features, model persistence and model interchange tasks, etc. To address these tasks, operations between models, transformations, and queries over these models are relevant problems that must be resolved. In the MDA context, they are resolved from the open standards point of view. The Meta Object Facility (MOF) standard [14] provides support for the meta-modeling capabilities. The Query/Views/Transformations (QVT [13]) standard describes how to provide support to queries and transformations. In contrast to other new languages, QVT uses the pre-existent *Object Constraint Language* (OCL) language to perform queries over software artifacts.
MOMENT [3] is a tool that provides support to the OMG standards giving capabilities to transform models. The tool uses both an industrial modeling front-end and an algebraic back-end for the execution of the transformation and query tasks. The algebraic background runs in the high performance rewriting system Maude [4]. The industrial modeling environment used by MOMENT is the Eclipse Modeling Framework (EMF). EMF [2] can be considered as an implementation of the MOF standard and can import software artifacts from several heterogeneous data sources: UML models, XML Schemas, etc. The tool offers an implementation of the QVT-Relations language as well as the OCL language. QVT-Relations is a declarative transformations language that provides

implicit traceability capabilities. For this language, MOMENT gives wide support for unidirectional transformations. Moreover, the tools provides full support to the query operators of the OCL language.

A MDSD APPROACH IN BIOLOGICAL DATA MIGRATION

In the initial work on the study of the TLR4 signal transduction *pathway*, data migration from the source database to the simulation tool (to represent this information as a coloured Petri net) was done manually.

The solution to the data migration problem is described as follows by means of model transformation techniques using the model-driven software development guides. This implies the following tasks: (a) development of the source domain data model (TRANSPATH®), (b) development of the target domain data model (*CPN Tools*), (c) definition of the transformation rules between the source domain and the target domain by means of the transformations language, (d) implementation of the pre-processing mechanism to obtain the instances of the source model from the original data; and finally, (e) definition of the post-processing tasks that persist the transformed data in the final file format.

Architecture and overview of the tool

The data migration process is performed in three steps: (1) recovering and pre-processing of the input data, (2) execution of the transformation by means of the transformations engine and (3) post-processing and persistence of the result data. In a MDSD approach, using a transformation engine implies that the source and the target models of the transformation must be developed in the first place to be able to establish the mappings between the two domains.

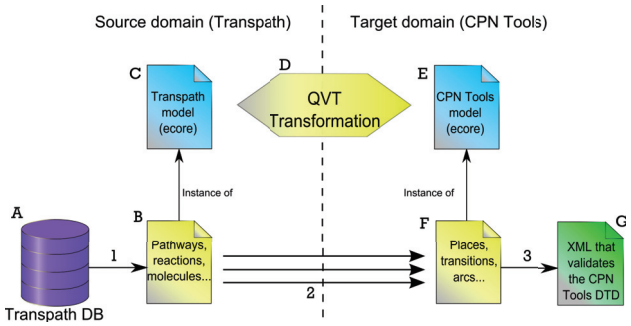


FIG. 1: Architecture of the tool.

Figure 1 shows the architecture of the tool. It represents the three steps that are needed to perform the data migration. First (1), the data is extracted from the TRANSPATH® database (A), and the corresponding XMI instance (B) of the TRANSPATH® Ecore model (C) is built.

The second step (2) is the most important and complex one of the transformation process. It is performed by means of the MOMENT tool and its transformation engine. It executes the transformation from the TRANSPATH® domain (reactions, molecules, etc.) to the *CPN Tools* domain (places, transitions, arcs, etc.). After the definition of the transformation rules (D) between the source domain (C) and the target domain (E), the transformation is executed over the data recovered from the database (B) obtaining the needed information in the *CPN Tools* domain (F).

Finally, the third step (3) in the data migration process is again a trivial process in which the EMF data is stored in an XML file readable from the *CPN Tools* application (G).

Transformation process

The transformation rules that can convert data from the source domain to the target domain have been defined in a declarative way. These rules express the mappings established by biologists between the data extracted from the TRANSPATH® database and the concepts available in the *CPN Tools* application. Table I shows the simplified mappings between both the source and the target domain. The rules that define the direct relationships between the two domains have been expressed in QVT-Relations. In this language, a transformation is a set of *relations* established between the domains that participate in that transformations that must hold for the transformation to be successful [13].

Transpath	**CPN Tools**
Network	Cpnet
Pathway	Globbox – Page
Molecule (complex)	Product
Molecule (simple)	Enumerated
Reaction	Trans
Molecule (reactant)	Place – Arc (from Place to Trans)
Molecule (product)	Place – Arc (from Trans to Place)

TABLE I: Mappings between the source and the target domain.

CONCLUSIONS AND FUTURE WORK

This work has presented a case study where the interoperability problem between applications is addressed using a model-driven approach. The situation where several data sources and simulation tools co-exist and must share heterogeneous data is very common in the bioinformatics field. In this situation, the easy representation of biological data using models allows us to deal with these problems more efficiently and more elegantly than the traditional (manual) approaches.

This work presents the following advantages over the traditional approaches: (1) It allows some tasks that were previously done by hand to be automated. (2) This approach produces more modular tools, making the transformation mechanism independent from the data persistence format, improving the extensibility and maintainability of these tools. (3) Biologists do not need to know technical details about the migration process, which increases their productivity. (4) It also takes advantage of model transformation technologies. Using models to represent the data to be transformed permits the data structure to be more clearly represented making its manipulation more intuitive since it deals with high-level concepts. (5) Traceability capabilities are provided implicitly. These capabilities help to locate invalid information in the data sources. Finally, (6) using languages such as QVT-Relations offers the advantage of expressing the mappings between the source and the target domains in a declarative way. This way of representing the relationships between the two domains is more expressive than the traditional and imperative approaches.

With our case-study we presented the first steps in using model-driven techniques in the live science. Further research are focussed in different goals. First, using models to represent biological data allows us to take advantage of the new *frameworks* such as *GMF*[1] or *MS DSL Tools*. These tools use models to automatically generate visual metaphors. Second, the research done in Model-Driven Engineering can provide a rich background not only for data transformation between two different domains, but also for other tasks such as heterogeneous data integration.

ACKNOWLEDGEMENTS

We would like to thank Claudia Täubner, Dr. Silke Eckstein and Prof. Hans-Dieter Ehrich from the Institut für Informationssysteme (Technische Universität Braunschweig) for its collaboration and support during this work. We would also like to thank P. Queralt for his valuable knowledge on the use of the MOMENT-QVT tool.

[1] Gmf. http://www.eclipse.org/gmf/.

[2] EMF. http://www.eclipse.org/emf/.

[3] A. Boronat, J. Iborra, J. Ángel Carsí, I. Ramos, and A. Gómez. Del método formal a la aplicación industrial en gestión de modelos: Maude aplicado a eclipse modeling framework. *Revista IEEE América Latina*, September 2005.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[5] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[6] S. R. S. A. e. a. Emmett, S. Towards 2020 science. Technical report, Microsoft Corporation, 2006. http://research.microsoft.com/towards2020science/downloads/T2020S_ReportA4.pdf.

[7] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*. Springer, Berlin, 2nd edition, 1997.

[8] K. Jensen, L. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *Int. J. on Software Tools for Technology Transfer (STTT)*, Sp. Sec. CPN 04/05, 2007.

[9] G. Joshi-Tope, M. Gillespie, I. Vastrik, P. D'Eustachio, E. Schmidt, B. de Bono, B. Jassal, G. R. Gopinath, G. R. Wu, L. Matthews, S. Lewis, E. Birney, and L. Stein. Reactome: a knowledgebase of biological pathways. *Nucleic Acids Research*, 33(suppl_1):D428–432, 2005.

[10] M. Kanehisa, S. Goto, M. Hattori, K. Aoki-Kinoshita, M. Itoh, S. Kawashima, T. Katayama, M. Araki, and M. Hirakawa. From genomics to chemical genomics: new developments in KEGG. *Nucleic Acids Research*, 34(suppl_1), 2006.

[11] M. Krull, S. Pistor, N. Voss, A. Kel, I. Reuter, D. Kronenberg, H. Michael, K. Schwarzer, A. Potapov, C. Choi, O. Kel-Margoulis, and E. Wingender. TRANSPATH(R): an information resource for storing and visualizing signaling pathways and their pathological aberrations. *Nucleic Acids Research*, 34(suppl_1):D546–551, 2006.

[12] Object Management Group. MDA Guide Version 1.0.1. 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[13] Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[14] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[15] C. Taubner, B. Mathiak, A. Kupfer, N. Fleischer, and S. Eckstein. Modelling and simulation of the TLR4 pathway with coloured petri nets. *Engineering in Medicine and Biology Society, 2006. EMBS '06. 28th Annual International Conference of the IEEE*, pages 2009–2012, August 2006.

JISBD2007-03: Biological Data Processing using Model Driven Engineerings

A. Gómez, A. Boronat, J.Á. Carsí, I. Ramos, C. Täubner y S. Eckstein

***Abstract*— This paper shows how Model-Driven Software Development (MDSD) can be applied in the bioinformatics field since biological data structures can be easily expressed by means of models. The existence of several heterogeneous data sources is usual in the bioinformatics context. In order to validate the information stored in these data sources, several formalisms and simulation tools have been adopted. The process of importing data from the source databases and introducing it in the simulation tools is usually done by hand. This work describes how to overcome this drawback by applying MDSD techniques (e.g. model transformations). Such techniques allow us to automate the data migration process between source databases and simulation tools, making the transformation process independent of the data persistence format, obtaining more modular tools and generating traceability information automatically.**

***Keywords*— Bioinformatics, Model Driven Software Development -MDSD-, data migration, biological systems modeling.**

I. Motivación.

La práctica tradicional en los estudios científicos de «experimentación → análisis → publicación» está cambiando a «experimentación → organización de datos → análisis → publicación». Esto se debe a que, hoy en día, los datos no se obtienen únicamente de experimentos, sino que también se obtienen de simulaciones. Esta gran cantidad de datos generados no siempre tiene la misma estructura y puede estar almacenada en bases de datos heterogéneas. Además, el elevado número de datos requiere el desarrollo de herramientas informáticas que nos permitan representarlos para analizarlos, y a su vez, realizar otras simulaciones con ellos.

Este artículo ha sido financiado por el Proyecto Nacional de Investigación Científica, Desarrollo e Innovación META TIN2006-15175-C05-01, la ayuda complementaria de la Generalitat Valenciana ref. ACOMP07/216, y a la beca del programa de Formación de Profesorado Universitario, ref. AP-2006-00690.

A. Gómez, J. Á. Carsí, e I. Ramos pertenecen al Departamento de Sistemas Informáticos y Computación en la Universidad Politécnica de Valencia, Cno. de Vera s/n, 46022 Valencia. (correos e.: agomez@dsic.upv.es, pcarsi@dsic.upv.es, iramos@dsic.upv.es).

A. Boronat pertenece en la actualidad al *Department of Computer Science*, F28 Mathematics & Computer Science Building, University of Leicester, University Road, Leicester, LE1 7RH, United Kingdom. (correo e.: aboronat@mcs.le.ac.uk).

C. Täubner y S. Eckstein pertenecen al Institut für Informationssysteme, Technische Universität Braunschweig, Mühlenpfordtstraße 23, 2.OG, D-38106 Braunschweig, Deutschland. (correo e.: c.taeubner@tu-bs.de, s.eckstein@tu-bs.de).

En el campo de la bioinformática encontramos esta problemática en el análisis y simulación de los mecanismos de señales de las células (*pathways*). Un *pathway* es un conjunto de reacciones químicas que se producen en el interior de una célula tras la recepción de un estímulo. En estudios de este tipo se observa tanto el rellenado independiente de fuentes de datos como el desarrollo independiente de herramientas de modelado. Por ello, estos datos deben ser reconvertidos de forma manual para que puedan ser utilizados en las herramientas de simulación. En un escenario así, disponer de herramientas interoperables es deseable.

El Desarrollo de Software Dirigido por Modelos — DSDM— es una aproximación que pretende dar solución a problemas como éste. En el DSDM, un modelo es una estructura de datos que puede ser definida mediante un lenguaje de modelado (generalmente llamado metamodelo). Un modelo permite definir la funcionalidad, estructura y/o comportamiento de un sistema [1] dependiendo del metamodelo utilizado. La utilización de modelos en un proceso de DSDM permite automatizar el desarrollo de aplicaciones y su evolución mediante técnicas de programación generativa [2], como las transformaciones de modelos y la generación de código.

Este artículo muestra cómo la filosofía de DSDM permite resolver los problemas surgidos en el estudio de los caminos de señales en el campo de la bioinformática. Los problemas de interoperabilidad entre aplicaciones pueden abordarse de una manera sistemática, donde (a) el formato de datos puede definirse como un modelo y (b) los datos se definen como una colección de objetos que son instancia de las clases de este modelo. El tratamiento de los datos desde la perspectiva del DSDM permite (c) el desarrollo de herramientas donde el tratamiento de éstos sea independiente de los formatos de persistencia, obteniendo herramientas más modulares. Esto a su vez facilita (d) la automatización en la migración de los datos mediante el uso de técnicas de transformaciones de modelos. Todos estos factores reducen el coste de producción de las herramientas afectando de forma directa y positiva en la productividad de los usuarios/biólogos.

El documento se organiza de la siguiente manera: en el apartado II se expone el contexto biológico, introduciendo las razones por las que es interesante el estudio de los caminos de señales (*pathways*), y se describe la aproximación actual, en la que esto es un proceso poco eficiente. El apartado III

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008.

introduce las bases del desarrollo dirigido por modelos y la tecnología empleada para llevar a cabo la migración de datos. En el apartado IV se describe la solución diseñada para resolver las deficiencias de la aproximación actual, y por último en el apartado V se exponen las conclusiones finales y trabajos futuros.

II. Caso de estudio

Numerosos procesos biológicos son controlados por los denominados mecanismos de señales. Un camino de señales —*signaling pathway*— es una colección de reacciones químicas que tienen lugar en una célula como reacción a un estímulo[1] (generalmente externo) disparando diversos eventos en el interior de ésta. Estos eventos son los que controlan la progresión del ciclo de vida (muerte, crecimiento, especialización), activación o desactivación de genes para el momento de la reproducción, el movimiento de la célula, etc. El estudio de estos caminos de señales se realiza mediante el ensayo empírico de los estímulos externos que recibe una célula y recogiendo los datos sobre las reacciones químicas desencadenadas en su interior. Estos datos obtenidos de forma experimental por los biólogos son almacenados en diferentes bases de datos para su posterior estudio y consulta por otros expertos. Existen diversas fuentes donde es almacenada la información como por ejemplo TRANSPATH®, BioCarta, KEGG o Reactome.

Las reacciones químicas involucradas en un *pathway* suelen ser un proceso en cascada, produciéndose entonces reacciones de forma concurrente. Por ello, para el modelado y análisis de estos procesos biológicos se han adaptado y aplicado algunos de los formalismos desarrollados para el análisis de sistemas concurrentes. Ejemplo de ello son aproximaciones en π-cálculo [3], ambient calculus [3], *Life Sequence Charts* (LSCs) [5] o redes de Petri [6].

El caso de estudio parte de los trabajos realizados por Taübner et al. [7], donde se ha modelado, simulado y validado el *pathway* del receptor de tipo toll 4 (TLR4) —que se introduce en el siguiente punto— mediante redes de Petri coloreadas. A continuación se expone en primer lugar el contexto biológico describiendo el problema abordado en [7] y en segundo lugar se describen los detalles tecnológicos de éste: fuente de datos, herramienta de simulación seleccionadas y proceso de extracción y tratamiento de los datos.

A. Los receptores de tipo Toll. El pathway TLR4.

Los receptores de tipo Toll —*Toll-like receptors* (TLR)— son una familia de proteínas que forman parte del sistema inmunitario. Permiten la adaptación del sistema inmune de diversos seres vivos siendo responsables del reconocimiento de diversos patrones (secuencias de moléculas) que identifican a diferentes patógenos. Una vez reconocido el patógeno (cualquier entidad biológica capaz de producir daño) estimulan la respuesta de defensa contra él por parte del sistema inmunitario.

[1] Un estímulo puede ser, por ejemplo, la presencia de una determinada molécula en el entorno de la célula.

Actualmente se han identificado trece receptores de tipo Toll en humanos (TLR1-TLR13). Estos receptores reconocen moléculas que están constantemente asociadas a amenazas contra el sistema inmune y que son muy específicas a estas amenazas, no pudiendo confundirlas con moléculas propias. Algunas de estas moléculas específicas pueden ser los lipopolisacáridos (LPS) presentes en la superficie de células bacterianas, proteínas presentes en los flagelos de bacterias, ARN de doble cadena presente en virus, etc.

Para el caso de estudio, por simplicidad, se toma como ejemplo una pequeña parte del *pathway* en el que interviene el receptor de tipo Toll 4, por ser el primero reconocido en mamíferos y el mejor estudiado hasta el momento. A continuación se indican las reacciones que serán transformadas en el caso de estudio.

1. LPS + LBP $\rightleftharpoons$ LPS:LBP
2. LPS:LBP + CD14 $\rightleftharpoons$ LPS:LBP:CD14
3. ST2 + TIRAP $\rightleftharpoons$ ST2:TIRAP
4. ST2 + MyD88 $\rightleftharpoons$ ST2:MyD88

La notación expresa a la izquierda los reactivos y a la derecha los productos. El símbolo «$\rightleftharpoons$» expresa que esta reacción es bidireccional, ya que se trata de una reacción de equilibrio químico —informalmente se puede describir como una reacción que se mantiene hasta que los reactivos y los productos alcanzan un estado de equilibrio, no llegándose a consumir ninguno de los dos—. Para (1), por ejemplo, podríamos expresar de forma sencilla lo siguiente: la mólecula LPS se une a la molécula LBP, dando lugar a la molécula compuesta LPS:LBP.

B. Aproximación actual en el estudio del pathway del TLR4 en bioinformática.

Son numerosas las fuentes de datos de las que se puede extraer información sobre el *pathway* del TLR4. Algunas de ellas pueden ser TRANSPATH®, KEGG, Reactome o BioCarta. Estas bases de datos han sido rellenadas a partir de estudios empíricos y pueden contener información incompleta e incluso contradictoria. Los actuales trabajos para la representación y simulación de *pathways* se dirigen en la dirección de poder validar la información obtenida en los estudios empíricos.

Con ese objetivo se han adaptado diversos formalismos gráficos para la simulación de *pathways* biológicos. Ejemplos de ello son los diagramas de estados [3], los *Life Sequence Charts* [3][5], o las redes de Petri coloreadas [7]. El trabajo en el que se fundamenta la solución aquí presentada describe un proceso mediante el cual es posible extraer los datos del *pathway* TLR4 de la base de datos TRANSPATH® con el objetivo de simularlo mediante la herramienta de simulación y validación de redes de Petri coloreadas *CPN Tools*. TRANSPATH® es una base de datos que almacena información acerca de caminos de señales. Permite la exportación de datos en XML, y en su versión 7.4 contiene información sobre 62.549 moléculas, 23.076 genes y 104.362 reacciones. Una red de Petri, por su parte, es una representación formal para un sistema distribuido discreto,

que permite representar eventos concurrentes. La red se forma por nodos (llamados lugares), transiciones y arcos dirigidos. Estos arcos conectan siempre un lugar con una transición o una transición con un lugar. En los lugares puede haber un determinado número de *tokens*, que pueden moverse de un lugar a otro cuando una transición se dispara (una transición se habilita cuando todas sus entradas contienen *tokens*). La Fig. 1 muestra un ejemplo de red de Petri, donde los círculos blancos son los lugares, el rectángulo relleno de color negro es una transición, las flechas son los arcos dirigidos, y los puntos negros, los *tokens*.

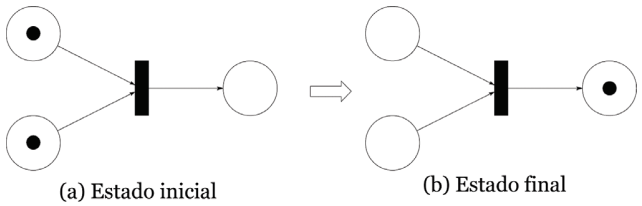


Fig. 1. Ejemplo de red de Petri.

Una red de Petri coloreada es aquella en la que los *tokens* pueden tener un determinado color (esto es, algún atributo distintivo), pudiendo caracterizar *tokens* de diferentes tipos. *CPN Tools* es una herramienta que permite la construcción, modificación y validación sintáctica de redes de Petri coloreadas de forma gráfica. Dispone además de un simulador (tanto interactivo como automático) para poder inspeccionar la evolución de estos modelos.

En [7] el método empleado para poder realizar la simulación a partir de los datos almacenados en TRANSPATH® es manual. Esto supone que el usuario/biólogo que va a realizar la simulación debe consultar de forma manual la base de datos obteniendo el listado de reacciones que intervienen en el *pathway* que desea estudiar. Con los datos obtenidos debe crear y editar de forma manual la correspondiente red de Petri coloreada en *CPN Tools*, creando uno a uno cada uno de los lugares, transiciones, arcos, *tokens*, etc. En el trabajo al que se hace referencia, esto supuso crear de forma manual —y únicamente para el ejemplo del *pathway* del TLR4— 75 lugares, 47 transiciones y casi un centenar de colores.

III. Ingeniería Dirigida por Modelos

La Ingeniería Dirigida por Modelos es un campo en la Ingeniería del Software que, durante años, ha representado los artefactos software como modelos con el objetivo de incrementar la productividad, calidad, y reducir los gastos en el proceso de desarrollo de software. Recientemente, existe un interés creciente en este campo. Prueba de ello es la aproximación de *Model Driven Architecture* [1], apoyada por el OMG.

El Desarrollo Dirigido por Modelos ha evolucionado del campo de la Ingeniería Dirigida por Modelos. En él, no sólo las tareas de diseño y generación de código están involucradas, sino que también se incluyen las capacidades de trazabilidad, tareas de meta-modelado, intercambio y persistencia de modelos, etc. Para poder abordar estas tareas, las operaciones entre modelos, transformaciones, y consultas sobre ellos son problemas relevantes que deben ser resueltos. En el contexto de MDA se abordan desde el punto de vista de los estándares abiertos. En este caso, el estándar *Meta Object Facility* (MOF) [9], proporciona un mecanismo para definir metamodelos. El estándar *Query/Views/Transformations* (QVT) [10] indica cómo proporcionar soporte tanto para transformaciones como para consultas. A diferencia de otros lenguajes nuevos, QVT se apoya en el ya existente lenguaje *Object Constraint Language* (OCL) para realizar las consultas sobre los artefactos software.

A. MOMENT. Un framework para la Gestión de Modelos.

MOMENT [11] es una herramienta que da soporte a los estándares propuestos por el OMG para dar soporte a transformaciones. La herramienta proporciona un soporte algebraico para las tareas de transformación y consulta de modelos mediante un eficiente sistema de reescritura de términos —*Maude*— y desde un entorno de modelado industrial —*Eclipse Modeling Framework* (EMF)—. EMF [12] puede ser visto como una implementación del estándar MOF, y permite la importación automática de artefactos software desde orígenes de datos heterogéneos: modelos UML, esquemas XML, etc. Respecto a *Maude*, MOMENT aprovecha las capacidades de modularidad y parametrización de este sistema para proporcionar un entorno de transformación y consulta de modelos de forma genérica e independiente de metamodelo.

Como lenguaje para transformaciones MOMENT se apoya sobre el estándar abierto QVT, proporcionando una implementación tanto del lenguaje QVT-Relations como de OCL. QVT-Relations es un lenguaje de transformaciones declarativo que proporciona de forma implícita capacidades de trazabilidad. Para este lenguaje, MOMENT ofrece un amplio soporte para transformaciones unidireccionales 1-a-1. Además, esta herramienta proporciona una completa implementación de los operadores de consulta del lenguaje OCL.

IV. Un enfoque de DSDM en la migración de datos biológicos

En el trabajo inicial sobre el estudio del *pathway* del TLR4, la migración de datos desde la base de datos origen hasta la herramienta de simulación para su representación mediante una red de Petri coloreada debe realizarse de forma manual. A continuación se muestra una solución al problema de migración de datos del caso de estudio empleando transformaciones, según la filosofía de desarrollo de software dirigido por modelos. Esto supone las siguientes tareas: (a) desarrollo del modelo de datos del dominio origen (TRANSPATH®) y (b) desarrollo del modelo de datos del dominio destino (CPN Tools). (c) Definición mediante un lenguaje de transformaciones las reglas de transformación entre el dominio origen y el dominio destino; (d) implementación del mecanismo de preprocesado de datos

permitiendo reconstruir los datos originales como instancias del modelo origen, y (e) definición del postprocesado de los datos, que implementa el mecanismo de persistencia al formato final.

A continuación se presenta la solución proporcionada. En primer lugar, se describe el proceso de transformación y sus etapas, en segundo lugar se describen los modelos del dominio origen y el modelo destino, y por último, se describe en mayor detalle el proceso de transformación.

A.Arquitectura e implementación de la herramienta.

El proceso de migración de datos, tal y como se deriva de las tareas anteriores, se realiza en tres pasos: (1) recuperación y preprocesado de los datos, (2) ejecución de la transformación mediante un motor de transformaciones y (3) postprocesado y persistencia de los datos. En una aproximación de DSDM, el uso de un motor de transformaciones implica que se deben definir en primer lugar los modelos origen y destino de la transformación para poder establecer las correspondencias entre uno y otro posteriormente.

La solución aquí presentada hace uso de MOMENT. MOMENT es una herramienta integrada en el entorno *Eclipse* que proporciona soporte para transformaciones. Como entorno de modelado, MOMENT emplea el *Eclipse Modeling Framework* (EMF). Este framework proporciona como lenguaje de modelado *Ecore*. Además, emplea como formato de persistencia XMI. EMF permite —entre otros— la creación de modelos *Ecore* a partir de un esquema XSD. No obstante, los modelos *Ecore* obtenidos a partir de esquemas XSD son complejos y no siempre representan la semántica de los datos de forma clara. Por ello, se ha decidido definir de forma manual los modelos origen y destino, teniendo sólo en cuenta aquella información relevante para el caso de estudio.

La Fig. 2 muestra la arquitectura de la herramienta. En ella se reflejan de forma clara los 3 pasos necesarios para realizar la migración de los datos. En primer lugar se extraen los datos de la base de datos TRANSPATH® (A), y se reconstruyen como una instancia en XMI (B) del modelo *Ecore* de TRANSPATH® (C). Este primer paso (1) se realiza de forma sencilla en Java ya que la correspondencia de los datos origen con el modelo en EMF es directa. La implementación de este primer paso de preprocesado de los datos es una adaptación del trabajo realizado en [13].

El segundo paso de la migración (2) es el principal y más complejo. Se realiza mediante MOMENT y su motor de transformaciones, ejecutando la transformación del dominio de TRANSPATH® (reacciones, moléculas, etc.) al dominio de CPN Tools (lugares, transiciones, arcos, etc.). Tras la definición de las reglas de transformación (D) entre los dominios origen (C) y destino (E), se aplica la transformación sobre los datos recuperados de la base de datos (B) obteniendo una instancia en el dominio de CPN Tools (F). Por último, el tercer paso de la migración (3) es de nuevo un proceso trivial en el que se persisten los datos desde EMF (F) a un fichero XML comprensible por la herramienta *CPN Tools* (G). En

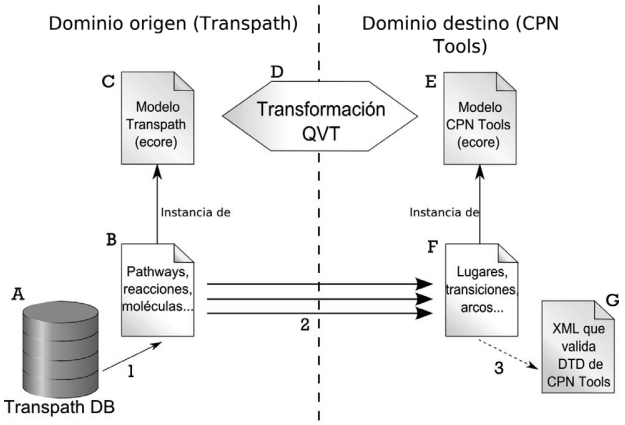


Fig. 2. Arquitectura de la herramienta.

este, paso además, pueden realizarse otras tareas de post-procesado, como por ejemplo, la inclusión de algún algoritmo de redibujado (*layout*) sobre los elementos de la red de Petri.

B.Desarrollo de los modelos.

En primer lugar se ha definido un modelo únicamente con las partes que resultan de interés para la simulación de un *pathway* eliminando los conceptos innecesarios de la compleja base de datos TRANSPATH®. La Fig. 3 muestra —mediante una metáfora visual similar al diagrama de clases de UML— el modelo *Ecore* desarrollado. En el modelo encontramos la clase *Network*, elemento raíz del modelo. Una Red —*Network*— contiene un conjunto de *Pathways*, *Reacciones* y *Moléculas*. A su vez, un *Pathway* se compone de diversas Cadenas —*Chains*— de reacciones, y una reacción puede estar involucrada en diferentes cadenas. Por último, las reacciones se relacionan con las moléculas. Una molécula puede ser un reactante de una reacción, puede ser un producto, o puede intervenir indirectamente como catalizador o inhibidor. Las clases *ReactantsCoefficient*, *ProductsCoefficient*, *EnzymesCo-efficient* e *InhibitorsCoefficient* heredan todas de la clase *Coefficient*, omitida por motivos de claridad, que contiene un atributo *coefficient* de tipo entero. Estas clases permiten expresar que una molécula interviene con un cierto coeficiente[2] en una reacción, e intentan representar, según la expresividad de *Ecore*, la semántica de una clase asociación.

La Fig. 4 muestra el modelo creado para la herramienta *CPN Tools*. En este caso, se ha decidido un diseño más alejado del diseño conceptual de una red de Petri Coloreada, incluyendo conceptos específicos de la propia herramienta *CPN Tools*. Esto se debe a que un diseño así que permite tratar con todos los conceptos interesantes de la herramienta *CPN Tools* desde EMF (color de los elementos gráficos y arcos, posiciones, etc.). Además, de esta manera el proceso de persistencia al fichero XML final es trivial, siendo una correspondencia 1-a-1.

[2] El coeficiente representa el número de moléculas que aparecen en la ecuación de una reacción. Por ejemplo, en la reacción $2H_2 + O_2 = 2H_2O$, los coeficientes son los números que aparecen a la izquierda de las móleculas H_2 y H_2O.

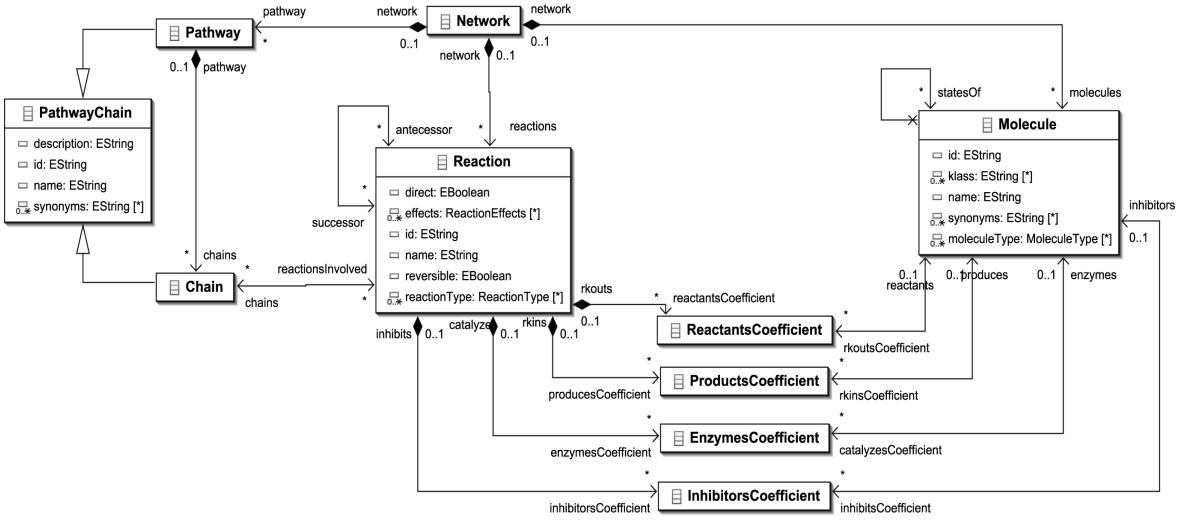


Fig. 3. Modelo de la base de datos Transpath.

El modelo —donde la clase raíz es *Cpnet*— se compone de dos grande bloques de clases: aquellas que cuelgan de *Globbox* y aquellas que lo hacen de *Page*.

La Fig. 4 muestra estos dos grande grupos separados mediante una línea discontinua. El primer grupo de clases permite representar las declaraciones de la red (*colores* —*Color sets*—, *variables*, *bloques*, etc.). La herramienta permite definir distintos tipos de colores (*Color Sets*). De todos los tipos que proporciona, por simplicidad (ya que son los únicos necesarios) sólo se han incluido el *Color Set* simple «*Enumerated*» y el *Color Set* compuesto «*Product*».

El segundo grupo de clases (aquellas contenidas en el elemento *Page*) representan a todos los elementos visuales de la red de Petri coloreada. Estos elementos visuales heredan todos de la clase *DiagramElement*, y además, pueden estar contenidos en distintos grupos —*Group*—. Así, dentro de una página podemos encontrar lugares —*Places*—, transiciones —*Trans*—, arcos —*Arcs*— , anotaciones —*Annot*—, etc. Por su parte, se dice que los lugares que se han definido son de un tipo (color) que debe haber sido definido en las declaraciones (mediante el rol *type* desde la clase *Place* a la clase *ColorSet*). Las clases *InitMark* y *Mark* permiten representar el estado de un determinado lugar, indicando los *tokens* que se encuentran en éste. El tipo de estos *tokens* viene dado por el rol

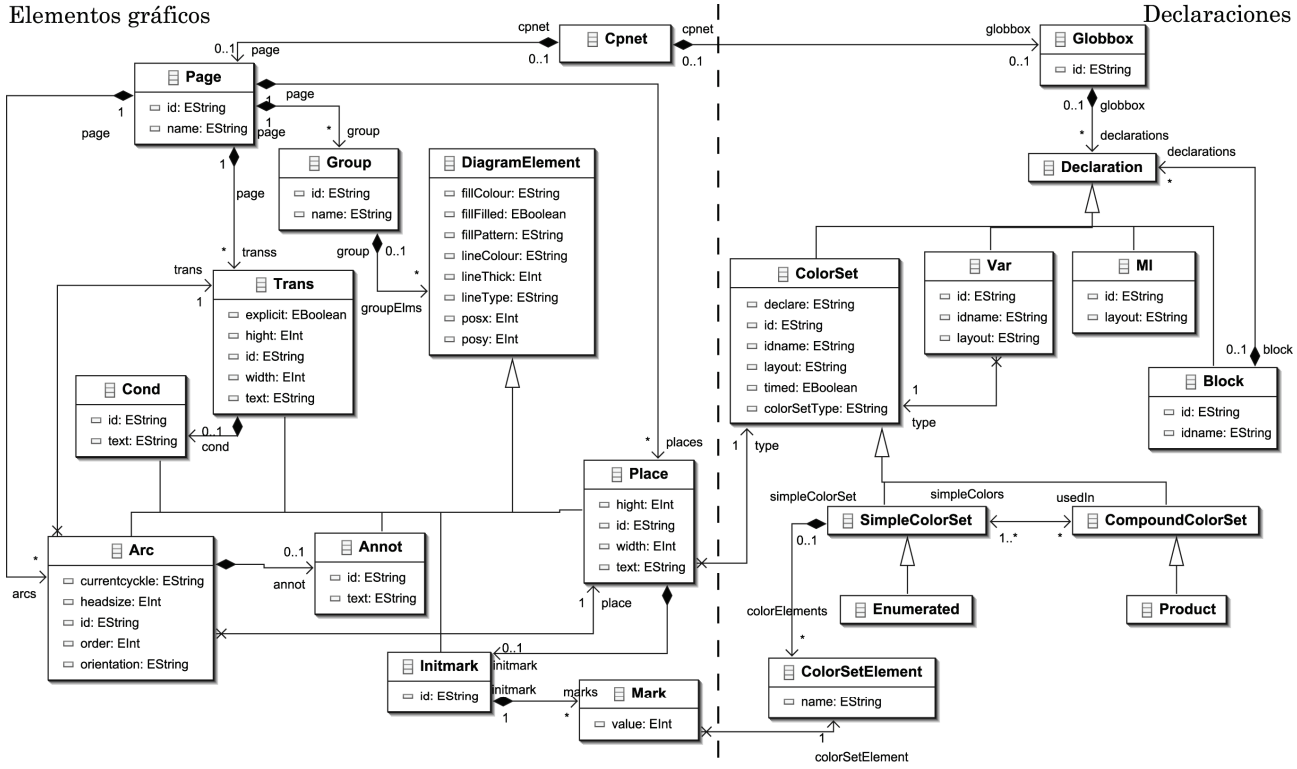


Fig. 4. Modelo de la herramienta CPNTools.

colorSetElement entre las clases *Mark* y *ColorSetElement*.

C.Proceso de transformación.

Finalmente, se han definido las reglas de transformación que permiten convertir los datos del dominio origen al dominio destino. Estas reglas, expresan las correspondencias establecidas por los biólogos —cuando construyen una red de Petri coloreada de forma manual— entre los datos de TRANSPATH® y las primitivas de la herramienta *CPN Tools*. La Tabla I muestra de forma simplificada estas correspondencias entre el dominio origen y el dominio destino.

TABLA I
CORRESPONDENCIAS ENTRE EL DOMINIO ORIGEN Y EL DOMINIO DESTINO.

Transpath	CPN Tools
Network	Cpnet
Pathway	Globbox Page
Molecule (complex)	Product
Molecule (simple)	Enumerated
Reaction	Trans
Molecule (reactant)	Place Arc (de Place a Trans)
Molecule (product)	Place Arc (de Trans a Place)

El lenguaje con el que se expresan las relaciones entre ambos dominios es QVT-Relations, que se describe de forma breve a continuación. En QVT-Relations, una transformación son un conjunto de relaciones establecidas entre los dominios participantes en la transformación que deben cumplirse para que ésta sea satisfactoria [10]. Un dominio es una variable tipada que puede corresponderse con algún elemento del modelo que va a transformarse. Éste puede tener un patrón, que puede considerarse como un conjunto de restricciones que deben cumplir los elementos del modelo candidato —modelo sobre el que se aplica la transformación— para que se trate de una correspondencia válida.

Los dominios además, pueden caracterizarse mediante el uso de las palabras clave `checkonly` y `enforce`. Para un dominio `checkonly`, la transformación comprobará que existe una correspondencia válida —que satisfaga el patrón del dominio— en el modelo candidato. En caso de que el dominio destino sea `enforce`, si al ejecutar la transformación no existe ninguna correspondencia posible, se creará un elemento que cumpla con el patrón del dominio.

La relación *ReactantsToPlaces*, muestra un ejemplo de la sintaxis para la definición de relaciones y dominios. Esta relación establece la correspondencia entre una molécula y un lugar. Encontramos dos dominios: *tpDomain* se corresponde con el dominio origen (TRANSPATH®) y *cpnDomain* con el dominio destino (CPN Tools). Para el dominio *tpDomain* se comprueba que exista una molécula con un atributo, *name*, de tipo *String*. Para cada molécula que cumpla este patrón deberá existir (y si no existe, se creará) un *Place* en el dominio destino (*cpnDomain*), cuyo atributo *id* contenga el nombre de dicha molécula.

Regla 1: Regla ReactantsToPlaces.

```
relation ReactantsToPlaces {
  molecName1 : String;
  checkonly domain tpDomain molecule1 : Molecule
    { name = molecName1 };

  enforce domain cpnDomain place1 : Place
    { id = molecName1 };

  //...

  when
    { IsSimpleMolecule(molecule1); }
  where
    { ReactantsToArcs(molecule1,place1,...); }
}
```

A la aplicación de una regla pueden añadírsele pre- y post-condiciones mediante el uso de las cláusulas `when` y `where`. Por ejemplo, la regla *ReactantsToPlaces* sólo se aplicará en caso de que la molécula sea de tipo simple —*IsSimpleMolecule(...)* es una función que mediante una expresión OCL comprueba si la molécula es simple o compuesta—. La cláusula `where` establece que tras la aplicación de la *relation* se procederá a la ejecución de la regla *ReactantsToArcs*.

El proceso de transformación completo se realiza de arriba a abajo, navegando el modelo origen a través de las relaciones de contención, definidas como asociaciones de composición. Esto es, se parte del elemento raíz del modelo origen (*Network*), y se navega hacia abajo (*Network* → *Pathways* → *Chains* → *Reactions* → *Coefficients* → *Molecules*) creando en el modelo destino los elementos correspondientes, tal y como expresa de forma resumida la Tabla I.

Para las declaraciones, la transformación creará un *ColorSet* de tipo *Enumerated* a partir de cada molécula simple. A partir de las moléculas complejas se crearán los *ColorSet* de tipo *Product*, y que estarán formados por los *Enumerated ColorSets* correspondientes a las moléculas simples que los forman.

Para los elementos visuales se parte de un objeto de tipo *Reaction*. Para cada reacción, se crea un objeto de tipo *Trans*. Navegando el rol *reactantsCoefficient* de la clase *Reaction* se obtienen los reactantes. Para cada uno de ellos, se crea un objeto *Place*. Por último ya es posible enlazar cada place recién creado con el elemento *Trans* correspondiente mediante un arco —*Arc*— de tipo *PtoT* (*Place to trans*, según la terminología de *CPN Tools*). Para los productos de la reacción se procede de forma similar, en este caso navegando el rol *productsCoefficient*. Como resultado del proceso de transformación podemos observar la Fig. 5, que muestra la representación de las reacciones del caso de estudio en *CPN Tools*. En la figura se pueden observar cuatro triángulos numerados. Cada triángulo encierra los elementos generados para la reacción identificada con el mismo número. De esta manera, para la reacción (1) —LPS + LBP ⇌ LPS:LBP—, se

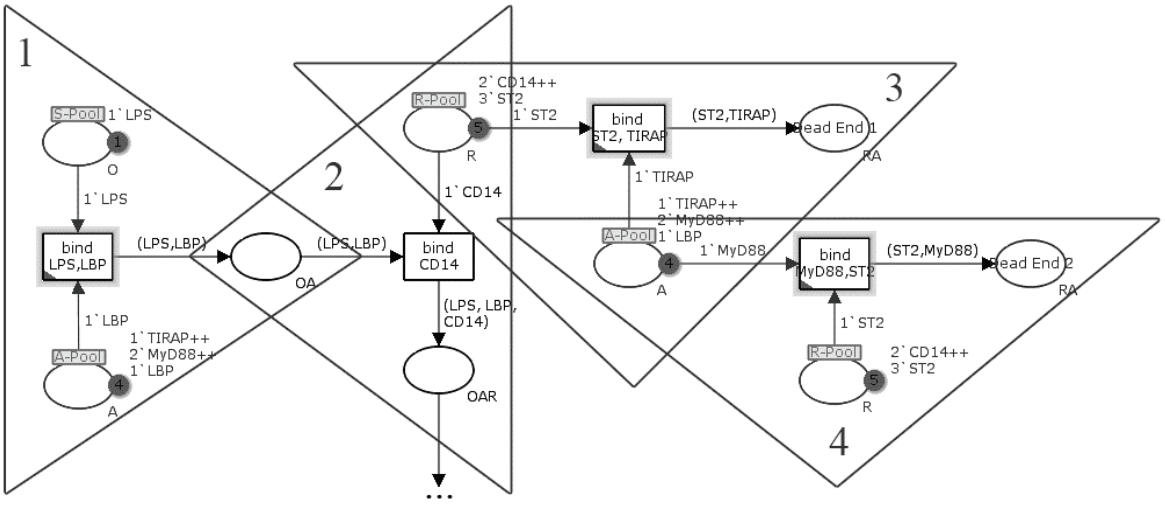


Fig. 5. Representación parcial del Pathway del TLR4 en CPN Tools.

han generado los elementos dentro del triángulo izquierdo (1). De igual manera ocurre con los triángulos 2, 3 y 4, que se corresponden con las consiguientes reacciones.

El código completo de la transformación y los ficheros de ejemplo se encuentran en [14].

V. Conclusiones

En este artículo se ha presentado un caso de estudio en el cual se aborda un problema de interoperabilidad entre aplicaciones en el campo de la bioinformática mediante un enfoque dirigido por modelos. En este entorno es común la existencia de fuentes de datos y herramientas de simulación heterogéneas, y la natural representación de los datos biológicos como modelos permite abordar estos problemas desde un punto de vista más eficiente (acortando el ciclo de vida del desarrollo de software) y más elegante (puesto que las operaciones se realizan con un lenguaje más expresivo dada su naturaleza declarativa).

El trabajo presenta como ventajas frente a las aproximaciones tradicionales (1) la automatización de tareas que anteriormente se realizaban de forma manual; (2) el desarrollo de herramientas modulares, independizando el mecanismo de transformación del formato de persistencia de los datos y facilitando la extensibilidad y mantenibilidad. (3) Aprovecha las ventajas de las transformaciones de modelos. El uso de modelos para representar los datos a ser transformados permite representar de forma más clara la estructura de éstos, siendo más intuitiva su manipulación ya que se trabaja con conceptos de alto nivel. (4) Se proporcionan capacidades de trazabilidad de forma implícita, ayudando a la localización de información errónea en los orígenes de datos. Por último, (5) el uso de un lenguaje como QVT-Relations ofrece como ventaja (frente a las aproximaciones imperativas tradicionales) que permite expresar de forma declarativa —y por tanto más descriptiva— las correspondencias entre los dominios origen y destino.

Futuras líneas de investigación de esta aproximación dirigida por modelos para el desarrollo de aplicaciones en bioinformática van dirigidas en dos direcciones. En primer lugar, la representación de los datos biológicos como modelos permite el aprovechamiento de los nuevos *frameworks* para la generación de metáforas visuales a partir de éstos, como es el caso de GMF [9] o MS DSL Tools. Por otra parte las investigaciones en ingeniería dirigida por modelos pueden proporcionar un rico background tecnológico no únicamente para la transformación de datos de un dominio a otro, sino para tareas como la integración de estos obtenidos desde orígenes heterogéneos.

Agradecimientos

Agradecemos al Prof. Dr. H.D. Ehrich y M. Ziegler del *Institut für Informationssyeteme* de la *Technische Universität Carolo-Wilhelmina zu Braunschweig*, su apoyo, colaboración y experiencia aportados en el desarrollo de este trabajo. Igualmente agradecemos su duro trabajo a Pascual Queralt como soporte en el uso de MOMENT-QVT.

Referencias

[1] Object Management Group. MDA Guide Version 1.0.1. 2003. Disponible: http: //www.omg.org/docs/omg/03-06-01.pdf.

[2] K. Czarnecki and U. W. Eisenecker. Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[3] A. Regev, W. Silverman, and E. Y. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In Pacific Symposium on Biocomputing, pages 459–470, 2001.

[4] Regev, Panina, Silverman, Cardelli, and Shapiro. Bioambients: An abstraction for biological compartments. TCS: Theoretical Computer Science, 325, 2004.

[5] W.Damm and D.Harel. LSCs:Breathing life into message sequence charts. Formal Methods in System Design, 19(1):45–80, 2001.

[6] M. Heiner, I. Koch, and K. Voss. Analysis and simulation of steady states in metabolic pathways with petri nets, Aug. 02 2001.

[7] C. Taubner, B. Mathiak, A. Kupfer, N. Fleischer, and S. Eckstein. Modelling and simulation of the TLR4 pathway with coloured petri nets. 2006. EMBS '06. 28th Annual International Conference of the IEEE,pages 2009– 2012, August 2006.

[8] C. Taubner and T. Merker. Discrete modelling of the ethylene-pathway. In ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops, page 1152, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. Disponible: http://www.omg.org/cgi-bin/doc?formal/2006-01-01.
[10] Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005. Disponible: http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.
[11] A. Boronat, J. Iborra, J. Ángel Carsí, I. Ramos, y A. Gómez. Del método formal a la aplicación industrial en gestión de modelos: Maude aplicado a eclipse modeling framework. Revista IEEE América Latina, September 2005.
[12] EMF. http://download.eclipse.org/tools/emf/scripts/home.php.
[13] M. Ziegler. Implementierung eines Transformationsmoduls in Java zur Abbildung von Signaltransduktions-Instanzen auf CPN-Instanzen. Master's thesis, Technische Universität Braunschweig, Braunschweig, Januar 2007.
[14] A. Gómez. Ficheros de ejemplo para el caso de estudio, 2007. ftp: //moment.dsic.upv.es/transpath2cpn/
[15] Eclipse Organization. The graphical modeling framework, 2006. http: //www.eclipse.org/gmf/.



Abel Gómez Llana es Ingeniero en Informática por la Universidad Politécnica de Valencia en 2005. Ha obtenido el título oficial de Máster Universitario en Ingeniería del Software, Métodos Formales, y Sistemas de Información en 2008. En 2005 realizó una estancia en el Instituto de Sistemas de Información (*Institut für Informationssysteme*, *Technische Universität Braunschweig*) donde inició sus trabajos de investigación sobre bioinformática. Actualmente es Becario FPU por el Ministerio de Educación y Ciencia en el Departamento de Sistemas Informáticos y Computación de la Universidad politécnica de Valencia. Sus áreas de interés son el desarrollo de software dirigido por modelos, las transformaciones de modelos, y su aplicabilidad en diversos campos como la bioinformática.



Artur Boronat Moll es *lecturer* en la Universidad de Leicester, en Reino Unido. Obtuvo el título de Ingeniero en Informática en 2002 y el de Doctor en Informática en 2007, ambos por la Universidad Politécnica de Valencia. Ha realizado estancias en la Universidad de Illinois en Urbana-Champaign (Estados Unidos), y en la actualidad colabora en proyectos de investigación en España, Reino Unido y Estados Unidos en el área de desarrollo de software dirigido por modelos. Sus áreas de interés son las transformaciones de modelos con reescritura de grafos y la verificación formal de técnicas de desarrollo de software dirigido por modelos.



José Á. Carsí Cubel es Licenciado en Informática en 1993 y Doctor en Informática en 1999 por la Universidad Politécnica de Valencia. En la actualidad es profesor Titular de Universidad en la Universidad Politécnica de Valencia siendo miembro del grupo de investigación de Ingeniería del Software y Sistemas de Información del Departamento de Sistemas Informáticos y Computación. Sus actuales reas de interés son: evolución del software, gestión de modelos, lenguajes de descripción de arquitecturas y modelado orientado a objetos.



Isidro Ramos Salavert es Catedrático de Universidad en el Área de Lenguajes y Sistemas Informáticos Universidad Politécnica de Valencia (España). 39 años de docencia en distintas universidades españolas y extranjeras.

Ha sido rector fundador de la Universidad de Castilla La Mancha. 20 años como Director de Departamento, de Centro. Director de 23 tesis doctorales. Investigador (principal) de 27 Proyectos de Investigación competitiva (Financiación nacional e Internacional) 3 de ellos activos. Miembro, Presidente Ejecutivo, Presidente de Comité de Programa de numerosas Conferencias y Congresos de investigación en Informática nacionales y extranjeros. Presidente de la Sociedad Española de Ingeniería del Software.

Ha trabajado en: CERN (Suiza), IMAG (Grenoble), CRIN (Nancy), Stony Brook (NYU USA). Coordinador Grupo Investigación PLIS de la UPV. Coordinador Internacional VII Subprograma Electrónica e Informática Aplicadas CYTED Coordinador Proyecto CYTED RITOS. Investigador Proyectos Europeos: BRA ASPIRE (Investigador Principal nodo Valencia).



Claudia Täubner se graduó en sistemas de información para empresas en 1998 y obtuvo el grado de doctora en Informática en la Universidad Técnica de Braunschweig (*Technische Universität Braunschweig*) en 2007.

Desde el año 2002 hasta 2007 ha estado trabajando en modelado y simulación de *pathways* de transducción de señales en el grupo de Bioinformática en el Instituto de Sistemas de Información de la Universidad Técnica de Braunschweig. Actualmente trabaja en Volkswagen IT en Wolfsburg.



Silke Eckstein obtuvo su grado de doctora en Informática en la Universidad Técnica de Braunschweig (*Technische Universität Braunschweig*) en 2001.

Desde el año 2002 ha trabajado en la consolidación y liderazgo del grupo de Bioinformática en el Instituto de Sistemas de Información en la Universidad Técnica de Braunschweig, instituto del que fue provisionalmente directora desde abril de 2007 a abril de 2008.

Incremental and Adaptive Software Systems Development of Natural Language Applications

Elena Lloret[†], Santiago Escobar[‡], Manuel Palomar[†], and Isidro Ramos[‡]

Abstract Natural Language (NL) processing tools, such as tokenizers, part-of-speech taggers or syntactic processors obtain knowledge from a set of documents (e.g., tokens, syntactic patterns, etc.) and produce the different elements that will take part on the discourse universe in a NL text (e,g., noun phrases, verbs, sentences, etc). In this paper, we present how NL software systems development can be performed incrementally by using a high-performance specification language like Maude. A generic algebraic specification for NL is defined, including sorts and subsorts apart from equational properties, such as associativity and commutativity for built-in lists and sets. Then, the full discourse universe, available for NL processing, is described in terms of the algebraic specification by providing a non-deterministic but terminating set of transformation rules. Finally, and as a proof of concept, a set of documents for NL processing is given to Maude as an input term and successfully transformed into a proper document, exploring all the non-deterministic possibilities, as well as resolving the ambiguity in language. The main advantages of implementing NL in this manner are: generality, transparency, extensibility, reusability, and maintainability. To the best of our knowledge, this is the first attempt to represent and develop complex NL software systems with this formal notation, and based on the analysis conducted, this implementation constitute the basis for the design and development of more specific NL processing applications, such as text summarization.

[†]Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante, Apdo. de correos, 99, E-03080 Alicante, Spain
e-mail: elloret@dlsi.ua.es, mpalomar@dlsi.ua.es
[‡]Departamentos de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia Valencia, Spain
e-mail: sescobar@dsic.upv.es, iramos@dsic.upv.es

Lloret, E., Escobar, S., Palomar, M., Ramos, I. Incremental and adaptive Software Systems Development of Natural Language Applications. Proc. of the 22nd International Conference on Information Systems Development (ISD 2013). 2013.

1 Introduction

Software Engineering (SE) is a systematic and disciplined approach to developing software [9]. It applies both computer science and engineering principles and practices to the creation, operation, and maintenance of software systems. Important principles in SE are abstraction, modularity, incremental development, or functional independence, among others [17]. A popular approach is Formal Methods (FM), where a specific notation with a formal semantics, along with automatic and effective tools for reasoning, is used to specify, design, analyze, verify, certify, and even implement software [10].

On the other hand, Natural Language Processing (NLP) is a research area within computer science that concerns with the design and implementation of artificially intelligent systems that are capable of using language as fluently and flexibly as humans do [7]. Intuitively, NLP, as a computer science discipline, should create computer software following a SE methodology, thus, taking into account SE principles. However, when designing and developing NLP systems, SE aspects are generally neglected [13], resulting in highly specific applications that work only for restricted domains that are really difficult to reuse, scale, or adapt. Moreover, such applications heavily depend on hardware and software local specifications, making hard to separate the logic of the system from the computing requirements.

NLP applications could significantly benefit when taking into account SE fundamentals in the design and development of the components. Currently, natural language is modelled differently, depending on several issues, such as the purpose of its application, or the available resources. No attention is paid, however, to the structures selected for its modelling, and the manner in which NLP systems are developed. For this, it should be necessary to study in a more detailed manner, methods and models that could be employed in the software development process that allow the NLP applications to work independently from the environment characteristics, and produce good quality programs in a systematic, cost-effective and efficient way.

An example of these SE techniques could be the Model-Driven Development philosophy [8], which considers models as the main assets in the software development process. Models collect the information that describes the information system at a high abstraction level, allowing the development of the application in an automated way following generative programming techniques [6]. For the Model-Driven Architecture, we choose the high-performance specification language Maude [4]. The FM approach for software specification and implementation is perfect here by providing high-level specification languages for rapid software development and easy-to-check semantics that are also easily extended by repeatedly adding more algebraic properties. This would allow us to process natural language in an incremental manner with: (i) the possibility to extent and adapt the developing systems gradually, (ii) guaranteeing SE principles, such as transparency, maintainability, and reusability of the developed components, and (iii) a high-performance prototype, (iv) providing a formal and easy-to-check semantics.

The main goal of this paper is to study and analyze how natural language can be formally defined and modelled. Our approach follows an algebraic specification

approach, via the Maude language [4]. As a result we obtain an implementation consisting of a set of transformation rules that will allow us to process natural language for each context and discourse. Achieving this objective, we will have an independent implementation for natural language that will be highly scalable and reusable, easy to adapt and extend and with the possibility to integrate more knowledge in an easy way through an incremental development. Moreover, this prototype may constitute the basis for developing wider NLP applications, such as text summarization. Specifically, this paper makes three contributions. First, it constitutes one of the first attempts to represent natural language in terms of a formal specification with a formal semantics, bringing NLP and SE/FM together. Second, it introduces the concept of incremental modelling for NLP, and shows how algebraic specification implemented through Maude can offer great advantages to the incremental development of NLP software applications. Third, it discusses how well-known problems of natural language, such as ambiguity, are resolved with this approach.

In the remainder of the paper, we first provide an analysis of previous work that bears some relation to ours (Section 2). Then, we focus on describing our formalization approach for natural language (Section 3) and its implementation (Section 4). Finally we summarize the main conclusions and outline future directions obtained from our research work (Section 5).

2 Related Work

Representation and modelling of natural language has always attracted the attention of NLP researchers. However, there is no consensus in the NLP research community about the form or the existence of a data structure that is able to describe natural language texts [5]. From a NLP perspective, natural language is modelled differently depending on: (i) the level of language analysis performed, and (ii) the specific task and/or application to be tackled.

Over the years, several mechanisms have been employed to represent natural language. Grammars were the ones which experimented a huge progress in the late 1990s, since they were a key element in the development of syntactic parsers. In those years, different types of grammars were developed, including probabilistic context-free grammars (lexicalized or unlexicalized) [12], definite clause grammars [16], or combinatory categorial grammars [19], among others. Most of them were derived from large corpora, such as Penn treebank. However, grammars had well-known limitations [18]. On the one hand, they were too big to be managed, whereas their coverage was somewhat limited. On the other hand, the degree of ambiguity was too great that exhaustive search was not feasible. Other existing approaches that also try to model natural language employ statistical techniques and probability, such as n-grams or bag-of-words [11], but their objective is to obtain knowledge, considering that a text is characterized by such elements, rather than proposing a reusable approach for defining language. Recent efforts have proposed to represent and model the semantics natural language using ontologies [1], as well to establish

formal mechanisms (e.g., NLP Interchange Format) to facilitate the interoperability of NLP tools [2]. However, these are still in their early stages.

Platforms such as GATE[1] or UIMA[2] allow the integration of different NLP resources and tools, that were initially developed in a different way, to build complete NLP systems. Our aim greatly differs from the development of these tools. It focuses more on the definition of NLP tasks in an incremental and reusable way, so that complex NLP processes can be built on top of the basic ones with the addition of minimum knowledge.

To the best of knowledge there is not previous research work with the goal of modelling natural language from a SE perspective, and more concretely using FM, and in particular, algebraic specification. By describing natural language in an incremental manner and through a high-performance specification language like Maude, as we proposed in our approach, we have a structure flexible and adaptative enough to be used for developing NLP applications following a standard methodology.

3 Formalization of Natural Language

To set the basis for modelling natural language by means of a formal specification, we can follow one of these approaches: (i) a manual approach, where a linguistic expert defines all necessary knowledge and takes into account all its possible variations; (ii) a fully automatic approach, where existing NLP tools are employed for extracting and obtaining knowledge from a set of documents; and (iii) a hybrid approach, where (i) and (ii) are combined, in the sense that the output of NLP tools is revised by an expert human in order to correct the possible errors made by the automatic processing. Our proposed approach follows this latter approach, relying on the knowledge obtained from automatic NLP tools, and correcting the output of these tools, when necessary, in order to avoid the time-consuming process of (i), and the errors derived from automatic tools in (ii). One important difference is that we explore all possibilities, so returning all possible meanings to a human user for further inspection.

The process of knowledge extraction follows a top-down methodology. Using existing NLP tools that take a document as input and obtain information about their components, we employ such knowledge for describing natural language. Two levels of language analysis are involved in this process: (i) lexical-morphological analysis through a part-of-speech tagger, and (ii) syntactic analysis through a syntactic parser. The former is useful for identifying the types of tokens (i.e. words) included in a document, whereas the second determines the types of syntactic structures.

Then, our framework for describing natural language is as follows. First of all, a generic algebraic specification for NLP is defined, including sorts and subsorts apart of equational properties, such as associativity and commutativity for built-in

[1] http://gate.ac.uk/

[2] uima.apache.org/

lists and sets. This establishes a cornerstone for the assets on generality and extensibility of our incremental natural representation. Then, the full discourse universe, obtained from existing NLP tools and which must be available for processing natural language, is described in terms of the algebraic specification. For each lexical element, a transformation rule is defined. If a lexical element has some ambiguity, several transformation rules would be defined, providing a non-deterministic set of transformation rules. Also, for each syntactic association, a transformation rule is defined but in terms of the lexical elements rather the original text elements. This part is clearly non-deterministic, since multiple combinations of associations may be possible. The addition of extra transformation rules for lexical elements or syntactic associations provides the main assets on transparency, reusability, and maintainability of our incremental natural language processing. Finally, a set of documents for natural language processing is given to Maude as an input term and successively transformed into a proper document, exploring all the non-deterministic possibilities. Since, all the transformation rules are terminating, we know the search space is finite and, thus, manageable by existing formal methods tools, such as a reachability command within the specification language Maude.

4 The Rewriting Logic Semantics of Natural Language

This section contains the technical implementation details for representing natural language in an incremental and adaptive manner, using the Maude language. Maude [3] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming. Functional modules describe data types and operations on them by means of equational theories. Mathematically, such a theory can be described as a pair $(\Sigma, E \cup A)$, where: Σ is the signature that specifies the type structure (sorts, subsorts, kinds, and overloaded operators); E is the collection of equations declared in the functional module, and A is the collection of equational attributes (associativity, commutativity, and so on) that are declared for the different operators. System modules allow the description of the dynamic behaviour of a system beyond data types and operations on them by allowing system transition rules that constitute a search space graph. Given an initial state and a search pattern, the `search` command performs the search through the rules and equations defined. These equations are, in fact, rules associated to deterministic actions and their application do not generate new states in the search space.

4.1 Incremental Implementation

The cornerstone of our framework is the built-in Maude sort for *Quoted Identifiers* (QID). A QID is denoted by symbol ' followed by letters and numbers, e.g. `'account123`, and is a wrapper for strings in order to provide a Maude representa-

tion for tokens of Maude syntax, which is essential for the meta-representation and meta-reasoning capabilities of Maude. We could have used the built-in Maude sort *String* for representing words but its treatment is much more inefficient than that of QID. The module QID-LIST defines a built-in data type of lists of quoted identifiers, and it is our most important tool because it provides an associative operator __ (this represents the symbol with empty syntax in Maude) with an identity operator `nil` for an empty list, allowing yuxtaposition of QID's. This is crucial for allowing us to define our types based on sequences of QIDs without having to look for the individual elements in a different data structure. That is, if we need to search for the word `'a` followed by the word `'tree` with zero or more words between them, for example to be replaced by `'a 'dog`, we just have to write the transition rule

```
rl X:QidList 'a Y:QidList 'tree Z:QidList
=> X:QidList 'a Y:QidList 'dog Z:QidList .
```

in Maude where `X:QidList`, `Y:QidList`, and `Z:QidList` can be substituted by the empty list or any sequence of QID's. This avoids writing code for searching for the word `'a` and then searching for the word `'tree` or restarting the search after finding `'a` but not finding `'tree`, thus easing the way for the specification of NLP.

From the knowledge obtained through the lexical-morphological and syntactic analysis, we begin with the definition of the most basic types, so complex types could be then generated from the basic ones. For the implementation of our prototype, a corpus of 567 English newswire documents provided by DUC[3] was employed. This corpus will constitute the discourse universe for our implementation.

From the part-of-speech tagger output (lexical-morphological analysis), the different categories were grouped into 9 general ones (*determiner*; *pronoun*; *noun*; *preposition*; *adjective*; *adverb*; *verb*; *stopword*; *punctuation mark*). The set of tags from the part-of-speech tagger[4] was automatically mapped to the previously mentioned categories. For instance, the *verb* category would consider tags, such as *VVN*, *VBD*, *VBG*, *VBZ*, or *VBP*. Each of these categories will be a different module in Maude.

For each module, we first define a sort for the category and we specify whether it is also a subsort of another one. All these basic categories are subsort from the QID sort. The next step is to define the signature and semantics for each of the modules. The signature consists of the operations for obtaining the categories. The semantics of each module will contain the equations and rules by means of which this category is built. For this, we use a symbol representing each category, i.e., `pronoun`, `noun`, `adverb`, etc., and the original word is kept as an argument. We could have defined a symbol for each word in each category, e.g. `pronoun-he` or `noun-attack`, but it is easier to implement by keeping the original word as an argument. However, this forces us to make all the category symbols to be frozen for further rewriting (denoted by the labels `[frozen strat (0)]` in Maude), i.e., to avoid nested applications of the transformation rules in the form `'you` → `pronoun('you)` → `pronoun(pronoun('you))`, leading to an infinite loop.

[3] http://www-nlpir.nist.gov/projects/duc/

[4] http://www.sketchengine.co.uk/tagsets/penn.html

We next provide an example of the signature and fragments of the semantics for the category *noun*.

```
mod NOUN is
 protecting QID .

 sort Noun .
 subsort Qid < Noun .

 op noun : Qid -> Noun [frozen strat (0)] .

 eq 'advantages = noun('advantages) .
 eq 'adventurers = noun('adventurers) .
 eq 'adventures = noun('adventures) .
 eq 'advice = noun('advice) .
 [...]
 rl 'account => noun('account) .
 rl 'act => noun('act) .
 rl 'aid => noun('aid) .
 rl 'bear => noun('bear) .
 [...]
 endm
```

As it can be seen, for this module, as well as for the remaining types (determiner, verb, adverb, adjective, stopword, and puntuation mark), we define that each type can take the form of a QID. The attribute *frozen* is also present in all of them and it is crucial in the definition, in order to ensure that the program terminates; otherwise, we could end up with an infinite loop. In the semantics of each type, the elements of our discourse universe that fall into the corresponding categories are included. Non-ambiguous terms (e.g., `eq 'advice = noun('advice) .`) are represented through equations (eq), whereas rules (rl) are used for the ambiguous ones (e.g., `rl 'account => noun('account) .`).

Having implemented the basic categories, we exploit the knowledge obtained from the syntactic parser (syntactic analysis) for defining types of language structures that are needed for creating sentences, and documents. In particular, the ones that we intially use for our research are: *noun phrase*, *verb phrase*, which will lead to the definition of the modules *SN*, *SV* in Maude. Below, the implementation of *SN* module is shown.

```
mod SN is
 protecting DETERMINER .
 protecting PRONOUN .
 protecting STOPWORD .
 protecting PREPOSITION .
 protecting ADJECTIVE .
 protecting NOUN .
 protecting QID-LIST .

 sort Sn .
 subsort QidList < Sn .
 op sn : QidList -> Sn [frozen strat (0)] .
```

```
  rl noun(A:Qid) => sn(A:Qid) .
  rl pronoun(A:Qid) => sn(A:Qid) .
  rl determiner(A:Qid) sn(B:QidList) => sn(A:Qid B:QidList) .
  rl preposition(A:Qid) sn(B:QidList) => sn(A:Qid B:QidList) .
  rl stopword(A:Qid) sn(B:QidList) => sn(A:Qid B:QidList) .
  rl adjective(A:Qid) sn(B:QidList) => sn(A:Qid B:QidList) .
  rl sn(A:QidList) sn(B:QidList) => sn(A:QidList B:QidList) .
endm
```

The *SN* module inherits the properties of some of the basic categories, as well as the built-in QID-LIST Maude module. The rules reflected in the definition of this type shows how a noun phrase could be produced. These are recursive rules, that will allow the construction of noun phrases following a recursive and transformational process starting from the basic types of words included in the text. Following an analogously process, but taking into account the elements that would constitute a verb phrase, we define the *SV* type.

With the *SN* and *SV* data types, we can define the signature and semantics for a sentence and a document. We define a sentence as a structure that contains a noun phrase, followed by a verb phrase, and ended with a punctuation mark. In addition, the syntactic analysis carried out also detected as sentences, grammatical structures involving either a noun or a verb phrase followed by a punctuation mark. Finally, we create a module for representing natural language documents. We assume that a document can be formed by a single sentence or a concatenation of them.

It should be immediate to understand for the reader that all the modules are easily extensible and reusable, since all the lexical and syntactical information is defined by transformation rules/equations and all such rules/equations are local to the specific word or token being used, so there is no necessary global data or context information. Furthermore, these modules provide a lot of generality and transparency, since the set of new defined sorts is minimal.

4.2 Analysis and Discussion

Once all the necessary data types via Maude modules were implemented, we conduct an analysis of its performance. In this analysis, we want to check if natural language can be represented through our proposed types. Moreover, we want to check whether our approach would be computationally efficient, even though optimization is out of the scope of this paper.

We conduct a battery of tests in order to evaluate our implementation in a qualitative manner. Next, we provide several representative examples for the different types and discuss their results.

Example 1: Basic categories.

The following examples show cases where individual words are tested in order to check if they belong to the correct category.

```
search 'happy =>* adjective(Q:Qid) .
```

```
Solution 1 (state 0)
Q:Qid --> 'happy
No more solutions.

search 'happy =>* verb(Q:Qid) .
No solution.

search 'account  =>* verb(Q:QidList) .
Solution 1 (state 2)
Q:QidList --> 'account
No more solutions.

search 'account  =>* noun(Q:QidList) .
Solution 1 (state 1)
Q:QidList --> 'account
No more solutions.
```

As it can be seen, the token "happy" is correctly recognized as an adjective, and not as a verb. It is worth mentioning how ambiguity is tackled at the execution stage. Let's consider the term *"account"*. This word is ambiguous since it can act as a noun as well as a verb. At the execution time, we perform a search within a particular category, so for all the possible states in the specific category, the program will check whether the word *account* exists. In our examples, when we search this word either in the space of a verb or a noun, the word will be recognized.

Another issue that is also important to note is the number of states visited and the time spent for processing the terms. Maude is really fast, consuming less than 1 second of CPU time. The predominant use of rules and equations makes the program to be faster than if we had used only rules for implementing the semantics for all the types.

Example 2: Language structures for creating sentences

These examples show how natural language structures can be also identified.

```
search 'the 'account =>* sn(Q:QidList) .
Solution 1 (state 5)
Q:QidList --> 'the 'account
No more solutions.


search
'the 'dog 'and 'the 'bear =>* sn(Q:QidList) .
Solution 1 (state 22)
Q:QidList --> 'the 'dog 'and 'the 'bear
No more solutions.
```

Concerning verb phrases (*SV*), our implementation deals with verb phrase that include prepositional phrases form with a preposition, a determiner and a noun.

```
search
'play 'with 'the 'cat =>* sv(Q:QidList) .
Solution 1 (state 20)
Q:QidList --> 'play 'with 'the 'cat
No more solutions.
```

Finally, we provide an example in order to ensure that noun phrases are not recognized as verb phrases.

```
search
'the 'dog 'and 'the 'bear =>* sv(Q:QidList) .
No solution.
```

From all these illustrative examples, we can highlight the fact that although the number of states and rewriting steps have increased with respect to the search performed for the basic categories, the time spent for obtaining the solution is marginal.

Example 3: Sentence and document recognition

Provided that we have the vocabulary employed in our discourse universe, our model for representing natural language will be able to recognize sentences and documents. In these cases, the number of rewriting steps is higher than in the previous examples. The longer and more complex the sentence or the document is, the longer it takes for the system to analyze it. Moreover, for their definition, we only have used rules and not equations, leading to the fact that the search space is higher.

```
search 'the 'dog 'plays 'with 'the 'cat ';
   =>* sentence(Q:QidList) .
Solution 1 (state 81)
states: 82
rewrites: 158 in 0ms cpu
Q:QidList --> 'the 'dog 'plays
              'with 'the 'cat ';
No more solutions.
states: 95
rewrites: 191 in 0ms cpu

search 'nero 'specializes 'in 'sniffing
   'out 'bombs 'and 'narcotics ';
    =>* sentence(Q:QidList) .
Solution 1 (state 602)
states: 603
rewrites: 1805 in 12ms cpu
Q:QidList --> 'nero 'specializes 'in 'sniffing
              'out 'bombs 'and 'narcotics ';
No more solutions.
states: 616
rewrites: 1848 in 12ms cpu

search 'the 'dog 'plays 'with 'the 'cat ';
     'nero 'specializes 'in 'sniffing
     'out 'bombs 'and 'narcotics ';
     =>* document(Q:QidList) .
Solution 1 (state 58515)
states: 58516
rewrites: 288541 in 1928ms cpu
Q:QidList --> 'the 'dog 'plays 'with 'the
              'cat '; 'nero 'specializes 'in
              'sniffing 'out 'bombs 'and
              'narcotics ';
No more solutions.
```

```
states: 58549
rewrites: 288737 in 1932ms cpu
```

In this paper, we have focused on the presentation of the framework and further optimizations on the search space have to be considered, which is a typical topic of research in formal methods applied to verification. The key idea is that the transformation rules are terminating and, thus, the search space would always be finite. Although our approach has not been evaluated in a quantitative manner, we have verify the adequacy of our method through a set of representative examples, focusing also on the time spent for processing the different modules. More evaluation examples and their performance analysis can be found in [14].

5 Conclusion and Future Directions

In this paper we modelled natural language following an incremental and adaptive algebraic specification approach via a high-performance language (Maude). As far as we know, this is the first attempt to represent natural language in this manner. NLP tools were employed for obtaining and extracting knowledge from documents, and such knowledge was used to define sorts, subsorts and equational properties. Not only was a novel method to represent natural language proposed, but also our implementation was designed in a way that is highly scalable and reusable, thus taking into consideration SE principles, which are not often paid attention by the NLP research community.

An important advantage of this representation is that it would be very easy to add further knowledge, as well as to extend it to other languages. Furthermore, the development of more complex NLP applications, such as information retrieval or text summarization, could be quite straightforward, given that the necessary language analysis levels are performed.

Taking this preliminary study as a starting point, there are several interesting issues that will guide our research work in the future. In the short-term we plan to define more complex data types, as well as optimizing the performance of our initial prototype. The addition of new knowledge as well as new data types will constitute the basis for the development of complex NLP applications, e.g., text summarization, that we plan to tackle in the long-term. Moreover, we also plan to adopt the ideas for natural language deconstruction proposed in [15] that could benefit our approach by providing a flexible approach and determining which language structures should be modelled depending on the aim of the NLP application.

Acknowledgements This research has been partially funded by the Spanish Government through the project TEXT-MESS 2.0 (TIN2009-13391-C04) and Técnicas de Deconstrucción en la Tecnologías del Lenguaje Humano (TIN2012-31224) and by the Generalitat Valenciana through project PROMETEO (PROMETEO/2009/199). Moreover, S. Escobar has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2010-21062-C02-02, and by Generalitat Valenciana PROMETEO2011/052.

References

1. Bateman, J.A., Hois, J., Ross, R., Tenbrink, T.: A linguistic ontology of space for natural language processing. Artificial Intelligence **174**(14), 1027 – 1071 (2010)
2. Chiarcos, C.: A generic formalism to represent linguistic corpora in rdf and owl/dl. In: Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12) (2012)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 system. In: Rewriting Techniques and Applications (RTA 2003), 2706, pp. 76–87 (2003)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, *Lecture Notes in Computer Science*, vol. 4350 (2007)
5. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. Journal of Machine Learning Research **12**, 2493–2537 (2011)
6. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
7. Dale, R., Somers, H.L., Moisl, H. (eds.): Handbook of Natural Language Processing. Marcel Dekker, Inc., New York, NY, USA (2000)
8. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons, Inc., New York, NY, USA (2002)
9. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering, 2nd edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2002)
10. Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., Margaria, T.: Software engineering and formal methods. Commun. ACM **51**(9), 54–59 (2008)
11. Huang, F., Yates, A., Ahuja, A., Downey, D.: Language models as representations for weakly-supervised nlp tasks. In: Proceedings of the Fifteenth Conference on Computational Natural Language Learning, pp. 125–134 (2011)
12. Klein, D., Manning, C.D.: Accurate unlexicalized parsing. In: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1, pp. 423–430. Association for Computational Linguistics, Stroudsburg, PA, USA (2003)
13. Leidner, J.: Current issues in software engineering for natural language processing. In: Proceedings of the Workshop on Software Engineering and Architecture of Language Technology Systems, pp. 45–50 (2003)
14. Lloret, E., Escobar, S., Palomar, M., Ramos, I.: Natural language modelling using maude. Tech. rep., University of Alicante (2013)
15. Martínez-Barco, P., Ferrández-Rodríguez, A., Tomás, D., Lloret, E., Saquete, E., Llopis, F., Peral, J., Palomar, M., Gmez-Soriano, J.M., Romá, M.T.: Legolang: Técnicas de deconstrucción en la tecnologías del lenguaje humano. Procesamiento de Lenguaje Natural (51) (2013)
16. Pereira, F., Warren, D.: Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence **13**, 231–278 (1980)
17. Pressman, R.S.: Software Engineering: A Practitioner's Approach, 5th edn. McGraw-Hill Higher Education (2001)
18. Steedman, M.: Some important problems in natural language processing. Tech. rep., University of Edinburgh (2010)
19. Steedman, M., Baldridge, J.: Combinatory Categorial Grammar, pp. 181–224. Wiley-Blackwell (2011)



ISIDRO RAMOS

PUBLICACIONES 1968-2013

VOLUMEN 5:
DESARROLLO DIRIGIDO POR MODELOS
LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)
LOS AMIGOS

VALENCIA, SEPTIEMBRE DE 2014

ÍNDICE GENERAL

VOLUMEN 1: MADRID, BILBAO, VALENCIA

Prólogo i

Días de hemeroteca v

… y noches de ordenador ix

Estructura del libro xi

El saber y el saber hacer en informática: una perspectiva histórica xiii

Índice general xxi

PARTE I: MADRID (1968-1974)

Prólogo de Roberto Moya Quiles 3

Isidro Ramos y el Centro de Cálculo de la Universidad de Madrid, por Ernesto García Camarero ex-director del CCUM 5

Seminarios del Centro De Cálculo De la Universidad Complutense de Madrid. 1968. 11

Gil Gayarre, M., Ramos, I., et al., Posibilidades de la Automatización en la corrección de exámenes. Revista de Automática. Año III, nº 7. 1970. 109

Fernández Flores, I. Ramos I., Traductor 70/13. Boletín CCUM. Volumen 10. 1970. 123

Moya R., Ramos, I. Aplicación de Hash-Coding al tratamiento automático de Gramáticas". Boletín CCUM. nº 17. 1971 137

Fernández Flórez, I., García Camarero, E., Ramos, I. A Basic Language Oriented to secondary School. Information Processing 71. 1972. 147

García Camarero, E., Ramos, I. Ensayo sobre la definición automática de tratamientos clínicos. Boletín CCUM. 1971. 151

Ramos, I. Lenguajes para la escritura de traductores. Revista de Automática. Enero-Marzo 1973. 163

Alfonseca, M., Ramos I. Un analizador sintáctico para un subconjunto de PL/I. Actas del Congreso Nacional de Automática. 1972. 169

Ramos, I. Un ejemplo de recursividad: HANOI. Boletín CCUM. nº 24. 1974. 205

Ramos, I., Lozano, C. Muñoz, A. Sistema de Creación y Mantenimiento de Bancos de Información. Revista de Automática, 1974 (Parte I), 1975 (Partes II y III). 211

PARTE II: BILBAO (1975-1983)

Prólogo de José Ramón Gariagoitia 255

Ramos, I. Système pour la création et gestion des banques de données. Seminaires Institut de Programmation (Université de Paris VI). 1974. 257

Moya R., Ramos I. Metaprogramación Estructurada. 3er Congreso Nacional de Informática y Automática. 1975. 263

Ramos, I., Moya, R., Martín, P. Evaluador Sintáctico del Lenguaje de Control de Comandos del Sistema Operativo EXEC-8. 3er Congreso Nacional de Informática y Automática. 1975. 289

Ramos, I. Aramberri, J. Cross-Macroassembler for Micro MCS-80 in a HP 2116-B Minicomputer. Euromicro. 1976. 301

Ramos, I. Informátique de Base: These, Antithèse, Synthèse. Convention Informatique. 1976. 309

Garitagoitia, J.R., Ramos, I. Relocating Direct Linking Loader for MCS-8 Microcomputer Series. Proc. of COMPCON 77. 1977. 313

Ramos, I. Sistema Editor / Perforador de textos. Revista de Automática e Informática. Volumen: nº 31. 1977. 319

Moya, R. Ramos, I. LPE: Un lenguaje en español para la programación estructurada. Proceso de Datos, no. 70, 1977. 325

Moya, R., Ramos I. SPL (Structured Programming Language). A language for structured programming. Proc. Of the 5th Annual III Conference. 1977. 331

Ramos, I. Aproximación prospectiva al software de base. Actas de la Convención Informática Latina. 1977. 339

Ramos, I., Aramberri, J. y Mellado, M. Modelo de realización de primitivas de sincronización para sistemas multimicroprocesador. Actas del 75 Aniversario Real Sociedad Española de Física y Química. 1978. 345

Aramberri, J., Ramos, I., Garitagoitia, J. R. Diseño y desarrollo de un sistema microprocesador. Actas I Jornadas sobre microprocesadores. 1978. 347

Moya, R., Ramos, I. SP Gramars (Context Free Gramars for Structured Programming). ACM SIGPLAN. Vol.4 Nº1. 1979. 357

Aramberri, J, Ramos, I., Mellado, M., Garitagoitia, J. R. Modelo de Sistema Microprocesador. Actas del 4º Congreso de Informática y Automática. 1979. 365

Sopeña, L., Ramos, I. Sistema de control en tiempo real de una red viaria. Actas del 4º Congreso de Informática y Automática. 1979. 375

Aguirre, J., Ramos, I. Sistema de programación estructurada con vuelta atrás. Actas del 4º Congreso de Informática y Automática, Madrid, 1979. 385

Ramos, I. Garitagoitia, J.R., Aramberri, J. Algunas soluciones al problema de interbloqueo entre procesos. 4º Congreso de Informática y Automática. 1979. 391

Guaresti, J.M., Garitagoitia, J.R. y Ramos I.A software chronological state controller. Actas del 4º Congreso de Informática y Automática. 1979. 403

Ramos, I. Calculadora de mesa. Un proyecto de clase sobre teoría de Lenguajes y Autómatas. Revista de Informática y Automática. 1979. 415

Zuloaga, J.L., Santamaria, R., Emaldi, R., Ramos, I. Sistema de Control lógico programable sobre microordenador. Revista de Informática y Automática. 1979. 421

Garitagoitia, J.R., Martinez,C., Ramos, I. MIKBUS: un bus para placas de microcomputadores cooperantes. Actas del II Simposio Español de Informática Repartida. 1982. 429

Garitagoitia, J.R., Ramos, I. Algoritmo de prevención para un sistema de gestión de ficheros. Actas de la Convención Informática Latina. 1983. 439

PARTE III: UNIVERSITAT DE VALÈNCIA - UNIVERSIDAD DE CASTILLA-LA MANCHA (1978-1987)

Prólogo de Francisco José Quiles Flor 455

Ramos, I. Semántica para SPL. Questió. Vol. 4. nº 1.1980. 459

Ramos, I. Un lenguaje de Programación con tipos Abstractos interpretado en PROLOG. Otra forma de compilar. Revista de Informática y Automática. 1984. 467

Ramos, I. Especificación de Pantallas. Revista de Informática y Automática, 1984. 471

Marín, J, Ramos, I. Metodología para la adquisición de material informático. Actas de las Jornadas de Informática en la Administración Local. 1984. 475

Ramos, I. The impact of new tecnologies on University organization: the example of Castilla-La Mancha. C.R.E. INFORMATION. Volumen: nº 70. 1985. 491

VOLUMEN 2: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

PARTE IV: MÉTODOS FORMALES EN INGENIERÍA DE REQUISITOS (1987-2002)

Prólogo de María Alpuente 527

Celma, M., Moreno, L., Pastor, O., Ramos, I. Implementación de un sistema de ges-

tión de bases de datos deductivas. Revista de Informática y Automática, Vol. XX, Nº 3. 1987. 529

Alpuente, M., Botti, V., Galiano, I., Sanchis, E., Vila, F., Ramos, I. Entorno de programación LALI implementado en PROLOG. Revista de Informática y Automática . Vol. XX, Nº 4. 1987. 537

González del Río, A., Alpuente, M., Casamayor, J.C., Pastor, M.A., Ramírez, M.J., Ramos, I. Protolog : A conceptual Schema Facility for Automated Prototype Generation. 7th. IASTED International Symposium. 1989. 543

Ramos I., Canós, J.H., Forradellas, R., Oliver, J. A conceptual schema specification system for rapid prototyping. 8th IASTED International Symposium. Austria. Feb. 1990. 547

Ramos, I. Logics and OO-Data Bases: a declarative approach. Proc. International Conference on Database and Expert Systems Applications (DEXA 90), 1990. 551

Ramos, I.; Pastor, O.; Casado, V. OO and Active Formal Information System Specification. Proc. International Conference on Database and Expert System Applications (DEXA 91) Vol. II. 1991. 557

Pastor, O. Ramos, I. Canós, J.H. Object Oriented and Relational Specification of Information Systems. Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS. 1991. 563

Ramos, I., Cuevas, J., Devesa, J., Pastor, O., Canós, J.H. On the use of Algebras as Semantic domains of Object Societies. Proc. of the 3rd International Workshop of the Deductive Approach for DB and IS. 1992. 579

Alpuente, M., Ramírez, M. J. An Equational Constraint Logic approach to Conceptual Modelling. In Ramos, I. and Tjoa, A. (eds), Proc. of the International Conference on Database and Expert Systems Applications (DEXA 92), 1992. 589

Ramos, I., Pastor, O., Cuevas J., Devesa , J. Objects as Observable Processes. Proc. of the 4th International Workshop of the Deductive Approach for DB and IS.1993. 595

Pastor, O., Ramos, I., Canós, J.H. From Analysis to Design in an OO and Deductive Environment. Proc of the International Conference on Database and Expert System Applications (DEXA´93). 1993. 611

Toval, A., Ramos, I. , Pastor, O. Prototyping Object Oriented Specifications in an Algebraic Environment. Proc. Of the International Conference on Database and Expert System Applications (DEXA'94), 1994. 617

Devesa, J., Cuevas, J., Ramos, I. Implementation of a Term Rewriting System for solving process queries in an object-oriented environment". Proc. Of the GULP-PRODE 94. 1994. 627

Canós, J. H., Penadés, M.C., Ramos, I., Pastor, O. A Knowledge-Base Architecture for Object Societies. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 643

Pastor, O., Ramos, I., Canós J.H. OASIS v2: A Class Definition Language. Proc. of the International Conference on Database and Expert System Applications (DEXA 95). 1995. 649

Ramos, I., Toval, A., García-Molina, J., Flores, M., González, M., Fernández, J. L., Nicolás, J., Pastor, O., Canós, J. H., Pelechano, V., Penadés, M.C., Bonet, B. Entorno General de Produccion automática de prototipos de software orientado a objetos: OOAP (Object-Oriented Assistant Prototyper). Actas de las I Jornadas de Informática. Asociación Española de Informática y Automática (AEIA). 1995. 659

Pastor O., Pelechano V., Bonet B., Ramos I. An object Oriented Approach for making Automated Prototyping Feasible. Proc. of the International Conference on Database and Expert System Applications (DEXA 96). 1996. 673

Carsí, J.A., Ramos, I., Penadés, M.C., Pelechano, V., Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic. Actas de las I Jornadas de Trabajo en Ingeniería del Software, 1996. 683

Penadés, M.C., Ramos, I., Canós, J.H., Tolsá, J., Ferrer, J. Análisis, diseño e implementación de un Entorno de Prototipación Automática. Actas de las I Jornadas de Trabajo en Ingeniería del Software,1996. 695

Sánchez, P. Letelier, P., Ramos, I. Representación de Objetos OASIS como Teorias y Traducción a un Modelo de Concurrencia. Computación y Sistemas, Vol. I, Nº. 2. 1997. 707

Lozano, M.D., Ramos, I. Evolución y perspectivas en el desarrollo de software: nuevas tendencias orientadas a objetos. Ensayos: revista de la Escuela Universitaria de Magisterio de Albacete, Nº. 12. 1997. 719

Sánchez, P., Letelier, P., Ramos, I. Constructs for Prototyping Information Systems using Object Petri Nets. Proc. of the IEEE International Conference on System, Man and Cybernetics.1997. 733

Lozano, M.D., Ramos, I., Cuenca, P. Application of an Object-Oriented Methodology for Automatic Software Production : OO-Method. Proc. Of the IEEE PACRIM 97. 1997. 739

Letelier, P., Sánchez, P., Ramos, I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitios. Proc. of the Argentine Symposium on Object Orientation. 1997. 743

Letelier, P. Sánchez, P., Ramos. I. Animation of Systems Specifications Using Concurrent Logic Programming. Proc. of the Symposium in Logical Approaches to Agent and Modelling and Design (ESSLLI´97). 1997. 751

Carsí, J.A., Canós, J.H., Ramos, I. Un intérprete de Transaction Frame Logic. Actas de las II Jornadas de Trabajo en Ingeniería del Software. 1997. 759

Tolsá, J.M., Ferrer, J., Penadés, M.C., Canós, J.H., Ramos, I. Meta-KAOS. Un metaentorno de prototipación automática. Actas de las II Jornadas de Trabajo en

Ingeniería del Software. 1997. 773

Ramos, I.,Canós, J.H., Pastor,O.+ investigadores proyecto MENHIR. MENHIR: Modelos, Entornos y Nuevas Herramientas para la Ingeniería de Requisitos. Actas de las III Jornadas sobre Tecnología de Objetos. 1997. 787

Canós, J.H., Lozano, M.D., Ramos, I. KAOS: An Object-Oriented Software Tool for the Objects Definition, Updating, Querying and Programming in an Object-Oriented Environment. Proc. of the IEEE Canadian Conference on Electrical & Computer Engineering.1998. 799

Letelier, P., Sánchez, P., Ramos, I. Especificaciones de proceso para objetos y su representación en Lógica Dinámica. Actas de las III Jornadas de Ingeniería de Software. 1998. 803

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto. Actas de las III Jornadas de Ingeniería del Software. 1998. 813

Sánchez, P. Letelier, P., Ramos, I. Una aproximación a las pautas de representación de OASIS en un formalismo gráfico. Actas de las III Jornadas de Ingeniería del Software. 1998. 827

Carsí, J. A., Camilleri, S., Canós, J.H., Ramos, I. Propuesta de Interfaz Gráfica de Usuario homogénea para el diseño y la explotación de Sistemas de Información. Actas de las III Jornadas de Ingeniería del Software. 1998. 839

Letelier, P., Sánchez, P., Ramos, I. Un ambiente para especificación incremental y validación de modelos conceptuales. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99).1999. 853

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto. Actas del 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'99). 1999. 867

Letelier, P., Sánchez, P., Ramos, I. Prototyping a requirements specification through an automatically generated concurrent logic program. Proc. of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99). 1999. 879

Pastor,O., Canós,J.H., Ramos,I. From CASE to CARE (Computer-Aided Requirements Engineering). Proc. of the International Conference on Conceptual Modeling (ER'99). 1999. 895

Lozano, M., Ramos, I. Integration of the User Model and Human Factors in an Object Oriented Software Production Environment. Workshop on Integrating Human Factors into Use Cases and OO Methods at ECOOP'99. 1999. 911

Letelier, P., Sánchez, P., Ramos, I. Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales". Actas del VI Congreso Internacional de Investigación en Ciencias Computacionales (CIICC'99). 1999. 917

Sánchez, P., Letelier, P., Ramos, I., Pastor, O. Modelado conceptual con un lenguaje

formal y orientado a objeto. Actas de la XXV Conferencia Latinoamericana de Informática (CLEI'99). 1999. 929

Letelier, P. Sánchez, P., Ramos, I. Animation of Conceptual Models using two Concurrent Environments: An overview. Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers. 1999. 943

Parets, J., Carsí, J., Canós, J.H., Anaya, A., Hurtado, M.V., Penadés, M.C., Paderewski, P., Ramos, I., Rodriguez, M.J., La evolución de modelos en el desarrollo de software: los enfoques MEDES y OASIS. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos. 1999. 951

Sánchez, P., Letelier, P., Ramos, I. Una Interpretación de la Herencia de OASIS en Lógica Dinámica. Computación y Sistemas, vol. 4, núm.4. 2000. 963

Sánchez, P., Letelier, P., Ramos, I. Animating formal specifications with Inheritance in a DL-based framework. Requirements Engineering Journal, vol.4, no.4. 2000. 973

Anaya, R., Ramos, I. Un lenguaje formal para especificación y refinamiento de Objetos de Negocio. Ingeniería del software y bases de datos: tendencias actuales. Universidad de Castilla-La Mancha, 2000. 995

Anaya, R., Ramos, I. AR2CA: Una Herramienta para la Construcción de componentes reutilizables a través de niveles de refinamiento. Actas de 3as Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software (IDEAS2000). 2000. 1011

Carsí, J. A., Ramos, I., Canós, J.H. Reflection in the OASIS object model to solve Schema Evolution Problems. Proc. of the International Symposium on the Principles of Software Evolution (ISPSE2000). 2000. 1025

Sánchez, P., Letelier, P., Ramos, I. Validation of Conceptual Models by Animation in an Scenario based approach. Workshop 'Scenario-based round-trip engineering' at OOPSLA 2000. 2000. 1031

Giret, A., Ramos, I.Full software agents as OASIS 3.0 objects. Proc. of the 2nd Constraint) Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2001. 2001. 1037

Letelier, P., Ramos, I. Trazabilidad de Requisitos y Calidad del Software: Una Aproximación basada en UML. En Piattini y García (Eds.), Calidad en el Desarrollo y Mantenimiento del Software. Ra-MA, 2002. 1053

Pérez, J., Carsí, J. A., Ramos, I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR) at ICSM'02. 2002. 1075

VOLUMEN 3: ARQUITECTURAS SOFTWARE (2002-2010)

PARTE V: ARQUITECTURAS SOFTWARE (2002-2010)

Prólogo de Patricio Letelier 1109

Ferrer, J., Lorenzo, M. A., Ramos, I.,Carsí, J. A., Pérez, J. Modeling Dynamic Aspects in Architectures and Multiagent Systems. Proc. of the Constraint Logic Programming and Software Engineering (CLPSE) Workshop at ICLP 2002. 2002. 1111

Pérez, J., Ramos, I., Lorenzo, M. A., Letelier, P., Jaén, J. PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos. Actas de las VII Jornadas de Ingeniería del Software y Bases de Datos. 2002. 1125

Ferrer, J., Lorenzo, M. A., Ramos, I., Carsí, J. A. Dinamismo en arquitecturas y sistemas multiagente. Actas de las V Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2002). 2002. 1137

Jaén, J., Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures. Proc. of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03). 2003. 1147

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. Proc. of the 3rd IEEE International Conference on Quality Software (QSIC 2003). 2003. 1153

Pérez, J., Ali, N., Ramos, I., Carsí, J. A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes. Workshop Desarrollo de Software Orientado a Aspectos (DSOA). Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1161

Pérez, J., Ali, N., Ramos, I., Pastor, J. A., Sánchez, P., Álvarez, B. Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003. 2003. 1167

Lorenzo, M. A., Peñarrubia, A., Carsí, J.A., Ramos, I. Modelando con UML Arquitecturas Software Reflexivas desde la Aproximación Orientada a Aspectos: Un Caso de Estudio. Actas de las VI Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software (IDEAS'2003). 2003. 1177

Navarro, E., Ramos, I. Requirements and Architecture: a marriage for Quality Assurance. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1189

Silva, J., Ali, N., Carsí, J. A.,Ramos, I. El aspecto de distribución de PRISMA. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003). 2003. 1199

Ali, N., Silva, J., Jaén, J., Ramos, I., Carsí, J. A., Pérez, J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proc. of the 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2003). 2003. 1209

Pérez J., Ali, N. Carsí, J. A., Ramos, I. Navarro, E. Designing Software Architectures with an Aspect-Oriented Language. ADVICE: Journal of Aspect Orientation, Vol. 1, No. 1, 2004. 1217

Navarro, E., Letelier, P., Ramos, I. Goals and Quality Characteristics: Separating

Concerns. Proc. of Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004. 2004. 1239

Navarro, E., Letelier, P., Ramos, I. UML Visualization for an aspect and goal oriented approach. Proc. of the 5th Aspect-Oriented Modeling Workshop (AOM'04) at UML 2004 Conference. 2004. 1247

Navarro, E., Ramos, I., Pérez, J. Goals Model Driving Software Architecture. 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004). 2004. 1255

Ali, N., Pérez, J., Ramos, I. High Level Specification of Distributed and Mobile Information Systems. Proc. of the Second International Symposium on Innovation in Information & Communication Technology (ISSICT2004). 2004. 1263

Ali, N., Carsí, J. A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2004. 1277

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004). 2004. 1285

Navarro, E., Letelier, P., Ramos, I., Álvarez, B. Orientación a aspectos y Orientación a objetivos: una propuesta para su integración. Taller de Desarrollo de Software Orientado a Aspectos, junto a JISBD'2004. 2004. 1293

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Dynamic Evolution in Aspect-Oriented Architectural Models. European Workshop on Software Architecture. 2005. 1299

Pérez, J., Ali, N., Costa, C., Carsí, J. A., Ramos, I. Executing Aspect-Oriented Component-Based Software Architectures on . NET Technology. Proc. of the 3rd International Conference on .NET Technologies. 2005. 1317

Ali, N., Ramos, I., Carsí, J. A. A Conceptual Model for Distributed Aspect-Oriented Software Architectures. Proc. of the International Conference on Information Technology Coding and Computing, ITCC 2005. 2005. 1329

Limón, R., Torres, J., Ramos, I. Separación de "Crosscutting Concerns" en el Desarrollo Orientado a Aspectos a Nivel de Diseño Arquitectónico. Actas del 6º Simposium Iberoamericano de Computación e Informática (SICI'05). 2005. 1335

Navarro, E., Letelier, P., Ramos, I. Integrating Expressiveness of Modern Requirements Modeling Approaches. Proc. of the 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005). 2005. 1345

Costa, C., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1355

Cabedo, R., Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Implementación Orientada a Aspectos en C# de un sistema robótico tele-operado. III Taller de Desarrollo de Software Orientado a Aspectos de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD). 2005. 1363

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas. Actas de las X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). 2005. 1371

Navarro, E., Letelier, P., Mocholí, J. A.,Ramos, I. A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems, XLVI(5), Special Issue on Systems Analysis and Design. 2006. 1379

Cabello, M. E., Costa, C., Ramos, I.,Carsí, J. A. Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System. Journal of the WSEAS Transactions on Information Science and Applications, Issue 10, Volume 3. 2006. 1391

Pérez, J., Ali, N., Carsí, J. A., Ramos, I. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. Proc. of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006). 2006. 1399

Pérez, J., Navarro, E., Letelier, P., Ramos, I. A Modelling Proposal for Aspect-Oriented Software Architectures. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). 2006. 1415

Navarro, E., Sánchez, P., Letelier, P., Pastor, J. A., Ramos, I. A Goal-Oriented Approach for Safety Requirements Specification. Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06). 2006. 1425

Limón, R., Ramos, I., Torres, J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development. Proc. of the International Conference the Computational Science and Its Applications (ICCSA 2006) . 2006. 1433

Limón, R., Ramos, I. Using Styles to Improve the Architectural Views Design. Proc. Of the International Conference on Software Engineering Advances (ICSEA 2006). 2006. 1443

Cabello, M. E., Costa, C., Ramos, I., Carsí, J. A. Generic Architecture of an Expert System for Diagnosis. Proc. of the 10th WSEAS International Conference on Computers. 2006. 1449

Cabello, M. E., Ramos, I., Costa, C. Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico. Actas del XIX Congreso Nacional y V Congreso Internacional de Informática y Computación del ANIEI-CIICC´07. 2006. 1455

Ali, N., Millan, C., Ramos, C. Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model. Proc. of the 8th International Sym-

posium on Distributed Objects and Applications (DOA). 2006. 1463

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes. IEEE Latin America Transactions, Special Edition JISBD'06, vol. 5, issue 4. 2007. 1481

Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J. A. Mobile Ambients in Aspect-Oriented Software Architectures. Proc. of the IFIP Working Conference on Software Engineering Techniques - SET 2006. 2006. 1489

Navarro, E., Letelier, P., Reolid, D., Ramos, I. Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques. Proc. of the 15th International Conference on Information System Development. 2006. 1501

Navarro, E., Letelier, P., Ramos, I. Requirements and Scenarios: running Aspect Oriented Software Architectures. Proc. of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). 2007. 1515

Limón, R., Ramos, I. Analyzing Styles of the Modular Software Architecture View. Proc. of the First European Conference on Software Architecture (ECSA 2007). 2007. 1521

Limón, R., Ramos, I. Relating Software Architecture Views by Using MDA. Proc. of the ICCSA 2007 International Conference on Computational Science and its Applications. 2007. 1525

Limón, R., Ramos, I., Morales, M. Zárate, J. Analyzing and Designing Software Architecture Views driven by their Relationships. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 1537

Pérez, J., Costa, C., Carsí, J. A.,Ramos, I. Verificación de Modelos Arquitectónicos Orientados a Aspectos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos. 2007. 1547

Costa, C., Ali, N., Pérez, J., Carsí, J. A., Ramos, I. Dynamic Reconfiguration of Software Architectures Through Aspects. Proc. of the First European Conference on Software Architecture (ECSA'07). 2007. 1557

Limón, R., Cabello, M. E., Ramos, I. Establishing Relations among Software Architecture Views through MDA for SPL. Proc. of the 14th International Congress on Computer Science Research (CIICC'07). 2007. 1563

Pérez, J., Ali, N., Carsí, J. A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J. A. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Journal of Information and Software Technology 50 (2008) . 2008. 1575

Ali, N., Solís, C., Ramos, I. Comparing architecture description languages for mobile software systems. Proc. of the First International Workshop on Software Architectures and Mobility at ICSE'08. 2008. 1597

Navarro, E., Gómez, A., Letelier, P., Ramos, I. MORPHEUS: a supporting tool for MDD. Proc. of the18th International Conference on Information Systems

Development (ISD2009). 2009. 1603

Ali, N., Ramos, I., Solís, C. Ambient-PRISMA: Ambients in Mobile Aspect-Oriented Software Architecture. Journal of Systems and Software 83. 2010. 1615

Pérez, J., Ramos, I., Carsí, J.A., Costa, C.: Model-Driven Development of Aspect-Oriented Software Architectures. Journal of Universal Computer Science, vol. 19, no. 10. 2013. 1637

VOLUMEN 4: INTERFACES DE USUARIO (2000-2002) - GESTIÓN DE MODELOS (2002-2006)

PARTE VI: INTERFACES DE USUARIO - IDEAS (2000-2002)

Prólogo de María Dolores Lozano 1701

Lozano, M.D., Ramos, I. An Object Oriented Approach for the Specification of User Interfaces. Proceedings of PACRIM'99 Conference. 2000. 1703

Lozano, M. D., Ramos, I., González, P. User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development". Proc. of the IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000). 2000. 1707

Lozano, M. D, González, P., Ramos, I. Models and Ontologies for the Specification of User Interfaces. Proc. of the Workshop on Model Engineering at ECOOP'00, 2000. 1717

Lozano, M. D., Ramos, I. Aproximación Basada en Modelos en la especificación de Interfaces de Usuario. Actas de las I Jornadas de Interacción Persona-Ordenador (INTERACCIÓN-20), 2000. 1725

Lozano, M. D., González, P., Ramos, I. User Interface Generation. Current Trends. INFORMATIK /INFORMATIQUE. Journal of the Swiss informatics societies. Vol. 2. 2001. 1733

Lozano, M. D., González, P., Ramos, I. IDEAS: An Object-Oriented Environment for developing User Interfaces". Proc. of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01). 2001. 1739

Lozano, M. D., González, P., Ramos, I., Montero, F., Molina, J. P. Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.16.2002. 1749

Lozano, M., González, P., Ramos, I. A Graphical User Interface Development Tool. 16th British Computer Society Conference on Human-Computer Interaction (BHCI'02). 1759

Lozano, M., González, P., Montero, F. Molina, J. P., Ramos, I. Integrating Usability within the User Interface Development Process of Web Applications. Proc. of the 2nd International Workshop on Web Oriented Software Technology at ECOOP'02. 2002. 1763

Montero, F. Lozano, M. D., González, P., Ramos, I. Designing Web Sites by Using Design Patterns. Proc. of SugarLoafPLoP 2002. 2002. 1779

PARTE VII: GESTIÓN DE MODELOS

Prólogo de José Ángel Carsí 1805

Carsí J.A, Ramos I., Silva J., Pérez J., Anaya V. Un Generador Automático de Planes de Migración de Datos. I+D Computación, Vol. 1, No. 1. 2002. 1809

Silva J.F., Carsí J.A., Ramos I. Comparación automática de esquemas conceptuales orientados a objeto, Revista Ingeniería Informática - Revista electrónica del Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción - Chile. 2002. 1821

Pérez J., Ramos I., Anaya V., Cubel J.M, Domínguez F., Boronat A., Carsí J.A. Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. Proc. of the Workshop on Software Evolution Through Transformations (SET'02) at ICGT2002. 2002. 1833

Silva, J., Carsí, J. A., Ramos, I. An algorithm to compare OO-Conceptual Schemas. Proc. of the International Conference on Software Maintenance (ICSM'2002). 2002. 1847

Pérez J., Carsí J.A, Ramos I. On the implication of application's requirements changes in the persistence layer: an automatic approach. Proc. of the Workshop on Database Maintenance and Reengineering (DBMR'2002) at the ICSM'02. 2002. 1857

Pérez, J., Carsí, J. A., Ramos, I. ADML: A Language for Automatic Generation of Migration Plans. Proc. of the First Eurasian Conference on Advances in Information and Communication Technology. 2002. 1869

Boronat, A., Carsí, J. A., Ramos, I. Una plataforma semántica para la gestión de modelos. Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos. 2003. 1879

Boronat, A., Pérez, J., Carsí, J. A.,Ramos, I. Two experiencies in software dynamics. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering. 2004. 1889

Boronat, A., Carsí, J. A., Ramos, I. An Algebraic Baseline for Automatic Transformations in MDA. Proc. of the Workshop on Software Evolution Through Transformations at ICGT2004. 2004. 1919

Boronat, A., Pedrós, J., Carsí, J. A., Ramos, I. Una Arquitectura para la Definición de Metáforas Gráficas para Metamodelos. Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004). 2004. 1935

Boronat, A., Carsí, J. A., Ramos, I., Pedrós, J. An Approach for Cross-Model Semantic Transformation on the . NET Framework. Proc. of the .NET Technologies'2005 Conference. 2005. 1947

Boronat, A., Boronat, A., Carsí, J. A.,Ramos, I. Una Arquitectura para la Definición

de Metáforas Gráficas para Metamodelos. Revista IEEE América Latina, Special Edition - JISBD'2004. 2005. 1957

Silva, J., Carsí, J. A., Ramos, I. Análisis teórico experimental de criterios de comparación de esquemas conceptuales orientados a objeto. Computación y Sistemas, Vol. IX no.1. 2005. 1967

Boronat, A., Carsí, J. A., Ramos, I. Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. Proc. of the 9th European Conference on Software Maintenance and Reengineering. 2005. 1983

Boronat, A., Carsí, J. A., Ramos, I. Automatic Support for Traceability in a Generic Model Management Framework. Proc. of the European Conference on Model Driven Architecture - Foundations and Applications. 2005. 1995

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). 2005. 2011

Boronat, A., Iborra, J., Carsí, J. A., Ramos, I., Gómez, A. Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. Actas de las X Jornadas sobre Ingeniería del Software y Bases de Datos - JISBD'05. 2005. 2019

Queralt, P., Hoyos, L., Boronat, A., Carsí, J. A., Ramos, I. Un motor de transformación de modelos con soporte para el lenguaje QVT Relations. Actas del Taller sobre Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). 2006. 2025

Boronat, A., Carsí, J. A., Ramos, I. Exogenous Model Merging by means of Model Management Operators. Proc. of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change at ICGT 2006. 2006. 2035

Boronat, A., Carsí, J. A., Ramos, I.,Letelier, P. Formal Model Merging Applied to Class Diagram Integration. Proc. of the International ERCIM Workshop on Software Evolution (2006). 2006. 2055

Boronat, A., Carsí, J. A., Ramos, I. Algebraic Specification of a Model Transformation Engine. Proc. of the Fundamental Approaches to Software Engineering Workshop (FASE'06) at ETAPS'06. 2006. 2061

Boronat, A., Ramos, I., Carsí, J. A. Definition of OCL 2. 0 Operational Semantics by means of a Parameterized Algebraic Specification. Proc. of the 1st International Workshop on Algebraic Foundations for OCL and Applications. 2006. 2077

Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J. A. An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework. Proc. of the Second European Conference on Model Driven Architecture. 2006. 2093

Gómez, A., Boronat, A., Hoyos, L., Carsí, J. A., Ramos, I. Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos.

Actas de las XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006). 2006. 2109

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I., Täubner, C., Eckstein, S. Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2119

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2129

Gómez, A., Boronat, A., Carsí, J. A., Ramos, I. Biological Data Transformation in Pathway Simulation. Actas de las VIII Jornadas Nacionales de Informática. Valencia. 2008. 2135

Lloret, E., Escobar, S., Palomar, M., Ramos, I. Incremental and adaptive Software Systems Development of Natural Language Applications. Proc. of the 22nd International Conference on Information Systems Development (ISD 2013). 2013. 2143

VOLUMEN 5: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

PARTE VIII: DESARROLLO DIRIGIDO POR MODELOS - LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

Prólogo de Emilio Insfrán y Silvia Abrahão 2177

Carsí, J. A., Insfrán, E., Abrahao, S., Genero, M., Piattini, M., Ramos, I. ¿Cómo utilizar experimentos controlados en la selección de transformaciones QVT para obtener diagramas de clases UML más fáciles de entender? Proc. of the 8th Argentine Symposium on Software Engineering (ASSE 2007). 2007. 2179

Limón, R. Ramos, I. Aragón, R. Transformación de Vistas Arquitectónicas Orientada por Modelos. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007. 2189

Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Marco de Trabajo basado en MDA para la Medición Genérica del Software. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007. 2199

Mora, B., García, F., Francisco Ruíz, Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Software Measurement by Using QVT Transformations in an MDA Context (ICEIS 2008). Proc. of the Tenth International Conference on Enterprise Information Systems, Volume DISI. 2008. 2209

Abrahao, S., Genero, M., Insfrán, E., Carsí, J. A., Ramos, I., Piattini, M. Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In Rech, J. and Bunse, C. (Eds.), Model-Driven Software Development: Integrating Quality Assurance. 2008. 2217

Insfrán, E., Carsí, J. A., Abrahao, S., Genero, M., Piattini, M., Ramos, I. Towards Quality-Driven Model Transformations: A Replication Study. Proc. of the International Workshop on Empirical Studies of Model-Driven Engineering (ESMDE 2008) at MODELS 2008. 2008. 2247

Cabello, M. E., Ramos, I. A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines. Proc. of the International Conference of Health Informatics- HEALTHINF´08. 2008. 2257

Cabello, M. E., Ramos, I. Variability Management in Product Lines for Decision Support Systems. Proc. of the 10th International Conference on Enterprise Information Systems (ICEIS´08). 2008. 2267

Cabello, M. E., Ramos, I. The Baseline: the milestone of Software Product Lines for Expert Systems automatic development. Proc. of the 9th Mexican International Conference on Computer Science. 2008. 2277

Gómez, M., Gómez, A., Cabello, M. E.,Ramos, I. BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. Actas del VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). 2009. 2287

Cabello, M. E., Ramos, I., Gómez, A., Limón, R. Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for Expert Systems development. Proc. of the 1st Asia Conference on Intelligent Information and Database Systems (ACIIDS´09). 2009. 2297

Cabello, M. E., Ramos, I. Expert Systems development through Software Product Lines techniques. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2303

Cabello, M. E., Ramos, I. Model-Driven Development of Decision Support Systems: tackling the variability problem. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008. 2315

Gómez, A., Ramos, I. Cardinality-based feature models and Model-Driven Engineering: fitting them together. Proc. of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. 2010. 2327

Gómez, A., Cabello, M. E., Ramos, I. BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. Proc. of the 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010) at SPLC 2010. 2010. 2335

Gómez, A., Ramos, I. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. Proc. of the 19th International Conference on Information Systems Development (ISD'10) 2010. 2343

Abrahao, S., Javier González, Insfrán, E.,Ramos, I. Software Evolution in Model-Driven Product Line Engineering. ERCIM News 88, Special Theme: Evolving Software. 2012. 2355

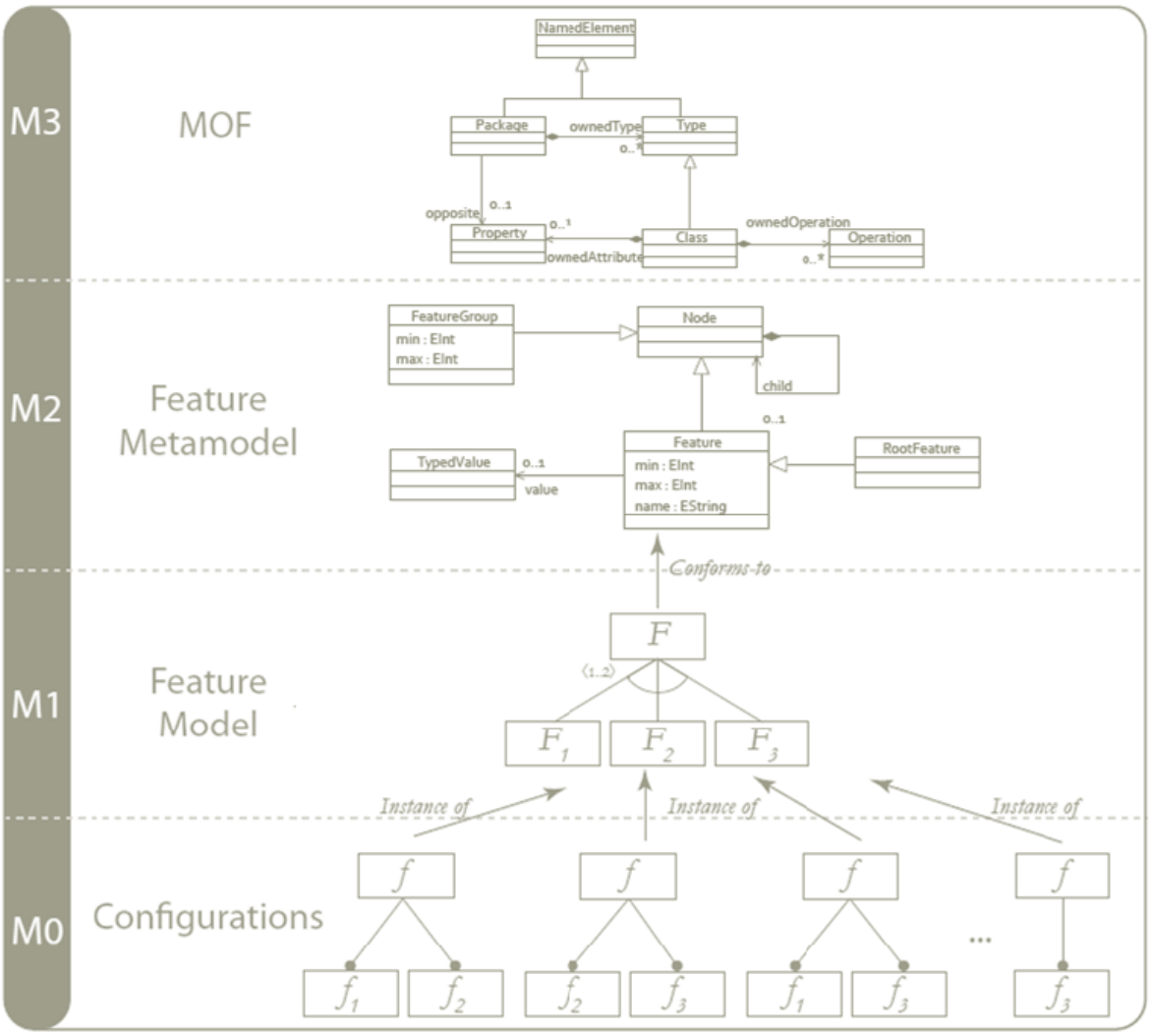
Cabello, M. E., Ramos, I., Gutiérrez, J. R., Gómez, A., Limón, R. SPL Variability Management, Cardinality and Types: an MDA Approach. International Journal of Intelligent Information and Database Systems (IJIIDS), Vol. 6, No. 2. 2012. 2357

Insfrán, E., Abrahao, S., Javier González, McGregor, J. D., Ramos, I. A Multimodeling Approach for Quality-Driven Architecture Derivation. Proc. of the 21st International Conference on Information Systems Development (ISD2012). 2012. 2383

PARTE IX: LOS AMIGOS

Los amigos 2397

Apéndice 2441



PARTE VIII

DESARROLLO DIRIGIDO POR MODELOS LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

PRÓLOGO DE EMILIO INSFRÁN Y SILVIA ABRAHÃO

DESARROLLO DIRIGIDO POR MODELOS Y LÍNEAS DE PRODUCTO SOFTWARE (2007-2012)

Parte VIII

La producción de software de calidad, en el tiempo adecuado y con unos costes razonables sigue siendo un problema abierto de la Ingeniería del Software que el profesor Isidro Ramos ha abordado desde distintas aproximaciones.

Esta sección describe algunos trabajos que representan la última etapa en la dilatada trayectoria académica e investigadora del profesor Isidro Ramos. Sin lugar a dudas el desarrollo de software dirigido por modelos (*MDE – Model-Driven Engineering*) y las Líneas de Producto Software han sido dos de sus áreas de investigación de interés donde ha realizado varias aportaciones. A modo de ejemplo se incluyen en esta sección del libro una selección de algunos artículos de investigación realizados en el contexto de dos proyectos: META (*Models, Environments, Transformations and Applications*) del 2006 al 2009 y MULTIPLE (*Multimodeling Approach For Quality-Aware Software Product Lines*), del 2010 al 2013.

En el contexto del proyecto META, además de coordinar el proyecto en el que han participado distintos grupos de investigación españoles, el profesor Ramos ha contribuido a la definición e implementación de un *framework* para la gestión de modelos y otras contribuciones tales como la definición de técnicas de transformación de vistas arquitectónicas orientadas por modelos, el uso de técnicas de desarrollo dirigido por modelos para el desarrollo de líneas de producto de sistemas expertos, técnicas de transformación de modelos dirigidas por atributos de calidad.

En el contexto del proyecto MULTIPLE, el profesor Ramos ha realizado aportaciones relacionadas a la definición e implantación de un marco tecnológico para el desarrollo de líneas de producto software de alta calidad. Entre otras aportaciones se puede mencionar la definición de un multimodelo que representa las distintas vistas de una línea de productos (variabilidad, funcional, calidad, transformaciones) así como los lenguajes específicos de dominio para especificar dichas vistas y relaciones entre ellas, y la definición de un plan de producción soportado por transformación de modelos para guiar el proceso de desarrollo de productos software específicos. Hay que destacar que estos trabajos se han realizado tras su jubilación voluntaria y que el profesor Ramos sigue contribuyendo de forma muy activa a las actividades del grupo.

No quisiéramos terminar este prólogo sobre la trayectoria investigadora del profesor Isidro Ramos sin hacer una referencia a su dimensión humana y su innegable espíritu marinero. Bajo una apariencia a veces seria subyace una persona que nos cautiva a todos con su infinita curiosidad intelectual y cultural.

Es evidente que con todos los rasgos descritos de la personalidad de Isidro Ramos, no podemos sino concluir que ha sido un privilegio para nosotros, sus colaboradores, poder haber contado durante estos años (y esperamos seguir haciéndolo por muchos más) con su apoyo constante, compañía y amistad.

Isidro, eres una inspiración para todos nosotros y tienes toda nuestra admiración y gratitud.

Silvia Abrahão y Emilio Insfrán

¿Cómo utilizar experimentos controlados en la selección de transformaciones QVT para obtener diagramas de clases UML más fáciles de entender?

José Ángel Carsí[1], Emilio Insfrán[1],
Silvia Abrahão[1], Marcela Genero[2], Mario Piattini[2], Isidro Ramos[1]

[1]Grupo de investigación ISSI,
Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022, Valencia
{pcarsi, einsfran, sabrahao, iramos}@dsic.upv.es
[2]Grupo de investigación ALARCOS,
Departamento de Tecnologías y Sistemas de Información, Universidad de Castilla La-Mancha
Paseo de la Universidad Nº 4, 13071, Ciudad Real
{Marcela.Genero, Mario.Piattini}@uclm.es

Resumen. Este trabajo forma parte de un proyecto que estamos desarrollando, cuyo objetivo es la automatización de las transformaciones entre modelos en el contexto de procesos MDA, guiada por atributos de calidad. El presente trabajo presenta un experimento para seleccionar transformaciones guiadas por atributos de calidad cuando existen alternativas. Se centra en un conjunto de reglas de transformación para producir relaciones estructurales entre clases (asociación, agregación y clase asociación) a partir de un modelo de requisitos y el atributo de calidad facilidad de entendimiento. El objetivo del experimento es obtener evidencia empírica sobre qué transformación alternativa produce el diagrama de clases UML que es más fácil de entender. Los resultados demuestran que hay una tendencia en el uso de las relaciones de asociación para guiar estas transformaciones. Sin embargo, más experimentación es necesaria para obtener resultados concluyentes.

Palabras clave: MDA, transformaciones, UML, QVT, diagrama de clase, entendibilidad, experimento

1 Introducción

Model-driven architecture (MDA) es una aproximación emergente que promueve el uso de modelos durante el proceso de desarrollo y permite que éstos puedan ser transformados hasta la obtención del código fuente del producto software final [12]. Por lo tanto, en este enfoque, los modelos ya no son simples medios para describir software, sino la pieza fundamental de su desarrollo. Como consecuencia, la calidad de los modelos tiene una gran importancia, ya que determinará la calidad de los productos software finalmente desarrollados.

Carsí, J. A., Insfrán, E., Abrahao, S., Genero, M., Piattini, M., Ramos, I. ¿Cómo utilizar experimentos controlados en la selección de transformaciones QVT para obtener diagramas de clases UML más fáciles de entender? Proc. of the 8th Argentine Symposium on Software Engineering (ASSE 2007). 2007.

En general, hay varias maneras de transformar un modelo origen a un modelo destino. Los modelos destino alternativos pueden tener la misma funcionalidad, pero pueden ser muy distintos en cuanto a sus atributos de calidad. Uno de los desafíos de un proceso de transformación automático es identificar las reglas de transformación que producirán el modelo destino que satisface en mayor grado los atributos de calidad deseados.

En este artículo, se presenta un experimento controlado para investigar la selección de transformaciones alternativas para obtener diagramas de clase UML a partir de un modelo de requisitos [5] [6]. Se centra en un conjunto de reglas de transformación para obtener relaciones estructurales entre clases (asociación, agregación y clase asociación) y el atributo de calidad 'facilidad de entendimiento'. El objetivo del experimento es obtener evidencia empírica sobre qué transformación alternativa produce el diagrama de clase UML que es más fácil de entender.

Las transformaciones para obtener relaciones estructurales entre clases han sido seleccionadas debido a que éstas constituyen la fuente principal de variación e impacto en el diseño cuando se define un diagrama de clases UML. La facilidad de entendimiento ha sido seleccionada porque es el principal atributo de calidad que influencia la mantenibilidad. Un diagrama de clases debe ser bien entendido antes de que cualquier cambio pueda ser identificado o realizado por el analista.

Además, la evaluación empírica de las reglas de transformación es particularmente importante cuando éstas se aplican de forma automática. El conjunto seleccionado de transformaciones se define usando el estándar Query-View-Transformations (QVT) [14] y se ejecuta en la plataforma de gestión de modelos MOMENT [2].

Este artículo se organiza como sigue. La sección 2 discute los trabajos existentes sobre calidad en la transformación de modelos. La sección 3 presenta la aproximación para transformar un modelo de requisitos en diagramas de clases UML. La sección 4 describe el diseño del experimento para validar la selección de las transformaciones alternativas según el atributo facilidad de entendimiento. La sección 5 presenta el análisis de datos y las lecciones aprendidas. Finalmente, la sección 6 presenta las conclusiones y trabajo futuro.

2 Trabajos relacionados

En los últimos años se han propuesto, en el ámbito de la Ingeniería del Software, algunos trabajos que tratan la calidad de las transformaciones desde la perspectiva de algún atributo de calidad. La tabla 1 presenta un resumen de estas propuestas organizadas cronológicamente. Una comparación más detallada se puede encontrar en [1]. La tabla 1 revela que algunas propuestas se centran en definir transformaciones horizontales para el refactoring de modelos [11] [10] [7]. Otras propuestas proporcionan transformaciones verticales para el refinamiento de modelos [15], síntesis [8] [9], o ingeniería inversa [18]. De estos estudios, sólo el de Kurtev [9] presenta una propuesta más sistemática para la selección de transformaciones alternativas en base al atributo de calidad adaptabilidad.

Estas aproximaciones plantean algunos criterios de calidad que se pueden utilizar para dirigir las transformaciones, pero sin embargo, muy pocas ilustran la selección

de transformaciones por medio de ejemplos prácticos [9] [10]. Además, a excepción de Markovic y Baar [10] y Kurtev [9], las transformaciones no están bien definidas. Por lo tanto, son necesarias aproximaciones más sistemáticas para asegurar la calidad de los artefactos software obtenidos como parte de un proceso basado en transformación de modelos.

Tabla 1. Clasificación de propuestas para la calidad de las transformaciones.

Propuesta	Objetivo	Tipo de transformac.	Artefacto de entrada	Atributos de Calidad	Automatizado
Zou y Kontogiannis, 2003 [18]	Ingeniería Inversa (migración)	Vertical (CM-a-PIM)	Programa fuente	Acoplamiento y cohesión	No
Rottger y Zschaler, 2004 [15]	Refinamiento	Horizontal	Modelos de contexto	Tiempo de respuesta	Parcial-mente
Merilinna, 2005 [11]	Refactoring	Horizontal (PIM-a-PIM)	Modelos arquitecturales	Rendimiento, disponibilidad, fiabilidad, mantenimiento y reusabilidad	Si
Kurtev, 2005 [9]	Síntesis	Vertical (PIM-a-PIM)	Diagramas de clase UML	Adaptabilidad	Si
Markovic y Baar, 2005 [10]	Refactoring	Horizontal (PIM-a-PIM)	Diagramas de clase UML	Correctitud semántica	No
Sottet et al., 2006 [16]	–	–	Modelos de Interfaz	Compatibilidad, protección de error, consistencia	No
Ivkovic y Kontogiannis, 2006 [7]	Refactoring	Horizontal (PIM-a-PIM)	Modelos arquitecturales expresados en UML	Mantenimiento, rendimiento y seguridad	No
Kerhervé et al., 2006 [8]	Síntesis, refinamiento	Horizontal y vertical	Modelos de información	Portabilidad de servicios, tiempo de respuesta, retraso de la red	No

(–) indica que la propuesta no proporciona esta información

Otra debilidad de estas propuestas es que la selección de transformaciones alternativas no ha sido validada empíricamente. La aplicabilidad práctica de las transformaciones es comúnmente descrita en base a la intuición de los investigadores. Según lo indicado por Czarnecki y Helsen [3], hay una carencia de experimentos controlados para validar las observaciones realizadas por los investigadores en el campo de la Ingeniería de Modelos. Intentando paliar las carencias mencionadas, en este trabajo, se presenta una propuesta concreta para validar empíricamente la selección de transformaciones alternativas dirigida por atributos de calidad.

3 Transformación de requisitos en diagramas de clases UML

El modelo de requisitos [5] [6] define la estructura y el proceso seguido para capturar los requisitos del software. Se compone de un Árbol de Refinamiento de las Funciones (FRT) para especificar la descomposición jerárquica de las funciones del sistema; un modelo de Caso de Uso (UC) para especificar la comunicación y la funcionalidad; y de Diagramas de Secuencia (SD) para especificar las interacciones entre las entidades para realizar cada caso de uso. La herramienta CASE RETO (http://reto.dsic.upv.es) da soporte a este método. Siguiendo una estrategia MDA de transformación de modelos, una vez especificado el modelo de requisitos, se puede obtener un diagrama de clases UML aplicando un conjunto de reglas de transformación [5].

Una regla de transformación está compuesta de un patrón de condición y una acción. Algunas reglas de transformación son sencillas de aplicar una vez que se ha satisfecho el patrón de condición de la regla (relaciones uno a uno). Por ejemplo, la generación de clases del diagrama de clases UML se basa en el análisis de actores y de las entidades que participan en todos los diagramas de secuencia. Cuando se aplica esta regla de transformación (TR2), una clase es generada en el diagrama de clases para cada entidad de tipo clase distinta que participa en cualquier diagrama de secuencia. Sin embargo, otras reglas no son fáciles de identificar y aplicar debido a tres razones principales: la complejidad del patrón de condición de la regla de transformación, la satisfacción de varias reglas debido a condiciones de transformación no disjuntas, y las posibles múltiples representaciones válidas para un elemento del modelo de requisitos en el diagrama de clases.

El metamodelado es un concepto fundamental del paradigma MDA y es usado en la Ingeniería del Software para describir las abstracciones que definen los modelos y las relaciones entre ellas. Básicamente, un metamodelo se puede representar como un diagrama de clases cuyas clases y asociaciones representan los conceptos y las relaciones entre conceptos. En este contexto, MOF (Meta-Object Facility) [13] proporciona un marco para definir un metamodelo y realizar consultas y manipulaciones sobre el mismo.

A continuación se detallan las partes más relevantes del metamodelo de requisitos. Las funciones del sistema son representadas por la clase UseCase. Cada caso de uso se especifica detalladamente por medio de uno o varios diagramas de secuencia. Los diagramas de secuencia (SequenceDiagram) se componen principalmente de entidades (entity) y mensajes (message). Para describir un diagrama de secuencia, se distinguen tres tipos de entidades: actor, interfaz y clase. El actor representa a usuarios del caso del uso; interfaz representa la frontera entre los actores y las clases del sistema; y clase representan las diversas clases del sistema que participan en la realización del caso de uso.

Finalmente, para caracterizar el tipo de interacción entre objetos, se utilizan cuatro tipos de mensajes: signal, service, query y connect. Los mensajes signal representan la interacción entre los actores y el interfaz del sistema. Los mensajes service representan interacciones entre objetos que tienen el propósito de modificar el estado del sistema (crear, borrar o modificar). Los mensajes query representan interacciones entre objetos necesarias para consultar el estado de un objeto o un conjunto de objetos.

Las reglas de transformación a aplicar para una especificación de requisitos que generan diferentes diagramas de clases. Aunque no todas las posibilidades se pueden explicar con detalle, debido a la limitación de espacio, es posible ver que dado una especificación de requisitos varios diagramas de clase válidos pueden ser obtenidos aplicando distintas reglas de transformación.

El ejemplo usado para ilustrar estas transformaciones alternativas es el de un sistema de alquiler de coches. Un diagrama de secuencia se utiliza para especificar las interacciones entre objetos que son necesarias para realizar el caso de uso Crea Seguro. Este caso de uso es iniciado por un actor Administrador y representa la creación de una póliza de seguro de coche que se contrata con una compañía de seguros para poder alquilar los coches. La Figura 1 muestra el diagrama de secuencia para este caso de uso. Se introducen los datos y se crea un nuevo objeto Seguro que tiene que estar relacionado con su Compañía de Seguros y con el Coche para el que se ha contratado el seguro.

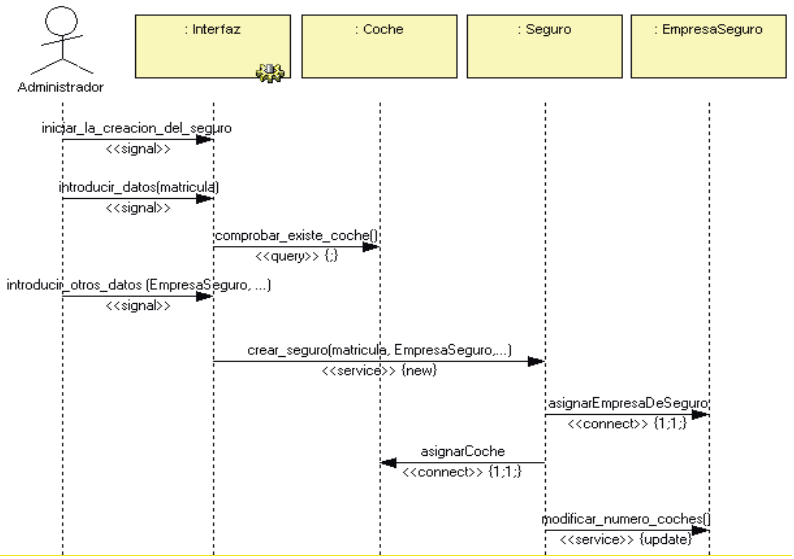


Fig. 1. Diagrama de Secuencia mostrando las interacciones entre objetos (estereotipadas) para el caso de uso Crear Seguro.

Siguiendo una aproximación MDA, se utiliza esta información para transformar la especificación de requisitos en un diagrama de clases. Este proceso de transformación debe decidir qué transformaciones son más convenientes de aplicar para producir el resultado esperado por el analista o un resultado que maximice un atributo de calidad (por ejemplo la entendibilidad).

Analizando el caso de uso Crear Seguro, especificado como interacciones entre clases en la figura 1, se observa que se puede aplicar la regla TR15 de transformación (“Por cada mensaje etiquetado con el estereotipo «connect», generar una relación de asociación entre las clases participantes”) y, consecuentemente, establecer relaciones de asociación entre Seguro y Coche y entre Seguro y Compañía de Seguros como se muestra en la Figura 2a.

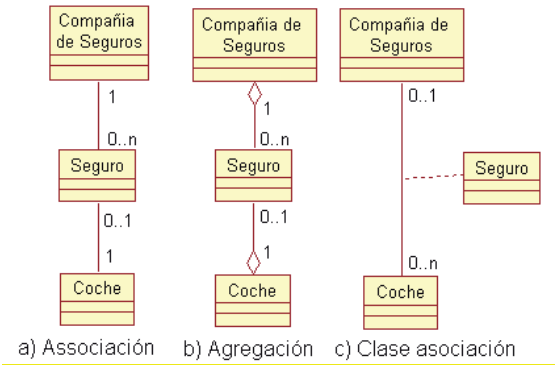


Fig. 2. Parte de diagramas de clases obtenidos por transformación.

Alternativamente, hay otras interpretaciones que pueden realizarse sobre las interacciones entre objetos para la construcción del diagrama de clases. Por ejemplo, una segunda alternativa, podría representar el Seguro como componente en relaciones de agregación con las clases Compañía de Seguros y Coche (Figura 2b, agregación). Una tercera alternativa podría representar el Seguro como una clase asociación entre la Compañía de Seguros y el Coche (Figura 2d, clase asociación).

En definitiva, se han detectado tres tipos de relaciones estructurales para representar estas interacciones entre objetos (por supuesto puede haber otras representaciones válidas) y por tanto, distintas reglas de transformación deben ser aplicadas para producir estas relaciones. En la Tabla 2, se presentan estas relaciones estructurales y las reglas de transformación alternativas que pueden ser aplicadas.

Tabla 2. Relaciones estructurales y sus correspondientes reglas de transformación.

Alternativas	**Reglas de Transformación**
A1 (asociación)	TR 14. Proceso de mensajes «service/new» TR 15. Proceso de mensajes «connect» TR 16. Proceso de mensajes «service/new» o «connect» con clases usando nombre de roles
A2 (agregación)	TR 28. Proceso de mensajes «service/new»
A3 (clase asociación)	TR 39. Proceso de mensajes «service/new» con mensajes «connect» TR 40. Proceso de mensajes «connect» con mensajes «service/new»

4 Descripción del experimento

Esta sección presenta el proceso experimental [17] seguido en la realización del experimento controlado. El principal objetivo del experimento es investigar cual de las alternativas de las reglas de transformación para relaciones estructurales entre

clases introducido en la sección 3.3 (A1, A2, A3) permite obtener el diagrama de clases más fácil de entender. Los participantes del experimento fueron 39 estudiantes de cuarto curso de la Ingeniería Informática de la Universidad Politécnica de Valencia, que estaban cursando un segundo curso de Ingeniería de Software. Se consideraron una "muestra por conveniencia" (es decir todos los estudiantes disponibles en la clase). Los sujetos tenían seis meses de experiencia en modelar con UML y cierta experiencia en el paradigma OO. Los sujetos fueron incentivados a participar ofreciéndoles un punto extra en la nota final por realizar las tareas requeridas correctamente. Las variables independientes son las reglas de la transformación de las relaciones estructurales entre las clases (A1, A2, A3). La variable dependiente es la Facilidad de Entendimiento (Understandability). El material y las tareas experimentales consistieron en:

- 9 diagramas de secuencia sobre tres casos de estudio, con 4 diagramas de la clase UML, cada uno. Cada diagrama de clase fue obtenido aplicando las reglas de transformación alternativas. El material experimental está disponible en: www.dsic.upv.es/~einsfran/experiment.
- Cada diagrama de secuencia tenía un cuestionario adjunto que consistía en 6 preguntas de tipo "Si/No" para comprobar si los sujetos entendían los diagramas de secuencia. La eficacia de los sujetos para contestar las preguntas (número de respuestas correctas/número de respuestas) se utilizó para excluir las observaciones que no satisfacían un nivel mínimo de la calidad. Las observaciones con un valor menor que 0,5 fueron excluidas. Consideramos que si los sujetos no entendían el diagrama de secuencia, no podían realizar el resto de tareas correctamente.
- Cada uno de los tres diagramas de clase UML tenía un cuestionario adjunto (con 6 preguntas) para determinar qué alternativa de diagrama de la clase UML mejor entendido por los sujetos. Además, los sujetos tenían que anotar el tiempo de comienzo y fin para contestar los cuestionarios. De esta tarea se obtuvieron tres medidas para el facilidad de entendimiento:
  - Tiempo de Entendimiento, que refleja el tiempo, en segundos, que los sujetos tardaron en contestar a cada cuestionario (calculado por la diferencia entre la hora de finalización y la hora de comienzo). Cada sujeto contestó tres cuestionarios uno para cada alternativa (A1, A2 y A3), obteniendo así tres medidas del Tiempo Entendimiento (A1Tiempo, A2Tiempo y A3Tiempo).
  - Eficacia, que refleja la corrección de las respuestas (calculadas dividiendo el número de las respuestas correctas por el número de respuestas). Así, se obtuvieron tres medidas de la Eficacia (A2Eficacia, A2Eficacia y A3Eficacia).
  - Eficiencia, que refleja la corrección de las respuestas por unidad de tiempo (calculada dividiendo el número de las respuestas correctas por el Tiempo de Entendimiento). De esta manera, se obtuvieron tres medidas para la Eficiencia (A2Eficiencia, A2Eficiencia y A3Eficiencia).
- La última tarea consistía en preguntar a los sujetos cuál de los tres diagramas de clase UML reflejaba mejor el problema modelado en el diagrama de secuencia. De esta manera, obtuvimos una medida subjetiva (Alternativa Seleccionada) basada en la opinión de los sujetos.

  Se plantearon las siguientes hipótesis:

- $H1_0$: El uso de diferentes transformaciones alternativas (A1, A2, A3) no afecta al Tiempo de Entendimiento (A1Tiempo, A2Tiempo y A3Tiempo). $H1_1 = \neg H1_0$
- Análogamente se definen hipótesis para las variables relacionadas con la Eficacia y la Eficiencia.

El experimento comenzó con una sesión introductoria en la cual se refrescaron los conceptos principales de los modelo de requisitos (es decir, la notación de los diagramas de secuencia). A cada sujeto se le entregó todo el material, con los nueve diagramas de secuencia (diseño balanceado intrasujetos). Los diagramas se entregaron en diferente orden para evitar el efecto de aprendizaje. Se explicó a los sujetos cómo realizar las tareas experimentales, y tuvieron dos horas para realizar todas las tareas.

Después de realizar el experimento, se recogieron los datos del mismo. En primer lugar, se hizo una "limpieza de datos", excluyendo aquellas observaciones que no estaban completas porque no habían anotado el tiempo o porque no habían seleccionado la mejor alternativa. Todas las preguntas fueron contestadas en cada cuestionario, asegurando así la completitud de las tareas realizadas. Por otra parte, se excluyeron las observaciones que tuvieron, para cada diagrama de secuencia, un valor de eficacia menor que 50%. Finalmente, se consideraron 323 observaciones para contrastar las hipótesis.

5 Análisis de datos

La transformación más seleccionada por los sujetos fue A1, como muestra la figura 3. Esto significa que los sujetos creyeron que el uso de asociaciones permitió obtener el mejor diagrama de clases (el más fácil entender). Las clases de asociación, representadas por la alternativa A3, fue la menos elegida, hecho que revela que puede ser la alternativa de alternativa menos apropiada.

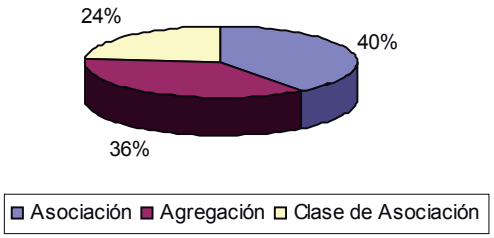


Fig. 3. Porcentajes por Alternativa Seleccionada.

Los estadísticos descriptivos para el Tiempo de Entendimiento, la Eficacia y la Eficiencia muestran que en promedio, los sujetos tardaron menos en realizar las tareas relacionadas con la alternativa A2; aunque realmente la diferencia con las otras no es muy significativa En promedio, los sujetos fueron más eficaces y eficientes realizando las tareas relacionadas con la alternativa A1, pero las diferencias en la eficacia con las otras alternativas no son demasiado muy significativas.

Para contrastar las hipótesis presentadas en la sección 4 se realizó un ANOVA para medidas repetidas. Los resultados obtenidos permitieron rechazar la hipótesis $H1_0$, $H2_0$, y $H3_0$ (con un nivel de significación = 0,05), lo que significa que el uso de diferentes transformaciones alternativas realmente afecta al Tiempo de Entendimiento, a la Eficacia y a la Eficiencia.

Resumiendo, los resultados principales obtenidos a partir de este experimento son:

- Los estadísticos descriptivos muestran una leve tendencia a favor de A1, la transformación basada en asociaciones. Sorprendentemente, la diferencia entre los valores mínimos y máximos del Tiempo es muy significativa. Hecho que puede ser debido a que los sujetos eran modeladores novatos.
- La transformación más seleccionada por los sujetos, es la que se basa en asociaciones.

6 Conclusiones y trabajos futuros

En este artículo se ha presentado cómo es posible utilizar experimentos controlados para seleccionar las transformaciones más adecuadas, cuando en un proceso de transformación existen varias alternativas posibles.

El experimento se diseñó para averiguar qué alternativa era la que mejoraba la facilidad de entendimiento de los diagramas de clases obtenidos a través de transformaciones, partiendo del modelo de requisitos. Se tuvieron en cuenta sólo aquellas transformaciones relacionadas con las relaciones estructurales entre clases.

El resultado más relevante es que la alternativa basada en asociaciones (A1) es la más efectiva y eficiente considerando tanto las medidas objetivas como las subjetivas de la facilidad de entendimiento. Una posible razón para ello podría ser la menor carga semántica que tiene la asociación frente al resto de relaciones.

Una limitación de este estudio es que las tres transformaciones alternativas no pueden usarse simultáneamente en todas las situaciones de modelado. Por ejemplo, para establecer una clase asociación (A4) es necesario tener al menos un mensaje «service/new» y dos mensajes «connect» en el modelo origen. El objetivo del experimento realizado en este trabajo fue obtener evidencia empírica cuando las tres alternativas para un caso específico pueden ser utilizadas indistintamente para obtener una relación entre clases. Somos conscientes de que hay más de tres alternativas para representar relaciones estructurales entre clases. Es necesario realizar más experimentos con estas y otras alternativas.

Otra limitación es el uso de estudiantes como sujetos experimentales. En general los estudiantes no tenían experiencia práctica en modelado conceptual con UML. Este hecho puede afectar la validez externa del experimento. Sin embargo, los sujetos fueron estudiantes de cuarto curso de Ingeniería Informática, por lo podemos considerarlos analistas novatos.

Estos resultados aunque preeliminares proveen evidencia empírica que servirá para definir un modelo de calidad para dirigir la transformación de modelos. Se va a replicar el experimento con otros estudiantes de la Universidad de Castilla-La Mancha y modeladores con más experiencia. Una mayor experiencia en modelado podría influir en el rendimiento de los sujetos. Como trabajo futuro se presente evaluar el

resto de transformaciones del catálogo de reglas [5] considerando otros atributos de calidad (p.e., usabilidad, modificabilidad).

Agradecimientos. Este trabajo es parte de los proyectos META TIN2006-15175-C05-05 y la red CALIPSO (TIN20005-24055-E) ambos financiados por la Ministerio de Educación y Ciencia (España).

Referencias

1. Abrahão S., Genero, M., Insfran E., Carsí J.A., Ramos I., Piattini M.: Quality-Driven Model Transformations: From Requirements to UML Class Diagrams Model-Driven Software Development: Integrating Quality Assurance, Idea Group (2007).
2. Boronat A., Carsí J.A., Ramos I.: Algebraic Specification of a Model Transformation Engine. Proc. of the Fundamental Approaches to Software Engineering (FASE'06). ETAPS'06. Vienna, Austria, (2006) 262–277.
3. Czarnecki K., Helsen S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), (2006) 621–645.
4. Gómez A., Boronat A., Hoyos L., Ramos I., Carsí J.Á.: Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos, XI Jornadas de Ingeniería del Software y Bases de Datos. JISBD'06. Octubre, 2006. Sitges, Barcelona.
5. Insfran E.: A Requirements Engineering Approach for Object-Oriented Conceptual Modeling, PhD Thesis, DSIC, Valencia University of Technology (2003).
6. Insfran E., Pastor O. and Wieringa R.: Requirements Engineering-Based Conceptual Modelling. Requirements Engineering, 7(2), (2002) 61–72, Springer-Verlag.
7. Ivkovic I., Kontogiannis K.: A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations, Proc. of the Conference on Software Maintenance and Reengineering, CSMR 2006.
8. Kerhervé B., Nguyen K.K., Gerbé O., Jaumard B.: A Framework for Quality-Driven Delivery in Distributed Multimedia Systems, Proc. of the Advanced Int. Conference on Telecommunications and Int. Conference on Internet and Web Applications and Services (AICT/ICIW 2006), (2006) 195–205.
9. Kurtev I.: Adaptability of Model Transformations. PhD Thesis, Univ. of Twente, (2005).
10. Markovic S., Baar T.: Refactoring OCL annotated UML class diagrams. 8th Int. Conference on Model Driven Engineering Languages and Systems (2005) 280–294.
11. Merilinna J.: A Tool for Quality-Driven Architecture Model Transformation. Espoo, VTT Electronics, VTT Publications (2005).
12. OMG: MDA Guide. Version 1.0.1, (2003).
13. OMG: Meta Object Facility (MOF) 2.0 Core Specification, ptc/04-10-15 (2004).
14. OMG: QVT MOF 2.0 QVT Final Adopted Specification, OMG 2005.
15. Rottger S., Zschaler, S.: Model-Driven Development for Non-functional Properties: Refinement Through Model Transformation, The Unified Modelling Language Conference, LNCS Volume 3273, (2004) 275–289.
16. Sottet J. S., Calvary G., Favre J.M.: Mapping Model: A First Step to Ensure Usability for sustaining User Interface Plasticity, Proc. of the Workshop on Model Driven Development of Advanced User Interfaces (2006).
17. Wohlin, C., P. Runeson, M. Hast, M. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering: an Introduction, Kluwer, 2000.
18. Zou Y., Kontogiannis K.: Quality Driven Transformation Framework for OO Migration. Proc. 2nd ASERC Workshop on Software Architecture, Banff, Canada (2003) 18–19.

Transformación de Vistas Arquitectónicas Orientada por Modelos

Rogelio Limón Cordero[1], Isidro Ramos Salavert[1], Arturo Aragon Sorroza[2]
[1]*Departamento de Sistemas y Computación, Universidad Politécnica de Valencia,*
Camino de Vera s/n E-46071 Valencia – España
{rlimon,iramos}@dsic.upv.es
[2]*Centro de Computo, Instituto Tecnológico de Oaxaca*
Av. Victor B. Ahuja 125 esq. Calz. Tecnológico, Oaxaca, México
arturoar@itoaxaca.edu.mx

Resumen

Las vistas arquitectónicas permiten la representación y documentación de las estructuras estáticas y dinámicas de la arquitectura de software de sistemas complejos, son "los planos arquitectónicos" del sistema con los cuales se logran hacer análisis previos al diseño detallado, y posterior a la implementación realizar seguimientos en la evolución y mantenimiento del sistema. Sin embargo dada la diversidad de propuestas de modelos existentes para las vistas arquitectónicas se requiere establecer vínculos entre éstas para lograr, por un lado emigrar entre un modelo y otro, y por otro lado generar varias instancias arquitectónicas de una manera más rápida. Por tal motivo se propone llevar a cabo la transformación de los modelos que soportan las vistas arquitectónicas utilizando una transformación entre éstas mediante la Arquitectura Orientada por Modelo (MDA: Model Driven Architecture) propuesta por el OMG, que posibilita la transformación entre modelos que están al mismo nivel o a diferente nivel de abstracción.

1. Introducción.

Las vistas arquitectónicas permiten separar los diferentes asuntos de interés que se presentan al desarrollar un sistema de software, modelando las estructuras de la arquitectura. Lo cual permite tomar decisiones oportunas antes del diseño detallado y constituye una de las primeras referencias cuando se realizan tareas de mantenimiento o cuando el sistema evoluciona.

Cada estructura del sistema que es representada por una vista arquitectónica establece correspondencias entre sí que las vincula por ser parte del mismo sistema. Sin embargo estas correspondencias no son claramente establecidas por los distintos enfoques que modelan a las vistas, y mucho menos se conservan a estos vínculos en la especificación arquitectónica.

Por lo que la propuesta que se presenta en este trabajo va encaminada a la especificación de las correspondencias a nivel de meta-modelos arquitectónicos de una vista con otra, con el propósito de mantener una traza entre los meta-modelos de las vistas mediante la cual se pueda generar el modelo de una vista a partir de un modelo de otra vista. Esto se propone hacerlo mediante la arquitectura orientada por modelos (MDA).

El desarrollo orientado por modelos se está consolidando como un nuevo paradigma de desarrollo a través del MDA, que ofrece posibilidades de desarrollo independiente de la plataforma tecnológica y re-uso, realizando transformación entre modelos que pueden estar a un mismo nivel o a diferente nivel de abstracción. El MDA proporciona el soporte y los mecanismos para la transformación a nivel de meta-modelos entre enfoques de modelado de software distinto.

El trabajo que se presenta está organizado de la siguiente manera, en la sección 2 se comienza por clarificar y analizar la relación entre el MDA con la arquitectura de software (AS), puesto que las dos áreas tratan con modelos (arquitectónicos) empleando incluso términos semejantes (arquitectura, vistas, modelos). En la sección tres se hace un esbozo de la propuesta. En la sección cuatro se exponen los diferentes modelos de vistas arquitectónicas que se han realizado, las relaciones que hay entre ellos y se

Limón, R. Ramos, I. Aragón, R. Transformación de Vistas Arquitectónicas Orientada por Modelos. Actas de las Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07). 2007.

presentan los meta-modelos de dos vistas usando la notación de UML 2.0. En la sección cinco se expone el soporte para llevar a cabo las transformaciones mediante un lenguaje (QVT) y en una notación gráfica se especificarán las correspondencias entre los meta-modelos de las dos vistas modeladas (modular y componente-conector). En la sección seis se ilustra con un caso de estudio y por último se exponen las conclusiones.

2. El desarrollo orientado por modelos y la arquitectura de software.

El desarrollo de software mediante la ingeniería orientada por modelos (MDE: Model-Driven Engineering) es una propuesta para enfrentar a la complejidad introducida por la diversidad tecnológica existente, cuya estrategia consiste en capturar los elementos de un modelo mediante meta-modelos, y genera otros al mismo o diferente nivel de abstracción mediante un mecanismo de transformación entre éstos [1].

Una de las propuestas de MDE hecha por el OMG (Object Management Group) es el MDA [2] la cual tiene como soporte a MOF para la especificación de meta-modelos y llevar a cabo las transformaciones entre modelos. Este enfoque al igual que la AS modela arquitecturas pero con diferente intencionalidad, aunque existan algunas semejanzas entre ellas.

2.1. MDA y AS.

La diferencia entre el MDA y la AS parte desde el concepto que maneja sobre arquitectura y su diferencia se hace más evidente en los propósitos que persigue cada área.

En cuanto al concepto de arquitectura, el MDA hace una generalización de la referencia que toma de [3] como base. Considera como arquitectura a cualquier modelo que especifique las "partes" del sistema y las reglas que establecen sus interacciones entre ellas mediante conectores [2], lo cual debe ser expresado formalmente o a través de una semántica claramente definida como por ejemplo el UML 2.0.

En el caso de la AS, la arquitectura es considerada como *el conjunto de estructuras que forman parte del sistema, cuyos elementos de software ("partes") sólo muestran sus propiedades externamente visibles y estos elementos están relacionados entre sí* [4].
De tal manera que si se ocupa por ejemplo un modelo como el *entidad-relación* (ER), éste es considerado como una arquitectura por el MDA donde las "partes" son las *entidades* y *atributos*, sus conectores e interacciones son sus ligas de *relaciones,* y las reglas son las que definen cómo deben establecerse las *relaciones*, en cambio en la AS este modelo ER no es considerado como una AS, puesto que sus elementos son de un nivel de abstracción superior al del software. Sin embargo un modelo de clases (orientación a objetos) es considerado como arquitectura en ambos enfoques.

Es importante señalar que cualquier arquitectura de software es considerada como una arquitectura en el MDA, pero no todas las arquitecturas del MDA son consideradas como arquitecturas de software.

En cuanto al propósito que persigue, el MDA tiene como principal objetivo el desarrollo del software usando a los modelos como elementos clave para el desarrollo y a la transformación entre modelos como el mecanismo para llevar a cabo el desarrollo.

En cambio la AS es ocupada para crear modelos del sistema (arquitecturas) con el fin de tomar decisiones tempranas antes de realizar el diseño del software, y servir como una guía para el desarrollo, mantenimiento y evolución del sistema.

Por lo tanto estas dos áreas son complementarias y pueden establecer una simbiosis entre ellas. La dos están siendo abordadas intensivamente en la actualidad como lo demuestran las publicaciones sobre el MDA en [5] y sobre la AS en [6].

En este trabajo se muestra cómo aplicar la estrategia de modelado del MDA para generar transformaciones entre vistas arquitectónicas, y cómo también es posible llegar a producir arquitecturas a niveles de abstracción cercanas a las del código.

2.2. Las transformaciones y la arquitectura de software.

Uno de los elementos claves del MDA es la transformación de modelos, con lo cual se logra pasar de un modelo de mayor nivel de abstracción a uno de menor nivel de abstracción, es decir modelar arquitecturas que son independientes de las plataformas llamadas modelos PIM (Platform Indepent Model) y llegar a generar modelos que se adecuen a la plataforma donde se ejecutará la aplicación, llamados modelos PSM (Platform Specific Model), tal como se muestra en a figura 1(a).

En la figura citada se observa también cómo se incorpora la información de una plataforma específica o información adicional (IA). De esta manera a partir de un PIM se pueden generar varios PSM. La tarea de modelado se concentra ahora sólo en el espacio del problema. Cuando se deseen obtener versiones para *x* plataforma se deberá aplicar una transformación incorporando la información de esa plataforma *x*. Es posible también no incorporar ninguna información de ninguna plataforma ni otra adicional, en todo caso la

transformación generará otro modelo PIM, es decir un modelo del mismo nivel de abstracción.

De la misma manera que se logra una transformación de un PIM a un PSM, es posible transformar un modelo de la AS partiendo de un modelo para un sistema que no depende de los detalles específicos de la aplicación (MAS) a un modelo arquitectónico de software generado para una aplicación específica (MASA), como se observa en la figura 1(b), que al realizar la transformación se incorporan los detalles de la aplicación. Vale la pena aclarar que siempre que se diseña una AS ésta es del tipo MAS (antes de diseño del sistema), y el modelo MASA corresponde a una AS de un sistema a nivel de diseño (a un nivel inferior de abstracción). También es posible generar modelos que están a un mismo nivel de abstracción, al no incorporar los detalles de la aplicación la transformación produciría otro MAS, esto es útil cuando se realizan transformaciones de vistas arquitectónicas del mismo nivel.

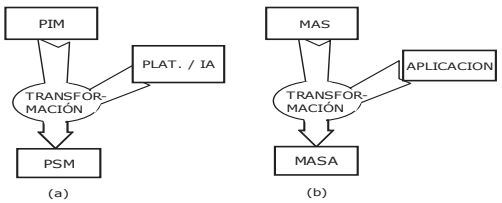


Fig. 1 Transformación de modelos (a) MDA (b) AS

3. La propuesta de transformación entre vistas arquitectónicas de software.

3.1 Los elementos de la transformación.

Para poder crear y realizar transformaciones de modelos MDA cuenta con el soporte de MOF 2.0 y de Query/Views/Transformations [2]. MOF proporciona un marco para la especificación del modelado y la notación para la representación de los modelos, y QVT es el lenguaje para realizar relaciones y hacer transformaciones.

MOF propone 3 niveles de abstracción por arriba de las instancias a modelar: el nivel 0 es el inferior y representa los objetos (instancias) a modelar. Por ejemplo una factura: el nivel 1 es el uso de un modelo para representar a los elementos físicos, como las tablas del modelo relacional que represente los datos de x facturas; el nivel 2 es la especificación del modelo usado en el nivel 1, es decir un meta-modelo; y el nivel 3 es el lenguaje (MOF) que se ocupa para representar meta-modelos (del nivel 2).

La estrategia en la transformación de modelos tiene su base en establecer primero las relaciones entre meta-modelos para después aplicar estas relaciones (o reglas) a nivel de modelos. En una transformación se dispondrá de un modelo origen, el cual mediante las reglas (a nivel de meta-modelo) llegará a producir una o más transformaciones. El modelo resultante es llamado modelo destino.

En la Figura 2 se presenta esta idea aplicada a las vistas arquitectónicas de software. En ella se aprecian los niveles de modelado. El más alto (M3) es donde está un meta-meta-modelo, porque es donde se encuentra definido el modelo MOF, que sirve para expresar a los meta-modelos del siguiente nivel (M2).

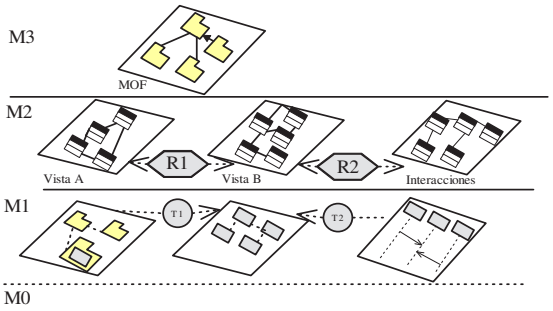


Fig. 2 Transformación de modelos entre vistas arquitectónicas de software.

En este segundo nivel se representan los meta-modelos a relacionar, donde se definen las relaciones de correspondencias. Para el caso de las vistas arquitectónicas se proponen dos tipos de éstas representadas en la figura referida como R1 y R2.

R1 establece las relaciones de correspondencias entre una vista y otra, esto es entre sus meta-modelos (de las vistas A y B). Ésta permitirá llegar a obtener una vista a partir de otra a nivel de modelo, el modelo origen será el *A* y el destino será el *B*.

La otra relación R2, se establece entre el meta-modelo de una vista (B) y el meta-modelo que contiene la información de la *interacción* de los elementos identificados en la fase de requisitos. Esto sirve para que el modelo de la vista que se llegue a obtener (B) se pueda refinar con información adicional (IA) agregándole detalles de dinamismo.

En la parte inferior de esta Figura 2 se indican las transformaciones (T1 y T2) a nivel de modelo, -> indica la dirección en que ésta debe ocurrir.

3.2 Tareas y elementos en una transformación.

Para llevar a cabo las trasformaciones se requiere precisar los elementos de entrada necesarios, y la secuencia a seguir para lograr este propósito. En la Figura 3 se indican éstos.

El primer conjunto de elementos consiste en los meta-modelos antes referidos (los de las vistas A y B, y el de interacciones), éstos se ocupan como entrada para identificar la relaciones entre sus elementos, las cuales después se especifican a través de un lenguaje que permitirá expresar de manera precisa y correcta a estas relaciones. Además éstas deben ser persistentes porque serán requeridas en la transformación de sus modelos tantas veces como se requiera.

La transformación etiquetada con el número 3 ocupa como entrada, además de las relaciones previamente establecidas, al modelo de la vista A, con el cual se obtiene la primera versión del modelo de la vista B, que a su vez será la fuente para la siguiente transformación que también será alimentada con los modelos de iteraciones.

Nótese que tanto el modelo de la vista A como el de las interacciones son extraídos de los requisitos (ésta tarea queda fuera del alcance de este trabajo).

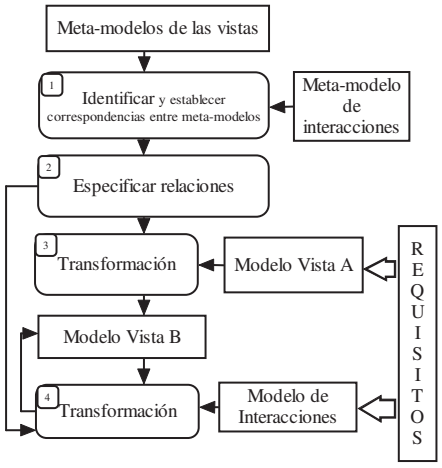


Fig. 3. Elementos requeridos y secuencia de tareas en la transformación de modelos de las vistas.

4. Las vistas arquitectónicas de software y sus meta-modelos.

4.1 Los enfoques de las vistas arquitectónicas.

El concepto de vista tiene muy variadas interpretaciones. El estándar de la IEEE [7], cuya parte se muestra en la Fig. 4, considera a una vista como parte de la arquitectura y la asocia con un punto de vista y con los modelos. Así una vista puede tener muchos, uno o varios modelos para representarla. La situación que se presenta por el lado de los puntos de vista que conforman a una arquitectura es que se abre la posibilidad de tener tantas vistas como puntos de vista se tengan, por lo tanto el problema se traslada a cómo generar sus meta-modelos. No obstante al ser un estándar se debe considerar (aunque sea en términos muy generales) lo que debe tomarse en cuenta en una vista.

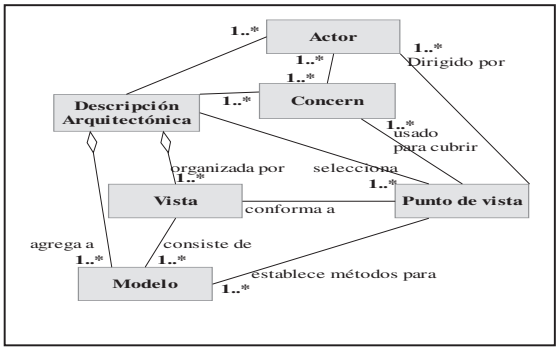


Fig. 4. Meta estructura de la descripción arquitectónica de software, especificado por el estándar 1471 de la IEEE.

Hay otras propuestas más específicas de lo que debe ser una AS. El modelo de 4+1 vistas de [8] considera a cuatro tipos de éstas y el +1 es una vista que relaciona (vincula) a las otras cuatro. En ese enfoque cada vista es una parte del proceso de desarrollo (funcional, de código, de procesos, de despliegue), por tal razón fue adaptada al proceso unificado de desarrollo (RUP), sin embargo estas vistas se tienen que ir construyendo a la vez que se diseña el sistema y la arquitectura pierde la intencionalidad de ser parte de la planeación del diseño. Algo parecido ocurre con la propuesta de [9].

El planteamiento que se hace en [4] y en [10] proponen que las vistas son la representación de las estructuras de software del sistema, lo cual resulta adecuado para diseñarlas antes de detallar al sistema. Así una vista se presenta como una forma de modelar con antelación a un sistema de software.

Por lo tanto este enfoque es el que se seguirá para hacer el modelado de dos vistas, y se seleccionan a la modular y a la de componentes-conectores (C-C) porque representan dos tipos de estructuras que tienen los sistemas de software, la estructura estática y la dinámica. La primera corresponde a la modular, la cual a través de módulos representa las relaciones de los

macro elementos de un sistema. La estructura dinámica es bien representada por la vista de componentes y conectores. En la figura 5 se presenta a estas dos vistas a nivel de modelo, ahí se pueden apreciar sus elementos primarios (módulos y componentes-conectores). Como se indicó en la sección 3.2, para poder llevar a cabo las transformaciones es necesario crear sus meta-modelos.

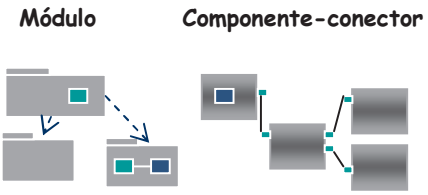
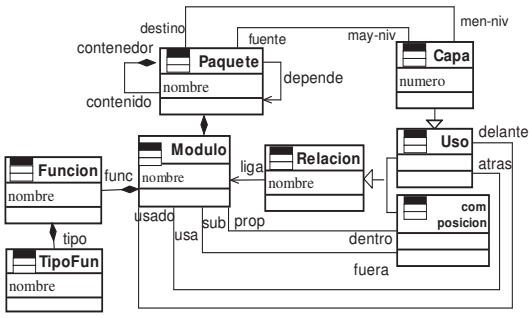


Fig. 5 Modelos de los tipos de vistas: modular y componente conectores

4.2 Meta-modelo para la vista-modular.

El meta-modelo de la vista-modular es mostrado en la figura 6, el cual está conformado por dos elementos primarios que son "Modulo" y "Paquete", el "Modulo" posee una propiedad que es la función que desempeña (Funcion), pudiendo ser más de una, a su vez la función tienen un propiedad de "Tipo", cuyo nombre puede ser: "LOCAL", "SALIDA" o "INTERACCION". Indicando la forma en que la "Funcion" es accedida: "LOCAL" significa que sólo se activa del exterior pero sus efectos son locales; "SALIDA" significa que produce alguna información; y por último "INTERACCION" significa que requiere información de otro "Modulo" y a su vez entrega información.

Se han modelado los estilos arquitectónicos propuestos en [4]: (capa, uso y composición) como tipos de relación que se establece entre los módulos o entre los paquetes y las etiquetas en las ligas de asociación son útiles para establecer las relaciones. Por ejemplo en la relación de "Uso" la etiquetas del lado de "Modulo" indican que un objeto de "Modulo" será el "usado" y otro el que "usa".



Fig,. 6 Meta-modelo de la vista-modular

4.3 Meta-modelo para la vista-C-C.

Este meta-modelo para la vista-C-C es mostrado en la figura 7, ahí se indica que sus elementos primarios son componente y conector, los cuales a su vez son sub-tipos del Tipo-comp, este último es un componente genérico que incluye como atributo el nombre (heredado a sus clases Componente y Conector),

Una clase "Componente" a su vez incluye como atributos a "Servicio" y "Puerto", a su vez un objeto de servicio está asociado con un objeto de Puerto, pues es por donde se establecerá la comunicación de dicho servicio al exterior del componente.

Por el otro lado, el Conector tiene a su vez como atributo al Rol, y debido a que un objeto de la clase Conector vincula a un módulo con otro, se establece una relación binaria de asociación entre Conector y Componente. Por esa misma razón hay una relación de asociación entre el Rol y el Puerto, pues a través de ellos se establece el vínculo de comunicación.

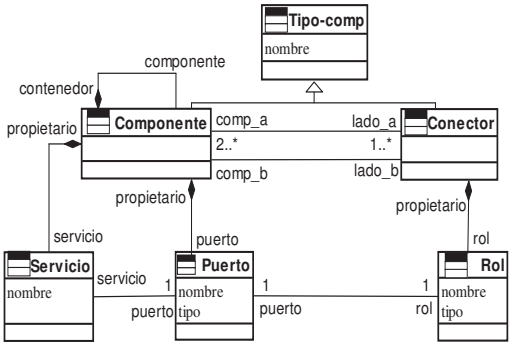


Fig 7. Meta-modelo de la vista-C-C

5. Estableciendo relaciones para la transformación.

5.1 El lenguaje.

Una vez que se tiene a los meta-modelos involucrados en la transformación, el siguiente paso es proceder a establecer las relaciones entre ellos, es decir, especificar las reglas que se seguirán para llevar a cabo una transformación, en este caso la transformación entre las vistas arquitectónicas.

Esto se hace mediante alguno de los lenguajes de transformación de modelos analizados en [11]. Los lenguajes más ocupados son el ATL y el QVT [5], el primero es soportado directamente por el entorno de trabajo de ECLIPSE [12], y el segundo es una propuesta dada por OMG, aquí se usará este último lenguaje ya que tiene como una de sus ventajas el poder expresar las reglas de relación en código y en notación gráfica lo cual permite ilustrar mejor la propuesta.

Algunas de sus características relevantes de éste son: que es un lenguaje declarativo pero también usa sentencias imperativas e incorpora también a cláusulas del OCL (Object Constrain Lenguaje).

Una relación se establece usando la palabra clave "relation" y un nombre para identificarla. Hay dos tipos de relaciones: las que se ejecutan sin ninguna llamada explícita (anteponiendo "top" a "relation") y las que se invocan explícitamente, la cual se incluye en un predicado "where". Cada relación involucra a dos dominios (domain), el origen y el destino. En cada dominio se llevan acciones de sólo verificación o de "enforce", lo primero no crea objetos, en cambio lo segundo se usa para crear elementos, esto es generar el modelo destino. La referencia completa de este lenguaje se encuentra en [6].

5.2 Estableciendo relaciones.

Para establecer las relaciones (reglas de transformación) mediante el QVT se ocupa un proceso heurístico: lo primero que se hace es identificar aquéllas relaciones que deben ser ejecutadas en forma automática, es decir, sin que sean llamadas explícitamente por otra relaciones, luego se establecen dependencias entre ellas, posteriormente se eligen aquéllas que no dependan de otras, las cuales serán relaciones de tipo "top" y las restantes se asocian con quienes las van a invocar (o llamar). Esta invocación se hace desde dentro del predicado "where", lo cual permite establecer una liga o secuencia de ejecución. Dentro de ese predicado también se pueden invocar funciones y usar con cláusulas del OCL.

Es importante hacer notar que no todos los elementos de un meta-modelo origen (o fuente) se relacionan o establecen alguna regla con elementos del meta-modelo destino, pues no siempre son modelos cien por ciento compatibles.

En los modelos aquí considerados, esta situación de "no correspondencia" se presenta en la clase relación "Capa" del meta-modelo de la vista modular, la cual no establece ninguna relación con ninguna clase del meta-modelo C-C. Porque esa propiedad de Capa sólo se presenta en la vista modular [4]. También puede darse el caso de que el modelo destino no se relacione con alguna clase del meta-modelo fuente. Sin embargo esto queda fuera del alcance de este trabajo. Siguiendo este proceso se identificaron las relaciones descritas en la tabla 1. donde hay tres tipos de relación que se activan automáticamente y sólo una (funcionAservicio) que es llamada por alguna otra.

Tabla 1. Relaciones identificadas en los meta-modelos: modular y C-C

Relacion	Ti-po	Clases que participan en la relación	
		Modular	C-C
moduloAcomponente	top	Modulo	Componente
funcionAservicio	-	Modulo, Funcion, Tipo	Componente Servicio Puerto
rUsoModAconector	top	Uso, Modulo, Servicio	Conector, Rol, Componente, Puerto, Servicio
rComposisicionModAcomp	top	Composición Modulo	Componente

Relación: moduloAcomponente

Este vínculo "mapea" cada "Modulo" con un "Componente". Note en el código mostrado en la Figura 7 que el objeto de dominio de Modulo sólo será verificado, en cambio el de dominio destino tiene un tipo "enforce" lo cual hace que cree un objeto de la clase "Componente" asociado con la clase "Modulo".

Por otro lado dentro de "where" existe una llamada a la relación "funcionAservicio" relacionando a un objeto de "Modulo" con uno de "Componente" en la invocación de la función (como paso de parámetros). Su diagrama se muestra en la Figura 8 y su código correspondiente en la Figura 9.

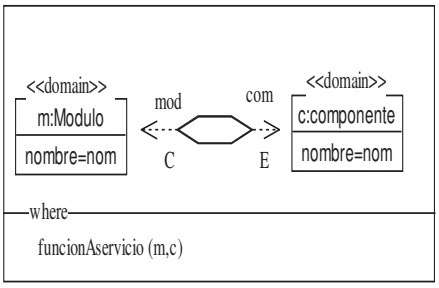


Fig. 8. Relación ModuloAcomponente

```
top relation moduloAComponente
{ nom: String;
  checkonly domain mod m: Modulo {
                                    nombre = nom};
   enforce   domain  com c: Componente{
                                    nombre = nom};
where{ funcionAservicio(m,c)}
}}
```

Fig 9 Relación: ModuloAcomponente (código)

Relación: funcionAservicio.

En este caso esta relación implica que una "Funcion" del meta-modelo "Modulo" generará un "Servicio" de "Componente", al hacerlo también el "Tipo" de la "Funcion" generará un elemento de "Puerto". El "Puerto" tendrá una correspondencia con el nombre del "Tipo" de la siguiente manera: "LOCAL" → "ENTRADA", "SALIDA"→ "SALIDA", "INTERACCION" → "ENT_SAL". Para este propósito se emplea una función de OCL que realizará dicha correspondencia. Note en la cláusula where en la Fig. 10 y 11, cómo la variable tipoPto que será el nombre del puerto, se obtiene llamando a la función tipoPuerto, dicha función tiene código en OCL.

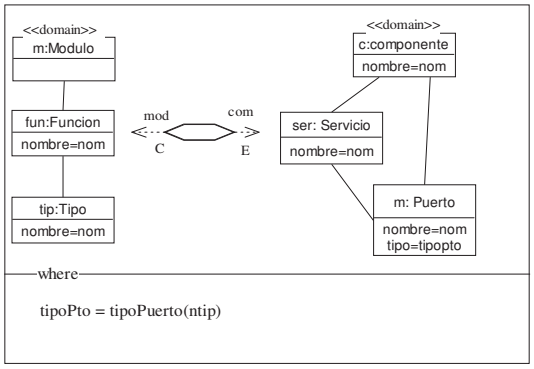


Fig 10. Relación funcionAservicio

Por el lado los meta-modelos fuente se emplean tres clases en esta relación, las cuales sólo serán verificadas al momento de ejecutar las relaciones.

En cambio, las clases los meta-modelos destino serán creadas, aunque solamente las clases "Servicio" y "Puerto", donde éstas establecen una relación con la clase del "Componente" de donde se llamó a este servicio, esto es mostrado en su diagrama de la figura 8 y su código en la figura 9.

```
relation funcionAservicio
{
 nom,ntip,nser,tipoPto:Strings;
 checkonly domain mod m:Modulo {
    funcion = fun:Funcion {nombre=nom,
                tipo = tip:Tipo {nombre=ntip}}}
    enforce domain c:Componente{
                servicio = ser:Servicio {nombre=nser},
                puerto     = pue:Puerto {nombre=nser +
'pto',
                tipo     = tipoPto}}
    where {tipoPto = tipoPuerto(ntip)
               }
}
 function tipoPuerto(tipoFun:String):String;
 { if (tipoFun = 'LOCAL') then 'ENTRADA'
     else if (tipoFun = 'RESULTADO') then 'SALIDA'
               else                  'ENT_SAL'
                endif
  endif;
 }
}
```

Fig. 11. Relación funcionAservicio (código)

Relación: rUsoModAconector

Aquí se transforma la relación Uso de la meta-clase Modular a un vínculo entre un conector y dos componentes, en la meta-clase C-C, como se ilustra gráficamente en la Fig. 12, la creación de objetos para esta relación es numerosa, puesto que se genera una relación de dos componentes con un conector con sus respectivos puertos. En la Fig 13 se muestra parte del código para destacar cómo dentro del where se llaman a las relaciones que crean la relaciones entre módulo componente y funcionAservicio, las primeras dos crean los componentes a conectar y la segunda sus servicios correspondientes. Además se aprecia cómo las variables de la clase Componente serán usadas como parámetros en la relación contenida que dentro de la cláusula *where*.

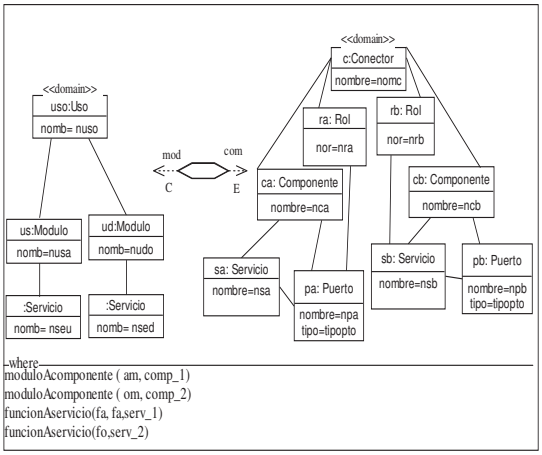


Fig. 12 Relación rUsoModAconector

```
top relation rusoModAConector
  {
     ser_1, ser_2  :Servicio;
     comp_1, comp2:Componente;
     nma,nmo,ncon,nfa,nfo,tfa,tfo : String.
     checkonly domain reluso ru:Uso {nombre=nuso,
     usa = am:Modulo {nombre,nma, fun =
fa:Funcion {nombre=nfa, tipo=tfa}}
              usado = om:Modulo{nombre=nmo,fun =
fo:Funcion {nombre=nfo, tipo=tfo}}
    }.....
.....
……
    where { moduloAComponente (am,comp_1);
              moduloAComponente (om,comp_2);
              funcionAservicio (fa,serv_1);
              funcionAservicio (fb,serv_2);
             }
```

Fig. 13 Relación rUsoModAconector (código)

Relación rComposicionModAcomp.

En este caso la composición de Modulos generará también una composición de componentes, note que al momento de crearse un componente dentro de él se crea el subordinado, de esta manera se establece también una composición de componentes. Ver figuras 12 y 13.

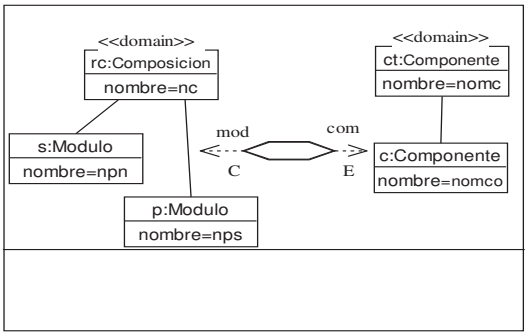


Fig 14. Relación rComposicionModAcomp

```
top relation rcomposicionModAcomp
  { nc, nct, nco, nms, ncomp : String;
    checkonly domain rcomp rc:Composicion
{nombre=nc,
                  prop = mp:Modulo {nombre= nmp}
                  sub = ms:Modulo {nombre=nms}
    }
  enforce domain cc: Componente{nombre =  nct
contenedor  = cc;
   componente  =  c : Componente{ nombre = nco
  }
}
```

Fig 15. Relación rComposicionModAcomp (código)

6. Aplicando la transformación entre vistas a nivel de modelo.

Se plantea como caso de estudio el diseño de una arquitectura de software para un sistema de reservación de boletos (tickets o billetes) para el transporte de pasajeros (avión, autobús, tren). Los requisitos ya fueron analizados y se ocupó el método propuesto en [14], el cual sigue un proceso iterativo y descendente para diseñar una vista. En este caso se generó sólo el modelo de la vista-componente, cuya parte de ésta se muestra en la Fig. 16 ocupando la notación de UML 2.0.

Existen cuatro módulos vinculados entre sí por una relación de uso. *QuerForm*, tiene como responsabilidad generar las interfaces para proporcionar los datos que se van a consultar sobre las distintas líneas de transporte, ahí se debe proporcionar la información necesaria (origen, destino, número de personas, fechas de salida y regreso). Además se mostrarán los distintos itinerarios disponibles que serán generados por el módulo *GeneItin*

El Módulo *QuerForm* será usado por el módulo *Parchase* que tendrá como responsabilidad llevar a cabo las funciones relacionadas con el pago de los boletos. Este último módulo usará al de reservación donde se supone se asigna el o los asientos correspondientes al tipo de medio de transporte.

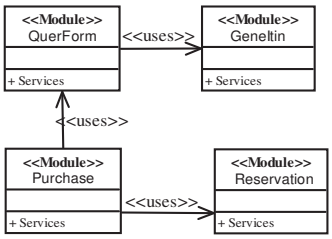


Fig 16. Modelo de una vista modular.

Ahora se aplican las reglas que se han establecido a nivel de meta-modelo, para lo cual se ocupó la propuesta de MOMENT [15] que usa como marco de trabajo a ECLIPSE, tanto para la representación de los meta-modelos y de los modelos, como para la ejecución de la transformación

Las reglas o tipo de relaciones que fueron usadas en este caso son las de ModuloAcomponente y la de "rUsoModAConector" puesto que sólo hay relaciones de <<uso>> en el modelo de la vista modular, además por supuesto de los módulos. La aplicación de estas reglas llega a producir la primera versión del modelo de una vista arquitectónica de C-C, tal como el que se muestra en la figura 17, donde Session y Access son de clase "Conector".

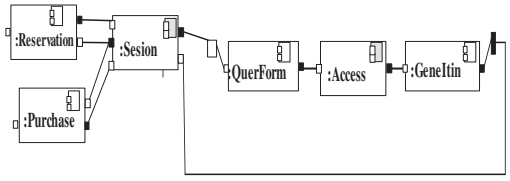


Fig 17. Modelo generado de una vista C-C.

Esta primera versión del modelo de la vista C-C puede ser refinada al aplicarle una transformación usando el modelo de escenarios. En la figura 17 se muestra una iteración entre QuerForm y GeneItin, donde se percibe cómo el elemento GenItin es descompuesto en dos: Itinerari y Routes, de tal manera que si se aplica de nuevo la transformación, el modelo de la vista reflejará este refinamiento tal como se muestra en la figura 19.

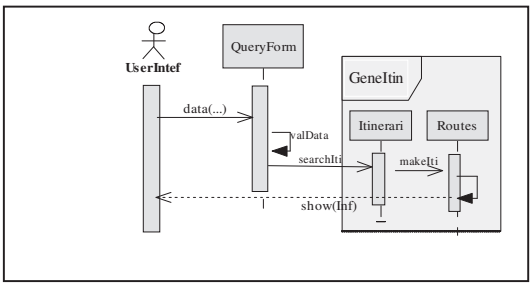


Fig. 18. Modelo de Escenarios usando diagrama de secuencias en UML 2.0

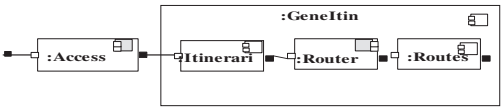


Fig. 19. Refinamiento del modelo de la vista C-C.

7. Conclusiones

Al haber establecido un vínculo entre el MDA y el AS se muestra cómo es posible diseñar arquitecturas de software con más rapidez y con re-uso mediante orientación por modelo. Con la creación de los meta-modelos arquitectónicos y de las reglas de transformación se ha mostrado la forma de modelar y diseñar arquitecturas de software aplicable inclusive para otros tipos de vistas. El generar más rápidamente la vista modular y la de C-C a nivel de modelo para un sistema, hará posible que más sistemas cuenten con una arquitectura más integral (de dos vistas). Tanto los meta-modelos como las relaciones realizadas son extensibles, esto es, pueden incorporarse más relaciones enriqueciendo las transformaciones. La heurística planteada para llevar a cabo el proceso de transformación, aunque informal**,** servirá como un primer paso para crear un método de diseño que permita hacer en un futuro cercano todo el proceso automáticamente, aunque es claro que la identificación de relaciones sigue siendo uno de los más grandes retos en el proceso de modelado.

Se reconoce que el camino hacia una completa automatización con el modelado aún es largo pero se ha dado un primer paso en relacionar estas dos novedosas áreas de la ingeniería de software

.

Referencias

[1] Arlow J., Neustadt I, "Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML, Addison Wesley Professional, ISBN: 978-0-321-11230-9, 2003.

[2] OMG, Object Management Group: Model Driven Architecture (MDA). 2004. http://www.omg.org/cgi-bin/doc?formal/03-06-01

[3] Shaw and Garlan, Software Architecture: perspectives on an emerging discipline, Prentice Hall ISBN 0-13-182957-2, 1996

[4] Bass L., Clements P., Kazman R., Software architecture in practice, Addison-Wesley, 2ª Ed. , ISBN 0-321-14595-9, 2003.

[5] IEEE Computer, Vol. 38, N. 2. Feb. 2006.

[6] IEEE Software, Volume: 23 Mar-Apr 2006.

[7] IEEE Product No. SH94869-TBR, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000

[8] Kruchten Philippe. The 4+1 View Model of Architecture, Paper published in IEEE Software Vol.12, No.6, pp. 42-50 Nov. 1995.

[9] .Hofmeister, C., Robert Nord, and Dilip Soni. Applied Software Architecture. Boston, MA: Addison Wesley, 2000

[10] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R. Stafford J. Documenting Software Architecture, Views and Beyond, Addison-Wesley, 2002.

[11] Krzysztof Czarnecki and Simon Helsen, Classification of Model Transformation Approaches OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[12] ECLIPSE: http://www.eclipse.org/

[13] OMG 2nd Revised Submission: MOF 2.0 Query/Views/Transformations. http://www.omg.org/cgi-bin/doc?ad/05-03-02

[14] Limon C. R., Ramos S.I., Torres J. J. Designing Aspectual Architecture Views in Aspect-Oriented Software Development , Computational Science and Its Applications, LNCS 3983, pp, 726-735, 2006.

[15] El proyecto MOMENT: http://moment.dsic.upv.es/

Marco de Trabajo basado en MDA para la Medición Genérica del Software

Beatriz Mora, Félix García,
Francisco Ruiz, Mario Piattini
Dep. de Tecnología y Sistemas de Información
Escuela Superior de Informática de Ciudad Real
Universidad de Castilla-La Mancha
{Beatriz.Mora | Felix.Garcia | Francisco.RuizG |
Mario.Piattini}@uclm.es

Artur Boronat, Abel Gómez,
José Á. Carsí, Isidro Ramos
Dep. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{aboronat | agomez | pcarsi | iramos}@dsic.upv.es

Resumen

Actualmente, con el objetivo de obtener productos software de calidad es necesario llevar a cabo una buena gestión de los procesos software donde la medición de los procesos se convierte en un factor fundamental. Debido a la gran variedad de entidades candidatas para la medición, se considera necesario un marco consistente para integrar la medición de los distintos tipos de entidades. En este trabajo se presenta la propuesta de un entorno genérico para la medición de cualquier entidad software a partir de los metamodelos que las representan. Partiendo de un metamodelo común de medición, y mediante transformaciones QVT, se puede llevar a cabo la medición de un modelo de dominio cualquiera. En el trabajo se explica como se ha llevado a cabo la propuesta: por un lado, se ha trabajado con la herramienta MOMENT, que proporciona el soporte necesario para la gestión automática de modelos de acuerdo a MDE y a la arquitectura MDA, por otro lado, se ha adaptado FMESP a MDA. FMESP es un marco de trabajo para la integración del modelado y de la medición de procesos software, que sirve de base conceptual y tecnológica para su mejora. Además, se muestran las etapas a seguir para conseguir la medición genérica basada en MDA, y un caso de ejemplo en el dominio de bases de datos relacionales.

Palabras clave: Medición, MDA, MOMENT.

1. Introducción

La actual necesidad de la industria del software por mejorar su competitividad fuerza a la búsqueda de la mejora continua de sus procesos. Para conseguirlo, es necesaria una gestión exitosa de dichos procesos [10], lo que implica su correcta definición, ejecución, medición, control y mejora. Entre estas fases del ciclo de vida de los procesos destaca la medición, que ayuda a controlar los errores y carencias dentro del desarrollo y mantenimiento del software facilitando la toma de decisiones. De hecho, la medición se ha convertido en un aspecto fundamental de la Ingeniería del Software [8].

Los procesos software constituyen la base a partir de la cual se realiza el trabajo dentro de una organización software. Dichos procesos se aplican en la práctica en forma de proyectos. Como resultado de la ejecución de proyectos concretos se obtienen productos. Por lo tanto, para facilitar y promover la mejora continua de sus procesos, las empresas requieren llevar a cabo la medición del software de manera efectiva y consistente. Esto implica la necesidad de una disciplina para la medición y análisis de datos [5] y la definición, recopilación y análisis de medidas sobre el propio proceso, los proyectos y los productos software.

La gran variedad de tipos de entidades y atributos que son candidatos para la medición motiva la necesidad de disponer de modelos de medición homogéneos, que puedan gestionarse por las empresas de la misma forma, independientemente de cual sea la entidad a

Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Marco de Trabajo basado en MDA para la Medición Genérica del Software. Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). 2007.

medir. Esto implica la necesidad de una referencia consistente y adecuada para la definición de sus modelos de medición del software así como el soporte tecnológico necesario para integrar la medición de los diferentes tipos de entidades.

Con el fin de satisfacer las necesidades expuestas, es muy útil considerar el paradigma MDE (*Model-Driven Engineering*) [3], de especial importancia en la actualidad. La filosofía de este enfoque consiste en que el desarrollo de software es dirigido por los modelos, siendo éstos los principales artefactos de los procesos de Ingeniería del Software. La arquitectura MDA (*Model Driven Architecture*) [22] y sus estándares relacionados (MOF [20], QVT [24], OCL [23] y XMI [21]) proporcionan la base conceptual y tecnológica necesaria para llevar a cabo las ideas de dicho paradigma. De acuerdo al estándar QVT, el proceso de desarrollo del software se puede considerar como una serie de transformaciones de modelos, a partir de un nivel de abstracción alto hasta un nivel más específico. En el nivel más abstracto se pueden ver los requisitos, y en el nivel más específico estaría el código que implementa la aplicación. En conclusión, la principal importancia a la formalización de los modelos y las transformaciones entre ellos es que facilita la automatización del proceso de desarrollo de software.

El campo de la medición software puede beneficiarse de la nueva filosofía MDE, proporcionando la integración y el soporte necesario a la automatización de la medición de las diversas entidades del proceso software. Esto implica: la definición de modelos de medición de manera homogénea y consistente a partir de un metamodelo adecuado; la definición de medidas genéricas que puedan aplicarse a cualquier modelo; y el soporte necesario para calcular de forma automática las medidas definidas, almacenar de forma homogénea los resultados y facilitar la toma de decisiones mediante el análisis de los mismos. Por tanto, se habla de homogeneidad en el sentido de que existe una manera general de definir y medir cualquier tipo de entidad o artefacto software. Estos aspectos constituyen el principal interés del presente trabajo, en el que se ilustra la aplicación de los principios, normas y herramientas de MDA al campo de la medición del software. El objetivo es desarrollar un entorno genérico para la definición de modelos de medición *bien formados respecto a*[1] un metamodelo común, y para la medición de cualquier entidad software en base a un metamodelo de dominio. Para llevar a cabo la propuesta, se ha trabajado con el entorno MOMENT [15], que proporciona el soporte necesario para la gestión automática de modelos de acuerdo a MDE y MDA.

El resto de este artículo está organizado de la siguiente forma: En el siguiente apartado se muestran los trabajos relacionados. Después se presenta la arquitectura conceptual y el metamodelo de medición del software. En el cuarto apartado se describe la adaptación realizada del marco FMESP[12] a MDA, para lo cual se indican el entorno tecnológico empleado, la adaptación propiamente dicha de la parte de medición de FMESP a MDA, y se muestra el procedimiento de uso de dicha adaptación. A continuación se presenta un caso de ejemplo en el dominio de bases de datos relacionales. Por último, se presentan algunas conclusiones y trabajos futuros.

2. Trabajos Relacionados

Muchas publicaciones mencionan las herramientas que dan soporte y automatización a la medición como importantes factores de éxito en los esfuerzos de la medición del software [18], proporcionando entornos de trabajo y aproximaciones generales [17], o dando arquitecturas de soluciones más específicas [16].

Como consecuencia, en la bibliografía existe una gran variedad de herramientas que dan soporte a la creación, control y análisis de métricas software. Una lista extensa puede consultarse en [6]. Por su parte, Auer examina en [2] distintas herramientas de medición del software en entornos heterogéneos, como son: MetricFlame, MetricCenter, Estimate Profesional, CostXPert y ProjectConsole.

También se pueden encontrar en la bibliografía algunas propuestas en las que se

[1] A lo largo del trabajo se habla de la relación existente entre modelo y metamodelo. Las expresiones que se utilizan en el trabajo son: un modelo "*está bien formado respecto a"* o "*está definido mediante"* un metamodelo. Otras expresiones como un modelo "*es conforme a"* o "*es instancia de"* un metamodelo no se utilizan porque se consideran menos precisas.

aborda la medición de software más integrada y menos específica que en las herramientas anteriores. En [26] se propone MMR, que es una herramienta basada en el modelo CMMI para la evaluación de procesos software. Otras herramientas similares pueden consultarse en [19], [14], y [28]. Sin embargo, muchas de estas herramientas están restringidas a dominios o modelos de evaluación de calidad específicos, lo que reduce su generalidad y alcance.

Para abordar la medición genérica, en [11-13] y como parte del entorno FMESP (*Framework for the Modeling and Evaluation of Software Processes*), se propone un marco de trabajo basado en la arquitectura conceptual de metadatos del estándar MOF. Esta propuesta está soportada por la herramienta GenMetric, que permite definir modelos de medición software y calcular las medidas definidas en dichos modelos sobre cualquier entidad software. Sin embargo, se ha considerado interesante adaptar FMESP al paradigma MDE para explotar los beneficios que puede aportar a la medición del software. Esto implica: i) el refinamiento del metamodelo de medición del software definido en la propuesta anterior; y ii) la adaptación y extensión de GenMETRIC hacia un entorno que permita de una forma intuitiva tanto la definición de modelos de medición software que puedan aplicarse sobre cualquier tipo de entidad software como el cálculo de las medidas definidas, todo ello en el contexto de modelos y transformaciones de modelos de la arquitectura MDA.

3. Medición de Software basada en MDA: Arquitectura Conceptual

Tal y como se ha comentado en el apartado anterior, ante la necesidad de disponer de un entorno genérico y homogéneo para la medición del software en trabajos previos [11-13] se propone una arquitectura conceptual y una herramienta para la integración de la medición del software. A continuación, se describen brevemente las principales características de dicha propuesta. Una descripción más detallada puede consultarse en [13].

3.1. Arquitectura Conceptual

La propuesta presentada sirve para dar el soporte necesario para gestionar y representar el conocimiento referente a la medición de cualquier entidad software de una forma integrada y consistente. Para ello, los distintos elementos de la arquitectura se organizan de acuerdo a los siguientes niveles de abstracción, siguiendo la línea del estándar MOF.

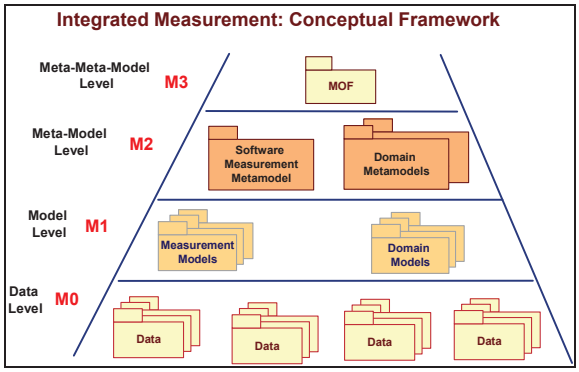


Figura 1. Marco de trabajo conceptual para gestionar la medición del Software

Como se observa en la Figura 1, la arquitectura se ha organizado en los siguientes niveles de metadatos:

- Nivel de Meta-Metamodelo (M3): en el que se encuentra un lenguaje abstracto para definir metamodelos. Este lenguaje es MOF.
- Nivel de Metamodelo (M2): en este nivel, de acuerdo al marco de trabajo se incluyen: el metamodelo de medición, que define los modelos de medición específicos (descrito en el apartado 3.2) y los metamodelos de dominio, que representan a los tipos de entidades candidatas para la medición (por ejemplo, el metamodelo UML para modelos de sistemas OO).
- Nivel de Modelo (M1): en este nivel se incluyen los modelos específicos definidos mediante los metamodelos de M2, pueden ser de dos tipos: los modelos de Medición, definidos mediante el metamodelo de medición; y los modelos de dominio, definidos mediante los correspondientes metamodelos de dominio.

3.2. Metamodelo de Medición del Software

Ante la diversidad terminológica existente en el campo de la medición software, como paso previo a la creación de la primera versión del metamodelo de Medición fue necesario desarrollar una ontología [11]. Dicha ontología permitió establecer y aclarar los elementos involucrados mediante la identificación de todos los conceptos, las definiciones precisas de todos los términos, y la aclaración de las relaciones entre ellos. Basándose en los conceptos y relaciones de esta ontología, se construyó el metamodelo de medición. La Figura 2 muestra en un diagrama de clases UML los principales constructores (o elementos) del Metamodelo de Medición del Software, que constituyen el núcleo de dicho metamodelo [9].

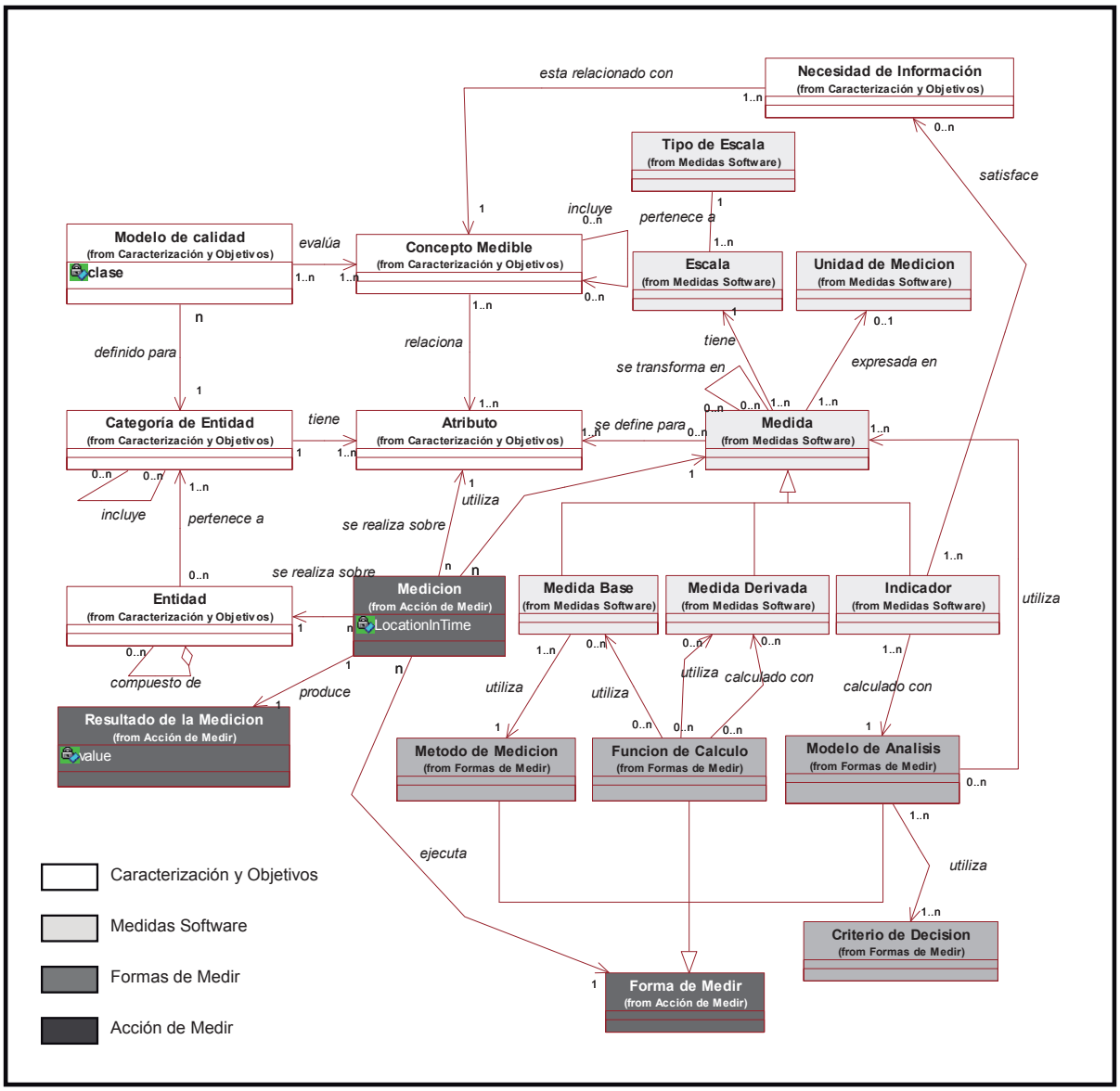


Figura 2. Metamodelo de Medición del Software

El metamodelo de Medición del Software está organizado en 4 paquetes principales (ver Figura 2) que se describen brevemente (para más detalle véase [9, 11]):

- **Caracterización y Objetivos de la medición del Software:** incluye los constructores necesarios para la definición de una vista fundamental del proceso de medición, basada en establecer los objetivos y en identificar y caracterizar los elementos necesarios para cumplir con dichos objetivos.
- **Medidas Software**: incluye los elementos necesarios para definir las características generales de las medidas.
- **Formas de medir**: incluye los elementos de modelado necesarios para la representación de la formas de medir de los diferentes tipos de medidas.
- **Acción de Medir**: incluye los constructores necesarios que permiten representar la forma de llevar a cabo el proceso de medición.

4. Adaptación de FMESP-Measurement a MDA

En este apartado se explican los detalles de la manera en que se ha realizado la adaptación del entorno FMESP a MDA.

4.1. Entorno Tecnológico

La herramienta utilizada en este trabajo ha sido el entorno de gestión de modelos MOMENT. MOMENT [4] es una herramienta que permite definir modelos como especificaciones algebraicas de una teoría expresada en lógica ecuacional condicional. Está basada en una aproximación híbrida entre una herramienta formal (Maude [1]) y un entorno industrial de modelado (Eclipse Modelling Framework, EMF [7]).

Desde un punto de vista funcional, MOMENT tiene dos componentes: ejecución de consultas OCL (MOMENT-OCL) y transformaciones QVT (MOMENT-QVT).

MOMENT-OCL [5] permite ejecutar consultas e invariantes OCL sobre un modelo definido mediante su metamodelo y almacenado en xmi. Incluye un editor (*OCLEditor*) que facilita la codificación de invariantes y consultas OCL que se van ejecutar sobre el modelo.

En este trabajo se ha empleado esta funcionalidad para comprobar y validar las consultas OCL utilizadas en las transformaciones QVT. Los resultados de la consultas OCL se muestran por pantalla (véase Figura 3).

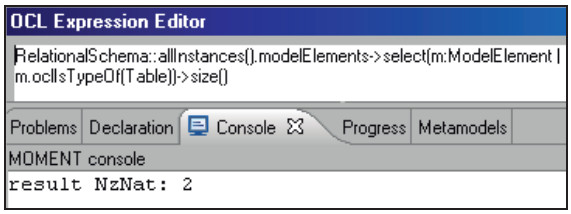


Figura 3. Ejemplo de ejecución de consulta OCL con MOMENT

Por otro lado, MOMENT-QVT [27] permite usar el lenguaje QVT Relations [24] para la definición de relaciones de equivalencia y transformaciones. El motor de transformaciones está integrado en esta herramienta, y permite obtener un modelo de salida a partir de uno o más modelos de entrada, siempre y cuando todos los modelos tengan su correspondiente metamodelo.

Tal y como se explica en el apartado 4.2, las transformaciones QVT son un elemento fundamental de esta propuesta.

Para ejecutar en MOMENT una transformación QVT es necesaria, bien la especificación textual de la transformación (codificada mediante el editor *Textual QVT Editor* y almacenada con extensión .qvtext), o bien, su equivalente modelo QVT Relation (almacenado con extensión .qvt). A continuación se muestran dos representaciones de ejemplo: una especificación textual (Figura 4), y su equivalente modelo QVT Relation (Figura 5).

Un modelo QVT Relation está definido mediante el metamodelo QVT, y puede obtenerse mediante un parseo de la especificación textual. También puede obtenerse mediante una transformación QVT.

```
top relation PackageToSchema
{
    packageName: String;

    checkonly domain ecoreDomain p:EPackage {
        name=packageName
    };

    enforce domain rdbmsDomain s:Schema {
        name=packageName
    };
}

top relation ClassToTable
{
    className: String;

    checkonly domain ecoreDomain c: EClass {
```

Figura 4. Parte de la Especificación textual de una transformación UML2RDBMS

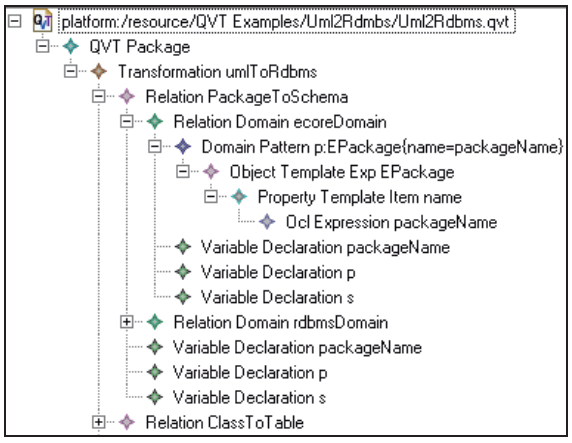


Figura 5. Modelo QVT Relation de una transformación UML2RDBMS

En esta propuesta se ha optado por trabajar con el modelo QVT Relation en lugar de la especificación textual.

Con respecto a la estructura de la plataforma EMF, es de interés resaltar que sigue la cultura de metamodelado presentada en el estándar MOF [20] con la diferencia de que en el nivel M3 se utiliza Ecore[2]. Puesto que Ecore está basado en MOF, la arquitectura de EMF es válida para la propuesta aquí presentada (véase Figura 1).

[2] Ecore es un lenguaje común basado en EMOF, que es parte de la especificación MOF 2.0 usado para la definición de metamodelos

4.2. Adaptación de FMESP-Measurement

En la Figura 6 se muestran los elementos clave para la adaptación de FMESP-Measurement, encuadrados en los niveles correspondientes de la arquitectura MOF. Como se puede observar, el metamodelo QVT (nivel M2) y el Modelo QVT Relation (M1) son los añadidos con respecto a la Figura 1, es decir, para la adaptación de FMESP-Measurement a MDA.

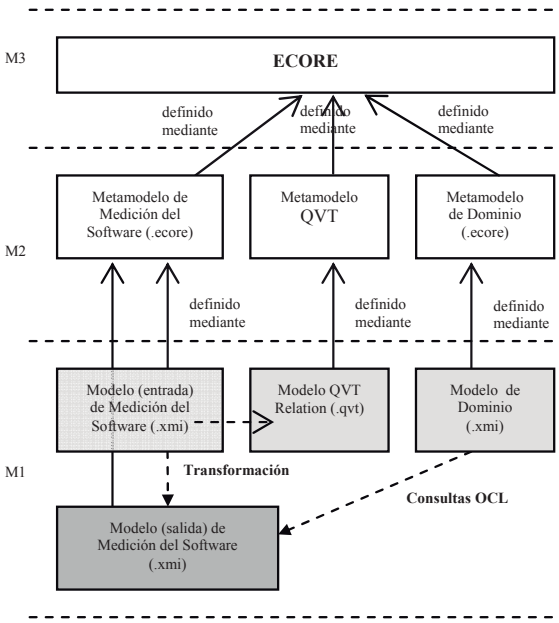


Figura 6. Elementos de la Adaptación de FMESP-Measurement a MDA

Con FMESP-Measurement, el resultado de la medición se obtiene mediante una transformación QVT, donde los modelos de entrada, son dos: un modelo de Medición del software y un modelo de dominio; y el modelo de salida es el modelo de Medición del software con los resultados obtenidos. Esta transformación se obtiene a partir del modelo QVT Relation (véase Figura 6).

A continuación se explican brevemente alguno de los elementos de la figura anterior:

- Metamodelo de Medición del Software (presentado en el apartado 3.2): metamodelo integrado en el marco de trabajo.
- Modelo QVT Relation: este modelo se obtiene mediante una transformación del modelo de de Medición. Contienen la información necesaria para llevar a cabo la transformación QVT de la propuesta FMESP-Measurement.

- Modelo (salida) de Medición del software: es el modelo resultante de la transformación definida en el modelo QVT Relation.

4.3. Procedimiento de Uso

A continuación se explican las etapas que debe llevar a cabo el usuario para poder medir usando la herramienta propuesta:

1. **Incorporación del metamodelo de dominio:** la medición se va a realizar sobre un dominio en concreto. Este dominio ha de ser definido mediante su correspondiente metamodelo (situado en el nivel M2 y bien formado respecto del meta-metamodelo Ecore). Ejemplos de dominios son: esquemas relacionales, diagramas de clases UML, diagramas de flujos de datos, código fuente en Java, manuales de usuario, etc.
2. **Creación del modelo de medición:** en base al metamodelo de medición integrado de base en el framework FMESP-Measurement, se crea el correspondiente modelo de medición, sobre el que se va a realizar la medición. Al ser un modelo de entrada, en este momento el modelo de medición no incluye la parte de los resultados, es decir, el paquete *Acción de Medir*.
3. **Creación del modelo de dominio:** a partir del metamodelo de dominio (incorporado en la etapa 1) se define el modelo de dominio, que representa "*lo que se va a medir*", es decir, el modelo sobre el cuál se va a aplicar la medición.
4. **Ejecución de la medición:** la ejecución de la medición se realiza mediante una transformación QVT, en la que partiendo de los dos modelos de entrada (modelo de medición y modelo de dominio) creados en las etapas 2 y 3 respectivamente, se obtiene un modelo de salida que es el modelo de medición pero ampliado con los resultados de la medición (paquete *Acción de Medir*). Estos resultados se calculan mediante consultas OCL realizadas sobre el modelo de dominio (véase Figura 7).

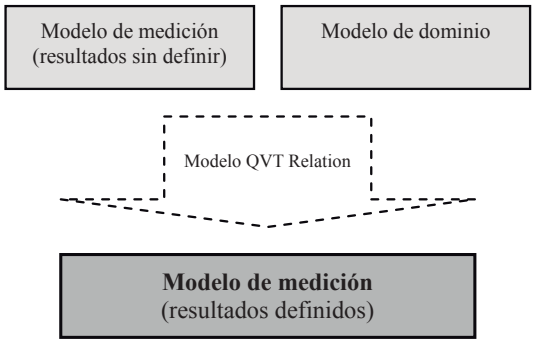


Figura 7. Ejecución de la medición

5. Caso de Ejemplo: Medición en el Dominio de Bases de Datos Relacionales

Para ilustrar la aplicación de la propuesta, se considera el ejemplo de la medición de esquemas relacionales. Por simplicidad se consideran los elementos mostrados en la Figura 8.

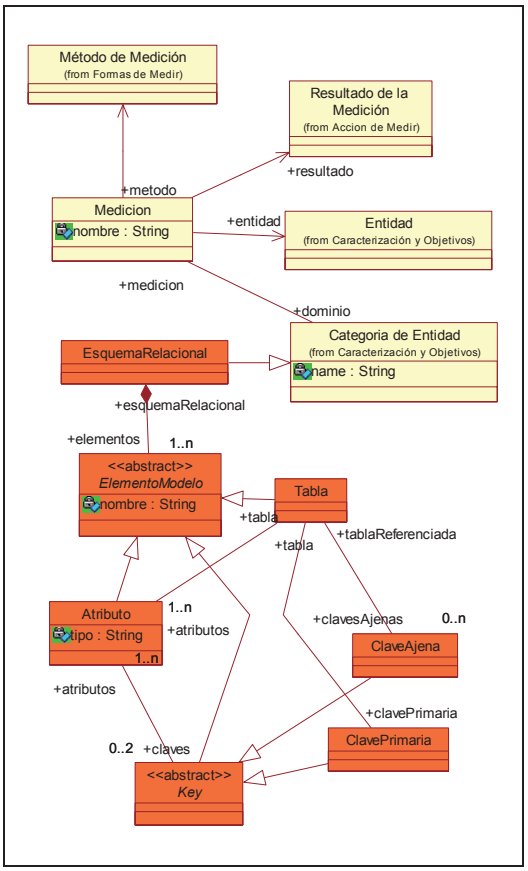


Figura 8. Metamodelo de Medición y de Dominio

Para la realización del caso de ejemplo, los elementos del metamodelo de medición más significativos son: *método de medición*, *entidad* a la que se va a aplicar el método de medición y *resultado* de la medición.

Por otro lado, se necesita el metamodelo de dominio, en este caso, se toma el metamodelo de e*squemas relacionales*. Ambos metamodelos son independientes físicamente (véase Figura 8), aunque están relacionados lógicamente. En la Figura 8 se ha distinguido el metamodelo de medición con color claro y el metamodelo de dominio con color oscuro.

5.1. Ejemplo del Método de Medición Contar Elementos

En este apartado, se explica cómo se realiza la medición "*contar el número de tablas"* de un esquema relacional. Tal y como se ha presentado en el apartado 4.3, la medición se realizaría en las siguientes cuatro etapas:

1. Incorporación del metamodelo de esquemas relacionales (partes de color oscuro en la Figura 8).
2. Creación del modelo de medición de esquemas relacionales. Para la forma de medir "contar elementos de tipo tabla", los valores para *Entidad* y *Método de Medición* son Tabla y Contar, respectivamente. Todavía no se incluye el *Resultado de la Medición*.
3. En base al metamodelo del paso 1 se crea el modelo del esquema relacional sobre el cuál se va a aplicar la medición. En este caso, el esquema relacional contiene información de una escuela universitaria. El esquema está compuesto de 5 tablas, con sus correspondientes atributos, claves primarias y ajenas (véase Figura 9).

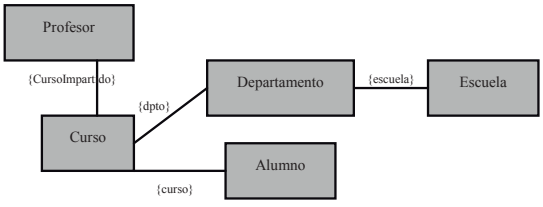


Figura 9. Ejemplo de modelo de dominio

4. Para ejecutar la medición, se utilizan como entradas el modelo de medición (etapa 2), el modelo de dominio (etapa 3); y el modelo QVT Relation (la obtención del modelo QVT Relation es automática y transparente al usuario, véase apartado 4.2). El modelo de salida que se consigue (véase Figura 10) contiene los resultados de medición definidos, es decir, el atributo *Resultado de la Medición* definido. En este caso con valor 5 (número de tablas).

```
<?xml version="1.0" encoding="ASCII"?>
<medicion:ModeloMedicion
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:medicion="http://procesoMedicion/medicion"
    nombreModelo="ModeloMedicion1">
  <mediciones name="Relaciones"
    metodo="Contar" elemento="Tabla"
    resultado="5"/>
</medicion:ModeloMedicion>
```

Figura 10. Resultado de la medición

La medición se lleva a cabo de forma automática gracias al modelo QVT Relation definido automáticamente mediante una transformación QVT del modelo de medición. A título informativo, en la Figura 11 y en la Figura 12 se muestra la especificación QVT equivalente al modelo QVT Relation. En dicha especificación QVT se pueden distinguir dos partes: una general (Figura 11) y otra más específica del modelo de medición (Figura 12).

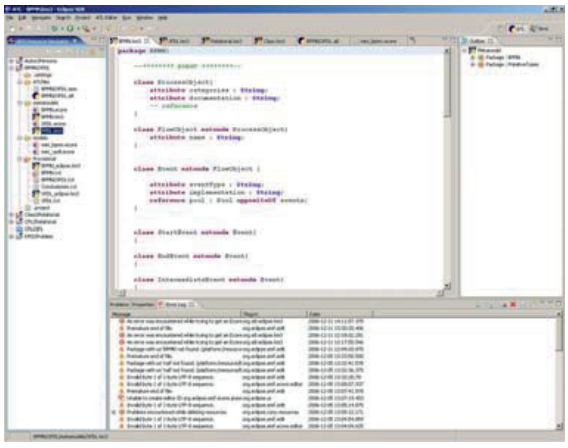


Figura 11. Parte genérica del código de la especificación QVT

En la parte general (Figura 11) se puede observar como la definición de los resultados es una llamada al método medición (representado en la Figura 12), que es una consulta OCL sobre el modelo de dominio. La consulta OCL es

específica en el sentido de que hay que conocer los elementos medibles y los métodos de medición para elaborar la consulta OCL.

```
function medicion(
    esquemaRelacional: EsquemaRelacional,
    metodo: String,
    elemento : String) : Integer
{
    if (metodo = 'contar') then
        if(elemento = 'Tabla') then
            relationalSchema.modelElements->
            select(m:ModelElement |
                m.oclIsTypeOf(Table))->size()
        else
            if(elemento = 'Atributo') then
                relationalSchema.modelElements->
                select(m:ModelElement |
                m.oclIsTypeOf(Atributo))->size()
            else
```

Figura 12. Parte específica del código de la especificación QVT

El proceso de transformación que permite obtener el modelo QVT Relation hace de la propuesta FMESP-Measurement una alternativa completamente genérica que le evita al usuario toda responsabilidad en cuanto a la escritura de la especificación de la transformación MDA para la medición.

6. Conclusiones y Trabajos Futuros

En este artículo se ha presentado un entorno genérico para la definición de modelos de medición basados en un metamodelo común y para la medición de cualquier entidad software a partir de los metamodelos que las representan. Siguiendo el enfoque de MDA, y partiendo del metamodelo de Medición (universal), es posible llevar a cabo la medición de cualquier dominio mediante transformaciones QVT, proceso que es completamente transparente al usuario.

Con el framework FMESP-Measurement, es posible medir cualquier entidad software sin más que disponer de los correspondientes metamodelos. La tarea del usuario consiste, únicamente, en la elección del metamodelo de dominio (según lo que se quiere medir en cada momento) y la definición de los modelos de entrada: modelo de dominio y de medición. El metamodelo de medición está integrado de base en el entorno.

El marco conceptual de la propuesta respeta completamente la filosofía MDA.

De cara al futuro, un trabajo importante es la realización de un plugin integrado en Eclipse que facilite al usuario la entrada de datos y el proceso de medición. Dicho plugin deberá servir para crear modelos de medición (de entrada) de manera fácil e intuitiva.

Adicionalmente, se pretende llevar a cabo diversas experiencias para validar esta propuesta con distintos dominios siguiendo las normas de estandarización adecuadas. En particular, se tendrá en cuenta la propuesta de la OMG, Software Metrics Meta-Model (SMM) [25], que actualmente está en fase de desarrollo.

Agradecimientos

Este trabajo ha sido parcialmente financiado por los proyectos ENIGMAS (Junta de Comunidades de Castilla-La Mancha, PBI-05-058), ESFINGE (Ministerio de Educación y Ciencia, TIN2006-15175-C05-05), COMPETISOFT (Programa Iberoamericano de Ciencia y Tecnología para el Desarrollo, 506AC0287) y META (Ministerio de Educación y Ciencia, TIN2006-15175-C05-01).

Referencias

[1] *The Maude System*, Department of Computer Science, University of Illinois, Urbana-Champaign.

[2] Auer, M., Graser, B. y Biffl, S.; *A Survey on the Fitness of Commercial Software Metric Tools for Service in Heterogeneous Environments: Common Pitfalls* Ninth International Software Metrics Symposium. (Metrics '03). (2003). pp.144.

[3] Bezivin, J., Jouault, F. y Touzet, D.; *Principles, standards and tools for model engineering*, 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'2005). (2005). pp.28-29.

[4] Boronat, A., Carsí, J. Á. y Ramos, I.; *An Algebraic Baseline for Automatic Transformations in MDA*, Workshop Software Evolution Through Transformations: Model-baased vs. Implementation-level Solutions (SEETra'04),2nd International Conference on Graph Transformation (ICGT2004), ENTCS. Roma (italia).

[5] Brown, M. y Dennis, G.; *Measurement and Analysis: What Can and Does Go Wrong?*, 10th IEEE International Symposium on Software Metrics (METRICS'04). (2004). pp.131-138.

[6] Dumke, R. R. y Grigoleit, H.; *Efficiency of CAMEtools in software quality assurance*, Software Quality Journal 6(2). (1997). pp.157-169.

[7] Eclipse, *EMF (Eclipse Modelling Framework)*,

[8] Fenton, N. y Pfleeger, S. L., *Software Metrics: A Rigorous & Practical Approach, Second Edition*, (1997). p.

[9] Ferreira, F., García, F., Ruiz, F., Bertoa, M. F., Calero, C., Vallecillo, A., Piattini, M. y Mora, B.; *Medición del Software: Ontología y Metamodelo*, Informe Técnico UCLM-TSI-001. (2006).

[10] Florac, W. A., Carleton, A. D. y Barnard, J.; *Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process*, IEEE Software. 17 (4). (2000).

[11] García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruíz, F., Piattini, M. y Genero, M.; *Towards a consistent terminology for software measurement*, Information and Software Technology 48. (2006). pp.631-644

[12] García, F., Piattini, M., Ruiz, F., Canfora, G. y Visaggio, C. A.; *FMESP: Framework for the modeling and evaluation of software processes*, Journal of Systems Architecture - Agile Methodologies for Software Production 52. (2006). pp.627-639

[13] García, F., Serrano, M., Cruz-Lemus, J., Ruiz, F. y Piattini, M.; *Managing Software Process Measurement: A Metamodel-Based Approach*, Information Sciences, In press. (2007).

[14] Harrison, W.; *A flexible method for maintaining software metrics data: a universal metrics repository*, Journal of Systems and Software 72. (2004). pp.225-234

[15] ISSI, G., Grupo ISSI: MOMENT (A framework for Model ManagemENT).

[16] Jokikyyny, T. y Lassenius, C.; *Using the internet to comunicate software metrics in a large organization*, Proceedings of GlobeCom'99. (1999).

[17] Kempkens, R., Rösch, P., Scott, L. y Zettel, J.; *Instrumenting Measurement Programs with Tools*, PROFES 2000. Oulu, Finland. (2000).

[18] Komi-Sirviö, S., Parviainen, P. y Ronkainen, J.; *Measurement Automation: Methodological Background and Practical Solutions-A Multiple Case Study*, Seventh International Software Metrics Symposium (METRICS'01). London. (2001).

[19] Lavazza, L. y Agostini, A.; *Automated Measurement of UML Models: an open toolset approach*, Object Technology. 4(4). (2005). pp.115-134.

[20] OMG, *Meta Object Facility (MOF) Specification; version 1.4*, Object Management Group. (2002).

[21] OMG, *XML Metadata Interchange (XMI) Specification. OMG Document formal/00-11-02*, Object Management Group. (2002).

[22] OMG, *Model-Driven Architecture (MDA) Guide Version 1.0.1*, Object Management Group. (2003).

[23] OMG, *OCL 2.0 - OMG Final Adopted Specification*, Object Management Group. (2003).

[24] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, Object Management Group. (2005).

[25] OMG, *Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (SMM). OMG Document: dmtf/2007-03-02*, Object Management Group. (2007).

[26] Palza, E., Fuhrman, C. y Abran, A.; *Establishing a Generic and Multidimensional Measurement Repository in CMMI context* 28th Annual NASA Goddard Software Engineering Workshop (SEW'03). Greenbelt (Maryland, USA). (2003). pp.12-22.

[27] Queralt, P., Hoyos, L., Boronat, A., Carsí, J. Á. y Ramos, I.; *Un motor de transformación de modelos con soporte para el lenguaje QVT relations*, Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). Sitges, Spain. (2006).

[28] Scotto, M., Sillitti, A., Succi, G. y Vernazza, T.; *A relational approach to software metrics*, Proceedings of the 2004 ACM symposium on Applied computing (SAC'2004). Nicosia, Cyprus. (2004). pp.1536-1540.

SOFTWARE MEASUREMENT BY USING QVT TRANSFORMATIONS IN AN MDA CONTEXT

Beatriz Mora, Félix García, Francisco Ruiz, Mario Piattini
Department of Information Technologies & Systems, University of Castilla-La Mancha, Ciudad Real, Spain
{Beatriz.Mora, Felix.Garcia, Francisco.RuizG, Mario.Piattini}@uclm.es

Artur Boronat, Abel Gómez, José Á. Carsí, Isidro Ramos
Department of Information Systems and Computation Polytechnic University of Valencia, Valencia, Spain
{aboronat, agomez, pcarsi, iramos}@dsic.upv.es

Keywords: Measurement, Model, FMESP, MDA, QVT, SMF.

Abstract: At present the objective of obtaining quality software products has led to the necessity of carrying out good software processes management in which measurement is a fundamental factor. Due to the great diversity of entities involved in software measurement, a consistent framework is necessary to integrate the different entities in the measurement process. In this work a Software Measurement Framework (SMF) is presented to measure any type of software entity. In this framework, any software entity in any domain could be measured with a common Software Measurement metamodel and QVT transformations. This work explains the three fundamental elements of the Software Measurement Framework (conceptual architecture, technological aspects and method). These elements have all been adapted to the MDE paradigm and to MDA technology, taking advantage of their benefits within the field of software measurement. Furthermore an example which illustrates the framework's application to a concrete domain is furthermore shown.

1 INTRODUCTION

The current necessity of the software industry to improve its competitiveness forces continuous process improvement. This must be obtained through successful process management (Florac, et al., 2000). Measurement is an important factor in the process life cycle due to the fact that it controls issues and lacks during software maintenance and development. In fact, measurement has become a fundamental aspect of Software Engineering (Fenton, et al., 1997).

Software Processes constitute the work base in a software organization. Companies therefore wish to carry out an effective and consistent software measurement to facilitate and promote continuous process improvement. To do this, a discipline for data analysis and measurement (Brown, et al., 2004), and measure definition, compilation and analysis in the process, projects and software products, is needed.

The great diversity in the kinds of entities which are candidates for measurement in the context of the software processes points to the importance of providing the means through which to define measurement models in companies in an integrated and consistent way. This involves providing companies with a suitable and consistent reference for the definition of their software measurement models along with the necessary technological support to integrate the measurement of the different kinds of entities.

With the objective of satisfying the exposed necessities, it is highly interesting to consider the MDE (Model-Driven Engineering) paradigm (Bézivin, et al., 2005) in which software measurement models (SMM) are the principal elements of the measurement process. Its main goal is to ensure that the core artifacts in software engineering processes will be models rather than code, so that designs are expressed and managed in the manner of models with a much higher level of abstraction than the code. MDA (Model-Driven Architecture) is the OMG proposal by which to carry out the MDE Paradigm. The core of MDA is a set of standards (MOF, QVT, OCL and XMI).

According to the QVT standard, the software development process is a set of model

Mora, B., García, F., Francisco Ruíz, Piattini, M., Boronat, A., Gómez, A., Carsí, J. A., Ramos, I. Software Measurement by Using QVT Transformations in an MDA Context (ICEIS 2008). Proc. of the Tenth International Conference on Enterprise Information Systems, Volume DISI. 2008.

transformations, from an abstract to a specific level. The requirements are in the more abstract level and the code is in the more specific level.
Software measurement can benefit from the MDE paradigm, providing integration and support to carry out an automatic software measurement of any software type. This implies that: a) the definition of measurement models conform to a Software Measurement metamodel; b) the definition of generic measurement methods are applicable to any model-based software artifact; and c) support for computing measures, for storing results and for enhancing decision making.

These aspects constitute the main interest of this paper, in which the application of MDA principles, standards and tools are used in software measurement. The goal of this proposal is to develop a generic framework to define measurement models which conform to a common measurement metamodel, and to measure any software entity with regard to a domain metamodel. In order to develop this proposal, MOMENT environment has been used, which supports the automatic model management MDA compliant.

Publications of García, Bertoa, et al. (2006); García, Piattini, et al. (2006); and García, et al. (2007) were used as a starting point for this work. These works present FMESP, which consists of a framework based on MOF Architecture. This includes a software measurement ontology and metamodel, and the GenMETRIC tool which is used to define software measurement models, and to calculate defined measures for these models. The ontology permits the identification of all the concepts, proportions exact definitions for all the terms and clarifies the relationship between them. This paper presents an adaptation of FMESP to MDA, which is described in detail in following sections.

The remainder of the paper is organized as follows. Section 2 provides an overview of related works and Section 3 describes the Software Measurement Framework (SMF), including conceptual architecture, technological aspects, and method. In Section 4 the use of the framework is illustrated with an example. Finally, conclusions and future works are outlined in Section 5.

2 RELATED WORKS

We have found numerous publications which deal with tools that have important success factors in software measurement efforts (Komi-Sirviö, et al., 2001), which supply work environments and general approximations (Kempkens, et al., 2000), or which give architectures more specific solutions (Jokikyyny, et al., 1999). Brown, et al. (2004) include a list of tools which support the creation, control and analysis of software measurements. Auer, et al. (2003) furthermore examine various software measurement tools, such as MetricFlame, MetricCenter, Estimate Professional, CostXPert and ProjectConsole, in heterogenic environments.

It is also possible to find certain proposals through which to tackle software measurement which are more integrated and less specific than in the aforementioned cases. Palza, et al. (2003) propose the MMR tool which is based on the CMMI model for the evolution of software processes, and it is possible to consult similar tools of Harrison, (2004); Lavazza, et al. (2005); and Scotto, et al. (2004). These proposals are, however, restricted to concrete domains or to evaluation models of specific quality.

Vépa, et al. (2006) present a metamodel which allows the storage of measurement data, and a set of transformations through which to carry out the measurement of models based on a metamodel is presented. This paper focuses upon the technological aspects needed to implement the software measurement with ATL technology, by offering the user a variety of graphic representations of the measurement results obtained.

This final proposal and that which is presented here are complementary as they both focus upon two key support elements of generic measurement: the conceptual base, which is the main contribution of FMESP, and technological implementation. Some differences from technological point of view exist.

The measurements which are applied in the work of Vépa, et al. (2006) are previously defined in the ATL transformation archives. The measurable entities are typical of the metamodels presented in this work (KM3 and UML2). For example, the measurable entities for a model which is expressed in km3 might be package, class, attribute, reference etc.

The measurements in the proposal presented here are defined by the user, i.e. the model transformation needed to carry out the measurement it is not a model previously defined, but this model is defined according to the users needs. The measurement definition is possible thanks to the software measurement model, which contains all that is relative to the measurement to be carried out in each case. Moreover, the measurable entities are those which are defined in their corresponding domain and measurement metamodel (expressed in ecore). A further difference is that SMF uses QVT.

3 SOFTWARE MEASUREMENT FRAMEWORK

In order to carry out this proposal it was considered of interest to adapt FMESP to the MDE paradigm. The objective of this was to exploit the benefits that the paradigm could contribute to software measurement by, on one hand adopting the software measurement metamodel defined in FMESP, and on the other by evolving GenMETRIC to an environment which would allow the definition of software measurement models and the computation of the models defined. All this would take place within the context of models and model transformations of the MDA architecture. The Software Measurement Framework (SMF) is the evolution of the FMESP, but is adapted to the MDE paradigm and uses MDA technology.

The following subsections explain the conceptual, technological and methodological elements which are part of SMF.

3.1 Conceptual Architecture

Due to the necessity of having a generic and homogeneous environment for software measurement (García, Bertoa, et al., 2006; García, Piattini, et al., 2006; García, et al., 2007), a conceptual architecture and a tool with which to integrate the software measurement are proposed. In the following section, the main characteristics of this proposal are described. In García, et al. (2007) a more detailed description can be found.

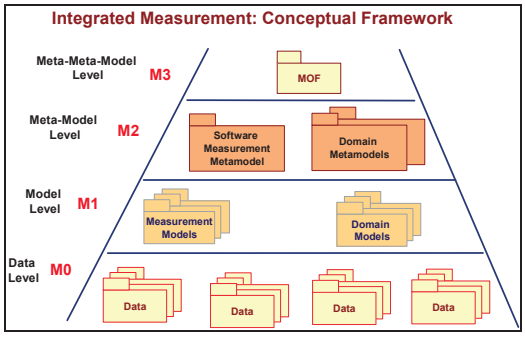


Figure 1: Conceptual framework with which to manage software measurement.

The proposed software measurement described in this paper is part of the FMESP framework (García, Piattini, et al., 2006). The FMESP framework permits representing and managing software processes from the perspectives of modeling and measurement. We focus on the measurement support of the framework whose elements are detailed according to the three layers of abstraction of metadata that they belong to, according to the MOF standard. In Figure 1, the conceptual architecture for integrated measurement is represented.

As can be observed in Figure 1, the architecture has been organized into the following conceptual levels of metadata:

- Meta-MetaModel Level (M3). At this level, an abstract language for the definition of metamodels, is found. This is the MOF language.
- Metamodel Level (M2). In the M2 level, two generic metamodels which conform with this framework are required. These are: the Measurement Metamodel, to define specific measurement models; and Domain Metamodels, to represent the kinds of entities which are candidates for measurement in the context of the evaluation of the software processes, such as, UML and Process metamodels.
- Model Level (M1). Specific models are included at this level. These models may be of two types: Measurement Models, which are examples of the measurement metamodel in the M2 level and which are defined in such a way as to satisfy some of the company's information needs; and Domain Models, which are defined according to their corresponding domain metamodels.

In order to establish and clarify the concepts and relationships that are involved in the software measurement domain before designing the metamodel, an ontology for software measurement was developed (García, Bertoa, et al., 2006). The measurement metamodel was derived by using the concepts and relationships stated in the ontology as a base. The Software Measurement metamodel (which is integrated in SMF) is organized around four main packages (for greater detail see the work of García, Bertoa, et al. (2006)):

- **Software Measurement Characterization and Objectives**, which includes the constructors required to establish the scope and objectives of the software measurement process.
- **Software Measures**, which aim at establishing and clarifying the key elements in the definition of a software measure.
- **Measurement Approaches**. This package introduces the element of measurement approach to generalize the different

approaches used by the three kinds of measures to obtain their respective measurement results. A base measure applies a measurement method. A derived measure uses a measurement function. Finally, an indicator uses an analysis model to obtain a measurement result that satisfies an information need.

- **Measurement Action**. This establishes the constructs related to the act of measuring software. A measurement (which is an action) is a set of measurement results, for a given attribute of an entity, using a measurement approach. Measurement results are obtained as the result of performing measurements (actions).

3.2 Technological Aspects

In this section the technological aspects of SMF are explained.

3.2.1 Adaptation to MDA

In Figure 2 the necessary elements for the FMESP adaptation to MDA are presented according to MOF levels.

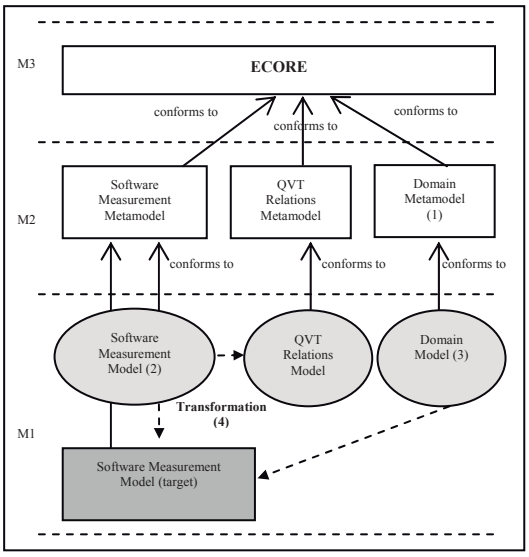


Figure 2: Elements of the FMESP adaptation in a MDA context.

As can be observed in Figure 2, two new elements, namely the QVT Relations model and metamodel, have been added to adapt the conceptual architecture illustrated Figure 1 to MDA. The QVT Relations Model (which is described in greater detail in Section 3.2.2) is obtained automatically through a transformation from a Measurement model. It contains all the information necessary to carry out the transformation of the SMF proposal. Ecore language has been selected because it is a common modeling language based on EMOF. EMOF is the part of the MOF 2.0 specification that is used for defining simple metamodels using UML-like concepts.

3.2.2 QVT Relations Transformation

The QVT Relations model is the transformation needed to perform the measurement. In this transformation two source models are involved: a Software Measurement model and a domain model; the target model is the Software Measurement Model with the measurement results (see Figure 2). Due to the fact that the proposal is about generic measurement, it is very important that the QVT model is obtained in a generic way. The MDE paradigm and MDA technology are applied for this reason.

This transformation is obtained automatically from the previous QVT transformation shown in Figure 3. The QVT Relations model, called the extended or final QVT Relations model, is obtained from a QVT transformation, where there are two source models: the basic or initial QVT Relations model (which conforms to the QVT Relations metamodel) and the Software Measurement model (previously defined).

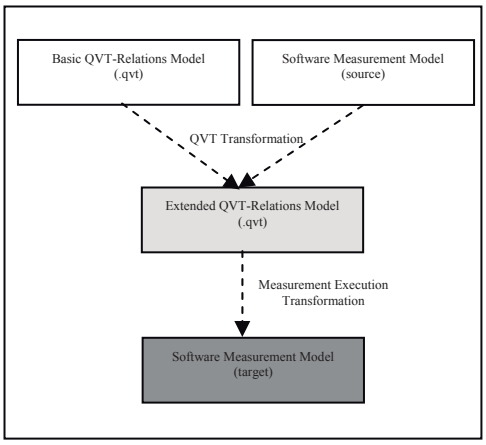


Figure 3: QVT Relations transformation model.

The extended QVT Relations model extends the basic QVT Relations model with the following aspects:

- **Transformation Model:** to obtain the extended QVT Relations model, the source

model specification is needed. In this case, there are two source models: the Software Measurement model and the domain model. Due to the fact that the Software Measurement model is always the same, this model is already defined in the basic QVT Relations model. Therefore, only the domain model needs to be defined. This information is taken from the Software Measurement model which contains all the measurement information.

- **Relation Domain:** in order to perform the transformation, it is necessary to define the *checkonly domain* rules. In this case there are two, one for each source model: the domain model and the Software Measurement model. It is only necessary to define *checkonly domain* of the domain model, because checkonly domain of the measurement model is already defined in the basic QVT Relations model.
- **Function:** this contains the necessary OCL queries to carry out the measurement. These OCL queries are the implementations of the "*Measurement Action*" package defined in the Software Measurement Metamodel.

These elements are empty in the basic QVT Relations model, and they are extended to obtain the extended QVT Relations model, the transformation model necessary to carry out the measurement. In the Figure **4** all the Software Measurement process is shown.

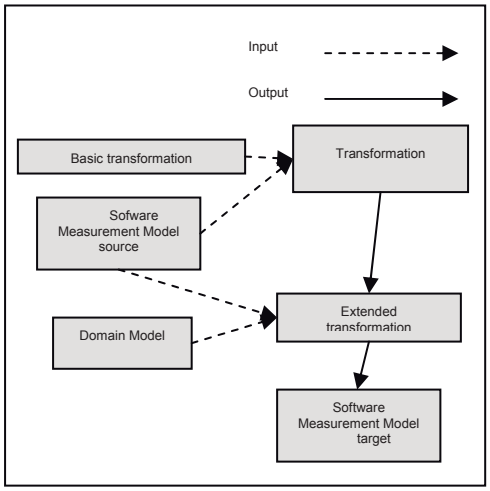


Figure 4. Software Measurement process.

3.2.3 Technological Environment

In this paper, the tool selected has been the model management environment called MOMENT (MOment manageMENT)(Boronat, et al., 2007). This framework is integrated in the Eclipse platform. It provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF)("Eclipse Modelling Framework (EMF) Main Page," 2007). The underlying formalism of the model management approach is the algebraic language Maude ("The Maude System," 2007).

From a functional point of view, MOMENT has two components: OCL query execution (MOMENT-OCL) and QVT Transformations (MOMENT-QVT).

MOMENT-OCL (Boronat, et al., 2006) has implemented an editor integrated in the Eclipse platform to check OCL invariants and to execute OCL queries over instances of ecore models. It uses Ecore models in the entire software development process to store the OCL expressions by following the Model Driven Engineering approach. One advantage of this is the persistence mechanisms in XMI that EMF provides automatically. In this work, it has been used in order to check and validate the OCL queries used in the QVT transformations. The results have been shown by screen.

On the other hand, the MOMENT-QVT tool (Queralt, et al., 2006) is a model transformation engine that provides partial support for the QVT Relations language. It implements the metamodel definition QVT, given in the QVT standard, and provides an editor for the QVT Relations language, which permits the definition of model transformations between EMF metamodels.

In order to carry out a QVT transformation in MOMENT, a transformation textual specification (coded by the Textual QVT Editor and stored in a .qvtext) or, its equivalent QVT Relations model (stored in a .qvt) can be used. This model conforms to the QVT Relations metamodel and it is possible to obtain it by parsing the textual specification.

3.3 Method

The necessary steps to carry out the software measurement by using the SMF are explained below (see Figure 2):

1. **Incorporation of Domain Metamodel:** the measurement is made in a specific domain. This domain must be defined according to its metamodel (it is situated in the M2 level and it conforms to the Ecore meta-metamodel).
2. **Creation of Measurement Model:** the measurement model is created according to the Software Measurement metamodel which is integrated in SMF. This first model is the source model, so the results are therefore still not

defined, i.e. the "*Measurement Action*" package from the Software Measurement metamodel is still not instantiated.

3. **Creation of Domain Model:** which is defined according to its corresponding domain metamodel (created in the first step). The domain models are the entities whose attributes are measured by calculating the measurements defined in the corresponding measurement models. Examples of domain models are: the UML models (use cases, class diagrams, etc.), or the E/R models.
4. **Measurement Execution:** the measurement execution is carried out through QVT transformation, in which, the measurement model is obtained by starting from the two source models (the measurement model and the domain model) where the results are defined, i.e. the "*Measurement Action*" package is instantiated. The target measurement model is the extension of the source measurement model. The measurement results are calculated by running OCL queries on the domain model.

An example of the method application is shown in the following section.

4 EXAMPLE

To illustrate the benefits of the proposal, consider the example of relational database measurement. For greater simplicity, only the following elements are shown in Figure 5: Measurement *Method, Entity* (to which the measurement method is applied) and *Measurement result* (the result is obtained by executing the measurement method on the entity).

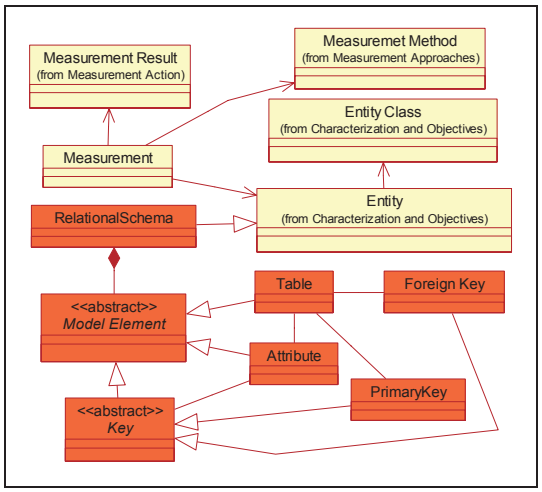


Figure 5: Relationship between Relational Database (domain) Metamodel and SMM.

Furthermore, it is necessary for the domain metamodel, in this case Relational Databases domain, to have been previously chosen. Both metamodels are independent (Figure 5), although they are logically related. In Figure 5 the measurement and domain metamodels have been represented in a clear and a dark colour, respectively.

In this example, the chosen measurement method has been "COUNT elements of type TABLE", which is an instantiation of the abstract method "COUNT elements of type X".

In order to carry out the measurement, the following steps (four steps) must take place:

1. Incorporation of Relational Databases metamodel (represented in a dark colour in Figure 6).
2. Creation of measurement model conforms to Software Measurement metamodel. For the measurement method "COUNT elements of type TABLE", the values of *Entity* and *Measurement Method* are Table and Count, respectively. The *Measurement Result* is not still defined.
3. Creation of model conforms to the Relation Database metamodel. In this case, the model (relational schema) is a university domain composed of five tables with their corresponding primary keys (bold and shaded), foreign keys (underlined and italic), and attributes (see Figure 6).

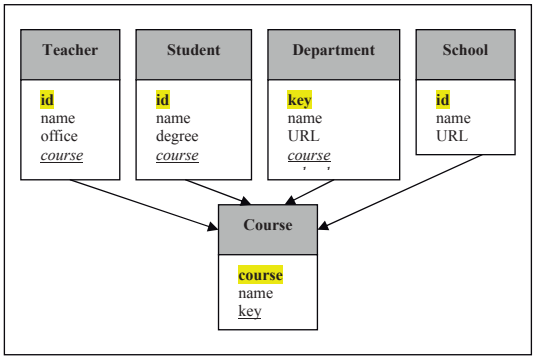


Figure 6: Relational Database model (relational schema).

The extended QVT Relations model was needed to carry out the fourth step. This transformation is obtained automatically (see section 3.2.2). The extended elements are detailed below:

- Transformation Model: the target model is the relational databases domain model.
- Relation Domain: the checkonly domain of the relational schema domain is indicated (see Figure 7).

```
checkonly domain relationalDomain
 srcRelationalSchema : RelationalSchema {
    name = myRelationalSchema
};
checkonly domain measurementDomainSrc
 srcMeasurementModel : MeasurementModel {

    modelName = myModelName,

    measurements = dstMeasurement1 : Measurement {
        name = myMeasurementName,
        method = dstMethod : MeasurementMethod{
            nameMethod = myMethod
        },
        entity = dstEntity : Entity{
            nameEntity = myEntity
        },
        result = dstResult : MeasurementResult{}
    }  //result not defined
};
```

Figure 7: "Relation Domain" elements from extended QVT Relations model.

- Function: this contains the OCL queries with which to perform the measurement, in this case, the queries necessary to implement the "count element of type X" measurement method where X is Table (see Figure 8).

```
    enforce domain measurementDomainDst
      dstMeasurementModel : MeasurementModel {
        modelName = myModelName,

        measurements = dstMeasurement1 : Measurement {
            name = myMeasurementName,
            method = dstMethod : MeasurementMethod{
                nameMethod = myMethod
            },
            entity = dstEntity : Entity{
                nameEntity = myEntity
            },
            result = measurementAction(srcRelationalSchema,
             myMethod, myEntity)
        }
    };
}//TOP RELATION
function measurementAction(relationalSchema: RelationalSchema,
    method: String, entity: String) : Integer
{
    relationalSchema.modelElements->select(m:ModelElement
    |m.oclIsTypeOf(Table))->size()
}
```

Figure 8: "Function" elements from extended QVT Relations model.

4. The source models used to carry out the measurement are: the measurement model (2nd step), the domain model (3rd step) and the extended QVT Relations model. The target model obtained is the measurement model with defined *Measurement Result* (see Figure 9). In this example the value of *Measurement Result* is 5 (number of tables).

```
<?xml version="1.0" encoding="ASCII"?>
<measurement:MeasurementModel xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:measurement="http://bmora/metamodels/measurement"
    modelName="ER_MEASUREMENT">
  <measurements name="RELATIONAL SCHEMA MEASUREMENT">
    <method nameMethod="COUNT"/>
    <entity nameEntity="TABLE"/>
    <result result="5"/>
  </measurements>
</measurement:MeasurementModel>
```

Figure 9: Measurement result.

In the same way as is illustrated with Relational Databases, the method can be applied to any other domains, such as for example, UML models, Project Management or Business Processes, etc.

5 CONCLUSIONS AND FUTURE WORK

In this paper, a generic framework for the definition of measurement models based on a common metamodel has been presented. The framework allows the integrated management and measurement of a great diversity of entities.

Following the MDA approach and starting from a (universal) measurement metamodel, it is possible to carry out the measurement of any domain by means of QVT transformation, and this process is completely transparent to the user.

With SMF, it is possible to measure any software entity. The user task consists in selecting the domain metamodel (the domain to be measured) and defining the source models. The software metamodel is integrated in the framework.

At the present time, a Software Measurement Modeling Language (SMML) is being developed to supply measurement engineers with the definition of software measurement models according to the proposed metamodel; this language will be integrated in SMF.

Among related future works, one important work is the realization of a plug-in based on Eclipse which will supply the user with the data introduction and the measurement process. This plug-in will enable users to instantiate measurement models in an easy and intuitive way. Other future work will be to align our metamodel with the Software Metrics Meta-Model (SMM) OMG proposal (OMG, 2007), which is at present in its development phase. Finally, we shall apply SMF to real environments to obtain further refinements and validation.

ACKNOWLEDGEMENTS

This work has been partially financed by the following projects: ENIGMAS (Junta de Comunidades de Castilla-La Mancha, PBI-05-058), ESFINGE (Ministerio de Educación y Ciencia, TIN2006-15175-C05-05) and META (Ministerio de Educación y Ciencia, TIN2006-15175-C05-01).

REFERENCES

Auer, M., Graser, B., & Biffl, S. (2003). *A Survey on the Fitness of Commercial Software Metric Tools for Service in Heterogeneous Environments: Common Pitfalls* Paper presented at the Ninth International Software Metrics Symposium. (Metrics '03).

Bézivin, J., Jouault, F., & Touzet, D. (2005). *Principles, standards and tools for model engineering.* Paper presented at the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'2005).

Boronat, A., & Meseguer, J. (2007). *Algebraic Semantics of EMOF/OCL Metamodels* (No. Technical Report UIUCDCS-R-2007-2904): CS Dept., University of Illinois at Urbana-Champaign.

Boronat, A., Ramos, I., & Carsí, J. Á. (2006, Marzo 2003). *Definition of OCL 2.0 Operational Semantics by means of a Parameterized Algebraic Specification.* Paper presented at the WAFOCA'06. First International Workshop on algebraic foundations for OCL and Applications.

Brown, M., & Dennis, G. (2004). Measurement and Analysis: What Can and Does Go Wrong? *10th IEEE International Symposium on Software Metrics (METRICS'04)*, 131-138.

. Eclipse Modelling Framework (EMF) Main Page (2007), from http://www.eclipse.org/emf

Fenton, N., & Pfleeger, S. L. (1997). *Software Metrics: A Rigorous & Practical Approach, Second Edition*: PWS Publishing Company.

Florac, W. A., Carleton, A. D., & Barnard, J. (2000). Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process. *IEEE Software, 17 (4)*(4).

García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruíz, F., Piattini, M., et al. (2006). Towards a consistent terminology for software measurement. *Information and Software Technology 48*(8), 631-644

García, F., Piattini, M., Ruiz, F., Canfora, G., & Visaggio, C. A. (2006). FMESP: Framework for the modeling and evaluation of software processes. *Journal of Systems Architecture - Agile Methodologies for Software Production 52*(11), 627-639

García, F., Serrano, M., Cruz-Lemus, J., Ruiz, F., & Piattini, M. (2007). Managing Software Process Measurement: A Metamodel-Based Approach. *Information Sciences*.

Harrison, W. (2004). A flexible method for maintaining software metrics data: a universal metrics repository. *Journal of Systems and Software 72*(2), 225-234

Jokikyyny, T., & Lassenius, C. (1999). *Using the internet to comunicate software metrics in a large organization.* Paper presented at the Proceedings of GlobeCom'99.

Kempkens, R., Rösch, P., Scott, L., & Zettel, J. (2000). *Instrumenting Measurement Programs with Tools.* Paper presented at the PROFES 2000, Oulu, Finland.

Komi-Sirviö, S., Parviainen, P., & Ronkainen, J. (2001). *Measurement Automation: Methodological Background and Practical Solutions-A Multiple Case Study.* Paper presented at the Seventh International Software Metrics Symposium (METRICS'01), London.

Lavazza, L., & Agostini, A. (2005). Automated Measurement of UML Models: an open toolset approach. *Object Technology, 4(4)*, 115-134.

. The Maude System (2007), from http://maude.cs.uiuc.edu/

OMG (2007). *Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (SMM). OMG Document: dmtf/2007-03-02*: Object Management Group.

Palza, E., Fuhrman, C., & Abran, A. (2003). *Establishing a Generic and Multidimensional Measurement Repository in CMMI context* Paper presented at the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), Greenbelt (Maryland, USA).

Queralt, P., Hoyos, L., Boronat, A., Carsí, J. Á., & Ramos, I. (2006, October 2006). *Un motor de transformación de modelos con soporte para el lenguaje QVT relations.* Paper presented at the Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06), Sitges, Spain.

Scotto, M., Sillitti, A., Succi, G., & Vernazza, T. (2004). *A relational approach to software metrics.* Paper presented at the Proceedings of the 2004 ACM symposium on Applied computing (SAC'2004), Nicosia, Cyprus.

Vépa, É., Bézivin, J., Brunelière, H., & Jouault, F. (2006). *Measuring Model Repositories.* Paper presented at the Model Size Metrics Workshop at the MoDELS/UML 2006 conference, Genoava, Italy.

*Quality-Driven Model Transformations: From Requirements to UML Class Diagrams**

Silvia Abrahão[1], Marcela Genero[2], Emilio Insfran[1], José Ángel Carsí[1] Isidro Ramos[1] and Mario Piattini[2]

[1] ISSI Research Group, Department of Information Systems and Computation, Valencia University of Technology, Camino de Vera s/n, 26022, Valencia, Spain, {sabrahao, einsfran, pcarsi, iramos}@dsic.upv.es

[2] Alarcos Research Group, Department of Information Systems and Technologies, University of Castilla-La-Mancha, Paseo de la Universidad 4, 13071, Ciudad Real, Spain, {Marcela.Genero, Mario.Piattini}@uclm.es

Abstract

Model-Driven Architecture (MDA) is a software engineering approach that promotes the use of models and model transformations as primary development artifacts. Usually, there are several ways to transform a source model into a target model. Alternative target models may have the same functionality but may differ in their quality attributes (e.g., understandability, modifiability). This chapter presents an approach to deal with quality-driven model transformations. Specifically, it focuses on a specific set of transformations to obtain UML class diagrams from a Requirements Model. A set of alternative transformations are identified, and the selection of the best alternative is done through a controlled experiment. The goal of the experiment is to empirically validate which alternative transformation produces the UML class diagram that is the easiest to understand. This evidence can be further used to define high-quality transformation processes, as it will be based on empirical knowledge rather than on common wisdom and the intuition of the researchers and developers.

* This work is supported by the META project (TIN2006-15175-C05-01).

Abrahao, S., Genero, M., Insfrán, E., Carsí, J. A., Ramos, I., Piattini, M. Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In Rech, J. and Bunse, C. (Eds.), Model-Driven Software Development: Integrating Quality Assurance. 2008.

1. Introduction

Nowadays, the software development community is moving towards model-driven development processes whose goal is the development of software at a higher level of abstraction based on models and model transformations. Within this context, the Model-Driven Architecture (MDA) initiative (OMG, 2003) has attracted interest from both the research community and software practitioners. This approach comprises the use of models in all the steps of a software development project, until the delivery of the software on a given platform.

A MDA development process basically transforms a platform-independent model (PIM) into one or more platform-specific models (PSM), which are transformed into code (code model – CM). The CM is just the actual code generated from PSMs through transformation. Here, the goal is to decouple the way, in which software systems are currently defined, which is dependant on the technology they use (OMG, 2003).

A model transformation is a process of converting one model to another model. A model may be transformed to several alternative models that may have the same functionality but different quality attributes. For example, one model may be more reusable while another model may be more comprehensive to its stakeholders. Therefore, it is necessary to identify those transformations that produce models with the desired quality attributes.

To cope with the problem of selecting alternative transformations, this chapter presents an approach for quality-driven model transformations. The mechanisms to choose the appropriate alternatives can greatly differ depending on the nature and the domain of the transformations as well as the quality perspective that is chosen. We focus on a set of transformations defined to obtain UML class diagrams from a Requirements Model (Insfran, 2003). Assuring quality in representing the system's conceptual model from requirements is particularly important, as the traceability between these models is not properly dealt with. Moreover, a conceptual model of good quality can help to minimize communication problems and misunderstandings of requirements among the stakeholders.

The quality perspective that we are interested in is the *pragmatic quality*[1] (Lindland, Sindre & Sølvberg, 1994). This quality category addresses the

[1] There are two other types of quality according to Lindland et al.'s framework: *syntactic quality*, which is the degree to which model contains flaws, and *semantic quality*, which is the degree to which the model is valid (contains all statements in the model that are correct and relevant to the problem domain) and complete (contains all statements about the problem domain that are correct and relevant). For the purpose of

comprehension aspect of the model from the stakeholders' perspective. Pragmatic quality captures how the model has selected an alternative "from among the many ways to express a single meaning", and it essentially deals with making the model *easy to understand*.

The comprehension goal specifies that all audience members (or interpreters) completely understand the statements in the model that are relevant to them. This is an import quality attribute since it is recognized as one of the main factors that influences maintainability (Selic, 2003) (Otero & Dolado, 2004) (Reinhartz-Berger & Dori, 2005) (Genero et al., 2005; 2007). A UML class diagram must first be understood before any desired changes to it can be identified, designed, or implemented. In terms of the Lindland et al. framework, improving pragmatic quality means increasing the degree of correspondence between the set of statements in the model and the set of statements that the user thinks the model presents (i.e. their understanding of the model).

Therefore, our main goal is to empirically evaluate which of the alternative transformations produces the UML class diagram that is easiest to understand. This evidence can be further used to define high-quality transformation processes, as it will be based on empirical knowledge rather than on common wisdom and the intuition of the researchers and developers.

The structure of the chapter is as follows. Section 2 presents the state-of-the-art for quality in model-driven development. Section 3 describes how UML class diagrams can be obtained from a Requirements Model using different transformation alternatives. This section also shows the definition of these transformations using QVT and their execution in a platform for model management called MOMENT. Section 4 describes the design and the results of the experiment carried out to empirically validate the selection of the alternative transformations according to the 'understandability' quality attribute. Section 5 describes our conclusions. Finally, section 6 presents a discussion on future research directions.

2. State-of-the-Art of Quality for Model-Driven Software Development

In the last few years, some proposals that deal with the quality of model transformations from the perspective of a quality attribute have been proposed. An organized chronological summary of these studies is presented in Table 1.

Zou and Kontogiannis (2003) proposed a quality-driven reengineering framework for object-oriented migration. Analysis tools, transformation rules, and non-functional requirements for the target migration systems characterize this framework. During the migration process, the source-code transformation

---

our work, we focus on the evaluation of the pragmatic quality of the models obtained with alternative transformations, leaving the evaluation of the other quality perspectives as a topic for future research.

rules are associated with quality features of the target system (i.e., coupling and cohesion). This approach was applied to transform a set of gnu AVL libraries into an UML class diagram.

Table 1. Comparison of approaches for quality in model-driven development

Proposal	Purpose	Type of Transformation	Input Artifact	Quality attributes	Automation
Zou and Kontogiannis, 2003	Reverse engineering (migration)	Vertical (CM-to-PIM)	Program code	Coupling and cohesion	No
Rottger and Zschaler, 2004	Refinement	Horizontal	Context Models	Response Time	Partial
Merilinna, 2005	Refactoring	Horizontal (PIM-to-PIM)	Architectural models	Performance, availability, reliability, maintainability, modifiability and reusability	Yes
Kurtev, 2005	Synthesis	Vertical (PIM-to-PIM)	UML class models	Adaptability	Yes (Mistral)
Markovic and Baar, 2005	Refactoring	Horizontal (PIM-to-PIM)	UML class models	Syntactical correctness	No
Sottet et al., 2006	–	–	Interface models	Compatibility, error protection, homogeneity-consistency	No
Ivkovic and Kontogiannis, 2006	Refactoring	Horizontal (PIM-to-PIM)	Architectural models expressed in UML	Maintenance, performance and security	No
Kerhervé et al., 2006	Synthesis, refinement	Horizontal and Vertical	Information models	Response time, network delay, network bandwidth	No

(–) means that the proposal does not provide this information

Röttger and Zschaler (2004) proposed an approach for refining non-functional requirements based on the definition of context models and their transformations. This approach has been defined in a software development process that separates the roles of the measurement designer and the application designer. It is the measurement designer's responsibility to specify measurements, context models and transformations among these models. Then, the application designer can apply the transformations when developing a system. Röttger and Zschaler defined a XML-based language for the

specification of transformations between abstract and concrete context models. The transformations used the response time quality attribute.

Merilinna (2005) proposed a tool for quality-driven model transformations for software architectures. Two types of quality attributes are considered: attributes related to software execution (e.g., performance, availability, reliability) and attributes related to software evolution (e.g., maintenance, modifiability, reusability). The transformations are described according to MDA and a proprietary transformation rule language. The approach only considers horizontal transformations (PIM-to-PIM transformations).

Kurtev (2005) proposed a formal technique for the definition of transformation spaces that support the analysis of alternative transformations for a given source model. This technique provides operations for the selection and reduction of transformation spaces based on certain desirable quality properties of the resulting target model. Specifically, this approach deals with the adaptability of model transformations. To generate the transformation space, the process takes a source model and its metamodel, the target metamodel, and the quality properties as input. The proposal has been applied to a set of transformations to obtain XML schemas from UML class diagrams.

Markovic and Baar (2005) defined a set of transformation rules for the refactoring of UML class diagrams. The rules have been defined using the Query/View/Transformation (QVT) standard of OMG (OMG, 2005). The refactoring is applied to UML class diagrams containing annotated OCL constraints that are preserved when the transformations are applied. Therefore, the syntactical correctness of the target model is preserved.

Similar to this proposal, Ivkovic and Kontogiannis (2006) presented an approach for the refactoring of software architectures using model transformations and semantic annotations. In this approach, the architectural view of a software system is represented as a UML profile with its corresponding stereotypes. Then, the instantiated architectural models are annotated using elements of the refactoring context, including soft goals, metrics, and constraints. Finally, the actions that are most advisable for a refactoring context are applied after being selected from a set of possible refactorings. The proposal has been applied to a case study to demonstrate that the refactoring transformations improve the maintenance, performance and the security of a software system.

Sottet et al. (2006) proposed an approach for model-driven mappings for embedding the description and control of usability. A mapping describes a model transformation that preserves properties. The mapping properties provide the designer with a means for both selecting the most appropriate transformation and previewing the resulting design. A case study that illustrates an application of the mapping metamodel using usability criteria (compatibility, error protection, and homogeneity-consistency) was presented.

Kerhervé et al. (2006) proposed a general framework for quality-driven delivery of distributed multimedia systems. The framework focuses on Quality of Services (QoS) information modeling and transformations. The transformations between models express the relationships among the concepts of the different quality information models. These relationships are defined in quality dimensions and are used to transform instances of a source model to a target model. Different types of transformations are applied to different layers and services: vertical transformations are applied to transform information between the different layers (user, service, system, and resource), and horizontal transformation are applied to interchange information between services of the same layer.

In summary, some proposals focus on defining horizontal transformations for model refactoring (Merilinna 2005) (Markovic & Baar 2005) (Ivkovic & Kontogiannis 2006). Other proposals are aimed at providing vertical transformations for model refinement (Rottger & Zschaler, 2004), synthesis (Kerhervé et al., 2006) (Kurtev, 2005), or reverse engineering (Zou & Kontogiannis, 2003). Of these studies, only the one by Kurtev (2005) presents a more systematic approach for selecting alternative transformations according to a given quality attribute.

All these approaches propose quality criteria that can be used to drive the transformations, but very few of these approaches (Kurtev, 2005) (Markovic & Baar, 2005) illustrate them by means of practical examples. With the exception of Markovic and Baar (2005) and Kurtev (2005), the transformations are poorly defined. Therefore, more systematic approaches to ensure quality in MDA processes are needed. Another weakness of these proposals is that they are not empirically validated. The practical applicability of model transformations is reported based on the intuition of the researcher. As pointed out by Czarnecki and Helsen (2006), there is a lack of controlled experiments to fully validate the observations made by the researchers.

3. A Quality-Driven Model Transformation Approach

This section presents a systematic approach to ensure quality in model-driven development processes. It takes a different approach to drive the selection of transformations, which is to empirically validate the selection of alternative transformations through controlled experiments. The rationale of this approach is to be able to automatically select the alternative transformation that an experienced software developer would select if the transformation process were manually applied.

In order to operationalize this approach, we propose the use of *quality attributes* to drive the selection of the most appropriate alternative transformation that contributes to the improvement of the target model according to a given quality

attribute. A quality attribute is a measurable physical or abstract property of an entity (i.e., a conceptual model) (ISO, 2001).

Currently, our controlled experiments are oriented to empirically validating the selection of the alternative transformation that maximizes the 'understandability' quality attribute. Fig. 1 presents an overview of our quality-driven model transformation approach.

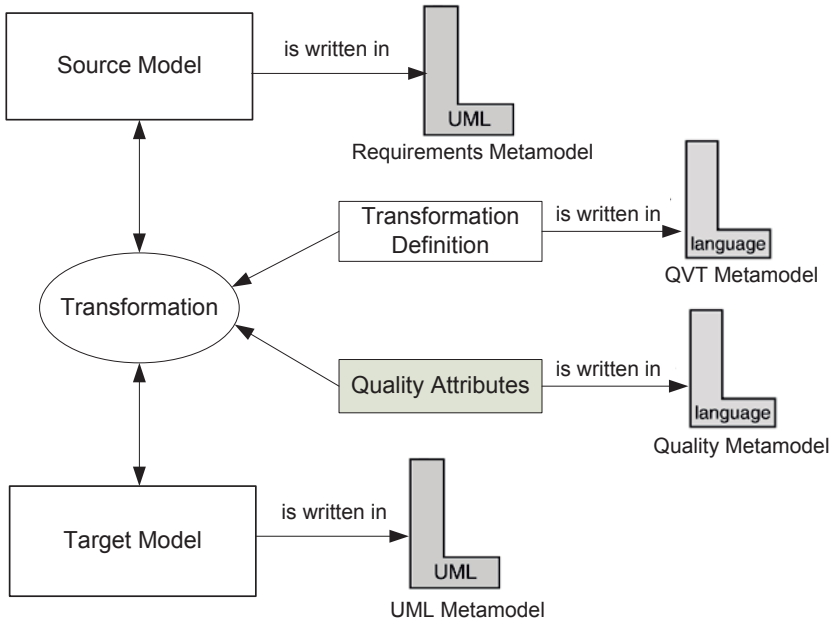


Fig. 1. Quality-driven model transformation approach

According to Fig. 1, a *transformation* is executed taking a *transformation definition* as input. A transformation definition contains transformation rules that relate constructs in the source model to constructs in the target model. These rules can be represented using the Query-View-Transformations (QVT) language proposed by the Object Management Group (OMG, 2005).

Another input for the transformation process is the definition of the *quality attributes* together with the corresponding empirical evidence gathered from the controlled experiments. This information will feed the transformation process with the criteria to choose the alternative transformation that maximize the selected quality attribute. Our final objective is to execute these transformations in a platform for Model Management called MOMENT (Boronat, Carsí & Ramos, 2005; 2006).

The following sections show a specific domain for applying our quality-driven model transformation approach using a Requirements Model as source model, a UML class diagram as target model, and "understandability" as the quality attribute to drive the transformations.

4. Transforming Requirements Models into UML Class Diagrams

The Requirements Model (Insfran, 2003) (Insfran, Pastor & Wieringa, 2002) defines the structures and the process followed to capture the software requirements. It is composed of a *Functions Refinement Tree (FRT)* to specify the hierarchical decomposition of the system, a *Use Case Model* to specify the system communication and functionality, and *Sequence Diagrams* to specify the required object-interactions that are necessary to realize each Use Case. Consequently, as only functional software requirements are gathered (business requirements are excluded), the Requirements Model can be placed at the PIM level. The Requirements Model is supported by a Requirements Engineering Tool[2] (RETO).

Following a MDA strategy of model transformation, once the Requirements Model has been specified, a conceptual model including a UML class diagram can be obtained by applying a set of transformation rules from a Transformation Rules Catalog[3] (Insfran, 2003). These transformations establish traceability relationships between the Requirements Model and the UML class diagrams.

According to the MOF terminology, the Requirements Model and the UML class diagram are located in the M1 level and their metamodels are located in the M2 level. The definition of a transformation is performed at the M2 level and implies that "*a certain structural pattern is identified in the source model (Requirements model), which corresponds to a valid structure in the target model (UML class diagram)*".

Fig. 2 describes a simplified traceability relationship map to go from the set of specified requirements to specific elements in the conceptual schema. These traceability relationships may be simple (*one-to-one* relationships). For example, the generation of classes for the UML class diagram is a process that is based on the analysis of participating actors and classes in all the Sequence Diagrams. It includes the application of the following Transformation Rules (TR), stated here in natural language:

- **TR 1.** For every distinct actor class participating in any Sequence Diagram, a class will be generated in the UML class diagram.
- **TR 2.** For every distinct class participating in any Sequence Diagram, a class will be generated in the UML class diagram.
- **TR 3.** The boundary classes (usually called Interface or System) in Sequence Diagrams will not have an explicit representation in the UML class diagram.

[2] RETO web site: http://reto.dsic.upv.es

[3] The Transformation Rules Catalog can be found in http://www.dsic.upv.es/~einsfran/thesis

However, the traceability relationships can also be *many-to-many* relationships. This is due to the variability of the transformations, which allows multiple possible representations in the UML class diagram that satisfy a given requirement pattern identified in the Requirements Model. If this occurs, a single alternative mapping must be properly selected according to some predefined quality attribute.

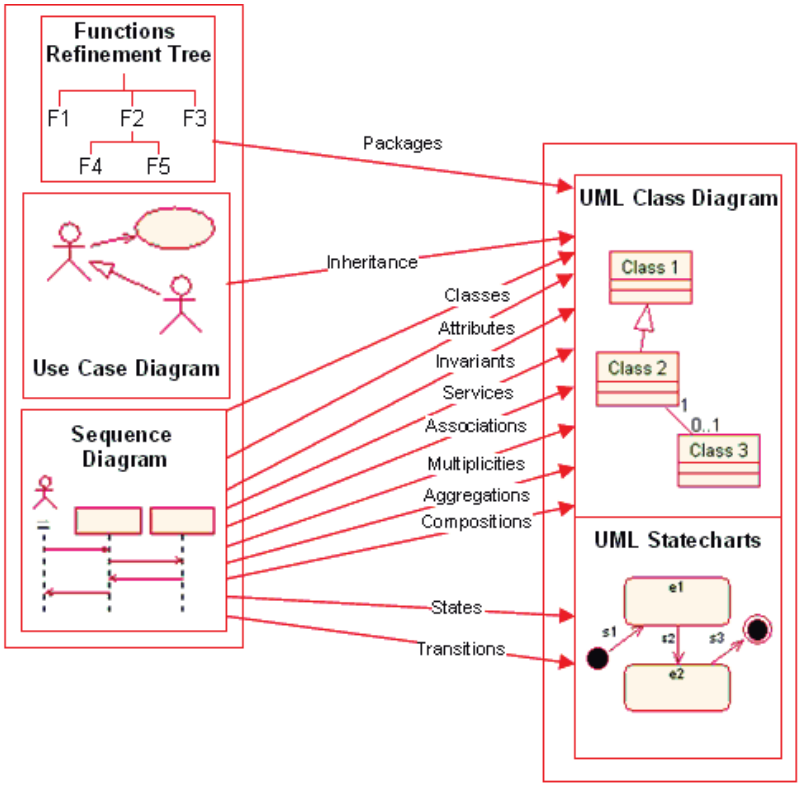


Fig. 2. Traceability from Requirements to Conceptual Models

Subsection 4.1 and 4.2 briefly introduce the requirements and the UML class diagram metamodels. The remainder of the section focus on transformations that have multiple valid representations in the UML class diagram that satisfy a given requirement pattern.

4.1 The Requirements Metamodel

Metamodeling is a key concept of the MDA paradigm and is used in Software Engineering (SE) to describe the basic abstractions that define the models and their relationships. A metamodel can be viewed as a class model whose classes and associations encode the concepts of the model and the relationships among them. The Meta Object Facility (MOF) (OMG, 2004) provides a framework for defining a metamodel and querying and manipulating the resulting models.

Fig. 3 shows an excerpt of the relevant parts of the Requirements Metamodel used as source in the transformation process. The *Use Case* class represents the functions of the system. Each Use Case is specified in detail by means of one or more Sequence Diagrams. Sequence Diagrams are composed mainly of *Entities* and *Messages*. We distinguish three types of Entities when describing a Sequence Diagram: Actor, Interface and Class. *Actor* represents the users of the Use Case (and may or may not be a class); *Interface* represents the boundary among the actors and the internal classes of the system; *Class* represents the different entity classes that participate in the realization of the Use Case.

Finally, in order to characterize the different nature of interaction between objects, we identify four types of messages: *Signal*, *Service*, *Query*, and *Connect*. Signal messages represent the interaction between actors and the interface. Service messages represent object interactions with the purpose of modifying the system (creation, deletion or update). Query messages represent object interactions to query the state of an object or a set of objects. Connect messages represent object interactions to establish a relationship between them.

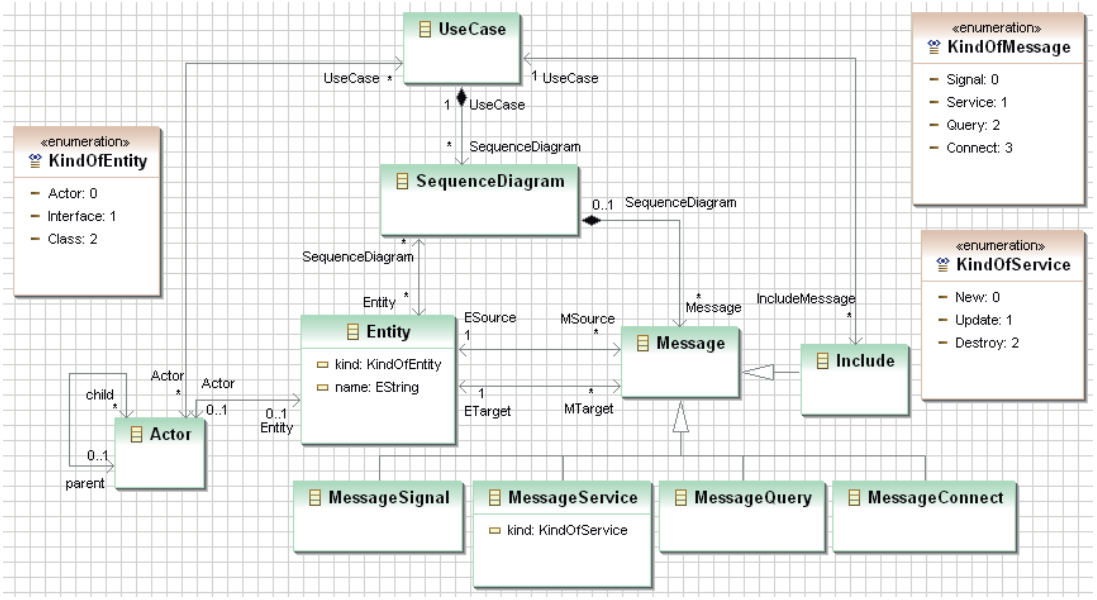


Fig. 3. Requirements Metamodel

4.2 The UML Class Diagram Metamodel

Once the source metamodel is defined, the UML target metamodel (OMG, 2006) must also be defined. At least three alternatives are possible:

- To use the UML2 metamodel directly. This has the advantage that the result can be used by all the tools that use this metamodel. However, the problem

is the size of the metamodel and its complexity. The use of the UML2 metamodel makes transformation rules difficult to specify and understand.

- To use the Ecore metamodel. This has the advantage that many tools directly use this metamodel, and it is also very well integrated in the Eclipse environment (www.eclipse.org). However, we could not represent two of the three types of relationships (*association class* and *aggregation*) that we needed to generate in this metamodel.
- To use the class diagram metamodel defined in the MOF QVT Final Adopted Specification (OMG, 2005). This metamodel is well known, simple, and it has the advantage that it can specify almost all the characteristics that are needed.

Finally, we decided to use a modified version of the class diagram of the MOF QVT specification, which we refer to as UMLite.

Fig. 4 shows the modified UMLite metamodel. The main part of the metamodel is the same as the metamodel defined in the QVT specification (OMG, 2005). A *Package* is formed by a set of *PackageElements*. Usually, an information system is formed by a set of *Packages*. A *PackageElement* can be a *Classifier* or a *Relationship*. *Classifier* is the generic name given to everything that can have attributes and operations. *PrimitiveDataTypes* and *Classes* are both *Classifiers*. The class *PrimitiveDataType* defines the Abstract Data Types used in the definition of a system. Typical *PrimitiveDataTypes* are integers, doubles, strings, and so on. Instances of the *Class* class will belong to a specific *Package*. A *Class* is formed by a set of *Attributes*. Each one of the *Attributes* has a name inherited from *UMLModelElement* (in fact, everything has a name because every class inherits from the *UMLModelElement* class) and its type must be a *Classifier* that was previously defined. The IS-A relationship between classes is maintained with the reflexive association relationship defined in the *Class* class. The *Relationship* class defines the relationships that can exist between two classes (the source and the destination classes).

In order to be able to define the characteristics of relationships between classes, two modifications have been added to the metamodel:

- An attribute named *kind* in the *Relationship* class to express the kind of relationship between two classes (association, aggregation, or composition).
- A new relationship between the *Relationship* and *Class* classes to express that a relationship has an association class.

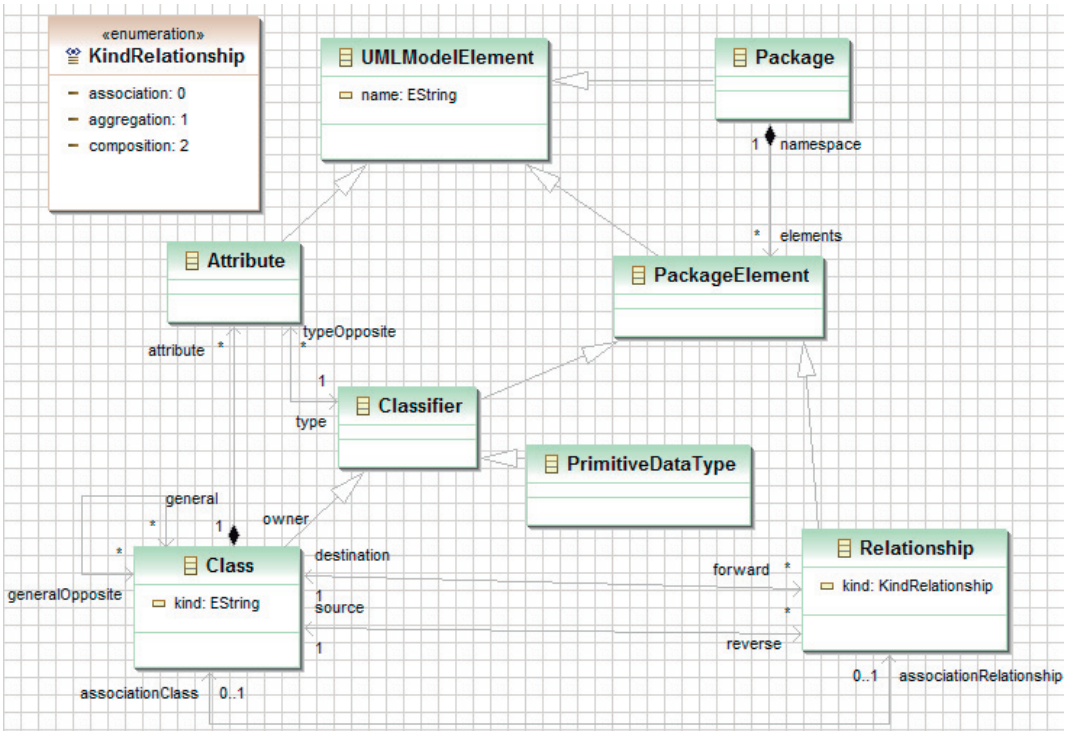


Fig. 4. The UMLite Metamodel

Even though there are no tools that use UMLite as their metamodel, it is still useful. Since the main concepts of UMLite are almost the same as the concepts in Ecore and UML2, they can be easily transformed to these metamodels.

4.3 Defining Alternative Transformations using QVT

This subsection shows how some alternative transformations for a requirement specification generate different UML class diagrams. Although not all the possibilities are fully explained due to space limitations, it is possible to see that, given a requirement specification, a set of conceptual model solutions can be identified. The example used to illustrate these alternative transformations is taken from the specification of a Car Rental system.

A Sequence Diagram is used to specify the necessary object interactions to realize the Use Case *Create Insurance* that is initiated by the *Administrator* actor. This Use Case represents the creation of a car *Insurance* policy that must be bought from an *Insurance Company* and assigned to the *Car* before using the car for rentals. Fig. 5 shows the Sequence Diagram for the Use Case *Create Insurance*. After introducing the necessary data and checking the existence of the corresponding car, a new *Insurance* object is created (messages 1 to 5). In addition, an *Insurance Company* object and a *Car* object must be connected to the new created Insurance policy (messages 6 and 7).

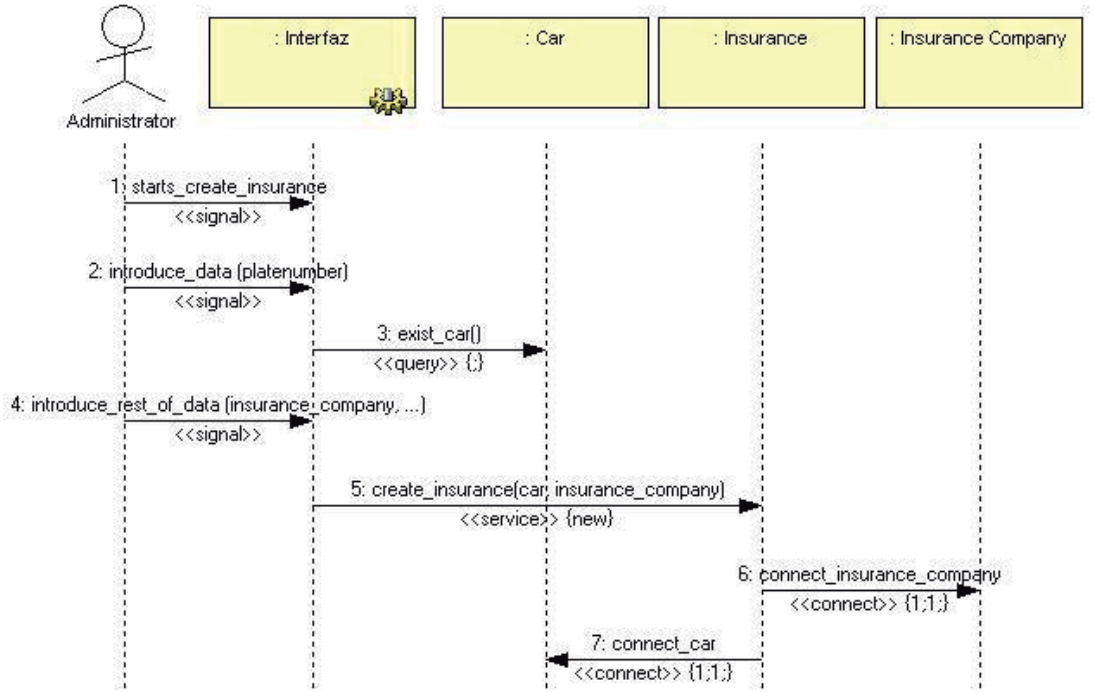


Fig. 5. Sequence Diagram showing the required interactions for the Use Case *Create Insurance*

In our approach, this information is used to transform the requirements specification into a UML class diagram[4] following an MDA approach. It is important to remark that, for an automated transformation process to be considered useful, it must make decisions about which transformations are more suitable to produce the expected result by the analyst or a result that maximizes a quality attribute (in our case the *understandability* attribute).

The analysis of the requirement specified as object interactions shown in Fig. 5 indicates that the messages 6 and 7 satisfy the transformation rule TR 15:

- **TR15:** For every message between two classes labeled with the stereotype «connect», THEN an *association relationship* will be generated.

As a result, this transformation rule is applied twice. This means that two association relationships are established in the target model: one from *Insurance* to *Car* and another from *Insurance* to *InsuranceCompany* (Fig. 6a, **association**). An association relationship indicates a connection (link) between two classes.

[4] The generated UML class diagram can later be modified according to traceability rules (*strong* and *weak* traceability), which are explained in (Insfran, 2003).

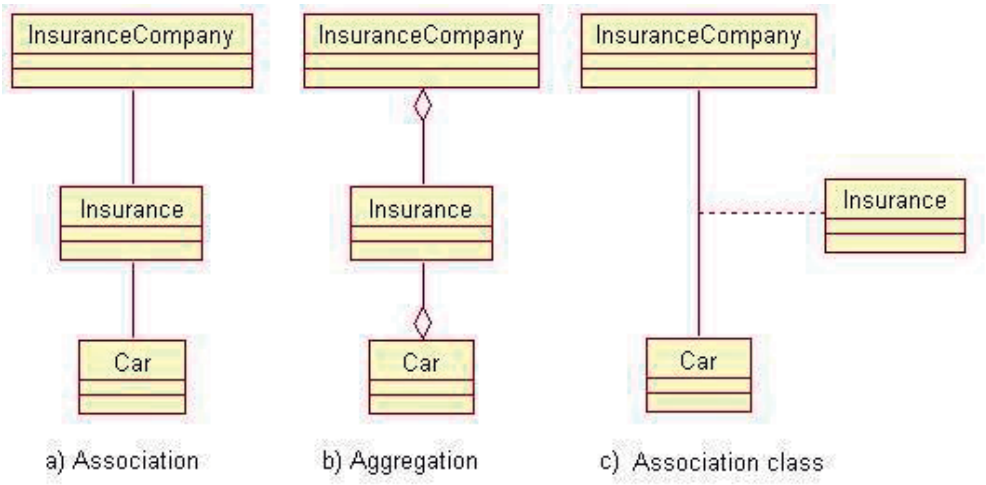


Fig. 6. Partial UML class diagram for the analyzed Sequence Diagram

Alternatively, there are other interpretations to the object interactions shown in Fig. 5. As a second alternative, the new created object *Insurance* can be represented as a component of both the *Insurance Company* compound class and the *Car* compound class (Fig. 6b, **aggregation**). This is because an aggregation is a special form of an association, specifying a whole-part relationship between two objects. This means that there is a connection between classes but also implies an additional semantics which indicates that an object 'is made up of other objects'. We are aware that not always an association relationship can be represented as an aggregation relationship, as this decision depends on the problem domain. However, in our example, both relationships can be applied as an Insurance "could be *related to* an insurance company and a car" or "be *part of* an insurance company and a car".

A third alternative is to consider the *Insurance* as an association class related to the association relationship between *Insurance Company* and *Car* (Fig. 6c, **association class**). This means that when an instance of an *InsuranceCompany* class is associated with an instance of a *Car* class, there will also be an instance of an *Insurance* class.

These three types of relationships can be alternative representations for the object interactions shown in Figure 5. In general, an association class can always be replaced by two association or aggregation relationships. Consequently, we have identified three types of structural relationships to represent object interactions (there may also be other representations). This implies the application of different transformation rules from the Transformation Rules Catalog (Insfran, 2003) to produce these relationships. Table 2 summarizes the alternative structural relationships and the transformation rules that can be applied to produce them.

Table 2. Three alternative structural relationships and their corresponding transformation rules

Alternatives	Transformation Rules
A1 (association)	**TR 14.** For every message between two classes labeled with the stereotype «service/new» where both classes are distinct from the "interface" class THEN an association relationship between these classes will be generated.
	TR 15. For every message between two classes labeled with the stereotype «connect», THEN an association relationship between these classes will be generated.
	TR 16. For every message with the stereotype «service/new» or «connect» where classes using role names appears THEN an association relationship between these classes will be generated using these role names on the ends of the relationship.
A2 (aggregation)	**TR 28.** For every message with the stereotype «service/new» between two classes A and B, which are distinct from the "interface" class, THEN an aggregation relationship between these classes will be generated.
A3 (association class)	**TR 39.** For every message with the stereotype «service/new» from the class A to the class B, if there exist two messages with the stereotype «connect» starting from the class B to the classes C and D respectively, THEN a new association class will be generated (called B) AND also an association relationship between C and D related to the new association class B will be generated.
	TR 40. For every message with the stereotype «connect» from the class A to the class B, if there exist a message with the stereotype «service/new» starting from the class A or B to the class C THEN a new association class will be generated (called C) AND also, an association relationship between A and B related to the new association class C will be generated..

All the transformation rules in the catalog are being specified using the declarative QVT relations language (OMG, 2005). Fig. 7 shows the specification of the transformation rule TR15 in QVT.

4.4 Executing the Alternative Transformations in MOMENT

MOMENT (Boronat, Carsí & Ramos, 2005) is a framework for model management that is fully integrated in the Eclipse environment. MOMENT combines the best features of Maude and Eclipse. First, it uses Maude (Clavel et al., 2005) as a backend. Maude is a reflective language and system supporting equational and rewriting logic specification. Maude has been used as a rapid prototyping environment to develop MOMENT using some of its properties: pattern matching, parameterization, and reflection. Second, Eclipse Modeling Language (EMF) is an industrial standard that includes a metamodel (Ecore) to define, modify, and serialize models with a very efficient reflexive API.

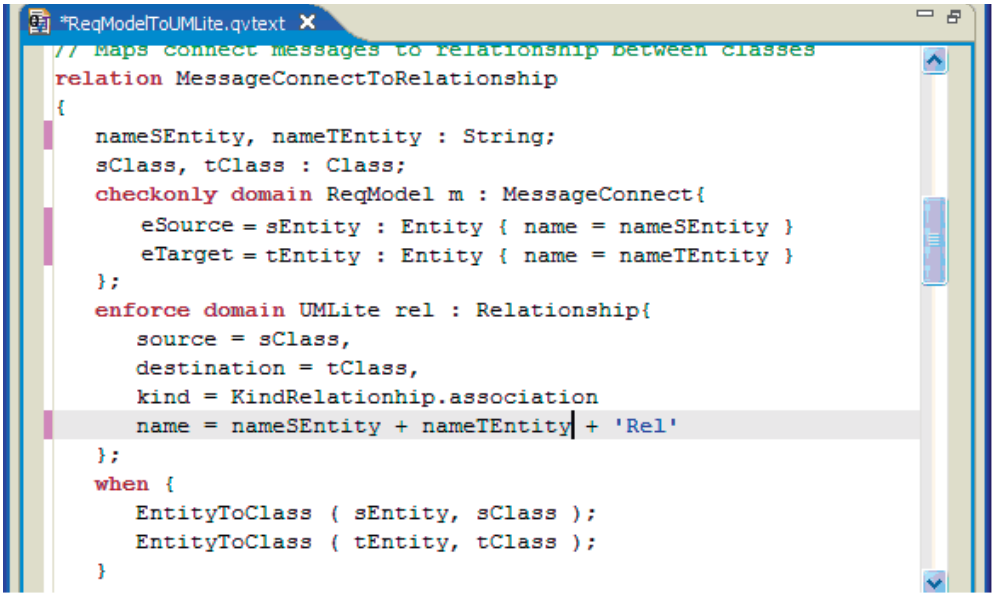


Fig. 7. TR15 specified using QVT-Relations

To use MOMENT to apply transformations (Boronat, Carsí & Ramos, 2006), the source metamodel (the Requirements Metamodel) and the target metamodel (the UMLite class diagram Metamodel) must be defined and registered as Ecore models. MOMENT is used to define the QVT-Relations transformation *ReqModelToUMLite*. This transformation is composed of a set of rules that defines how to transform information belonging to a model that conforms to the Requirements Metamodel into a model that conforms to the UMLite class diagram Metamodel.

Finally, a configuration for the MODELGEN operator is defined. This configuration is composed of the *ReqModelToUMLite* transformation, a source model that is defined with the RETO tool, and the name of the target UML model. Additionally, a traceability model is generated to relate the elements of the source model with the elements of the target model.

Fig. 8 shows MOMENT integrated in the Eclipse environment. On the left side, the Package Explorer shows all the files related to the transformation: *ModelGen.mop* is the model that defines the generic operator MODELGEN; *reqModel.ecore* is the model that defines the Requirements Metamodel used as the source in the transformation; *UMLite.ecore* is the metamodel used as the target in the transformation; *rentacar.reqmodel* is a model that defines the rentacar system (instance of the metamodel reqModel); *rentacar.umlite* is the model (instance of UMLite metamodel) resulting from the application of the transformation to rentacar.reqmodel; *rentacar.traceabilitymodel* is the traceability model that maps elements of the source and target models; *ReqModelToUMLite.qvtext* is the transformation expressed in QVT-Relations. The top of the figure shows part of the UMLite metamodel. The bottom of the

figure shows a part of the ReqModelToUMLite transformation inside the QVT text editor.

An additional transformation called *UMLite2UML2* was defined to transform UMLite class diagram models into UML2 class diagram models (see section 3.2).

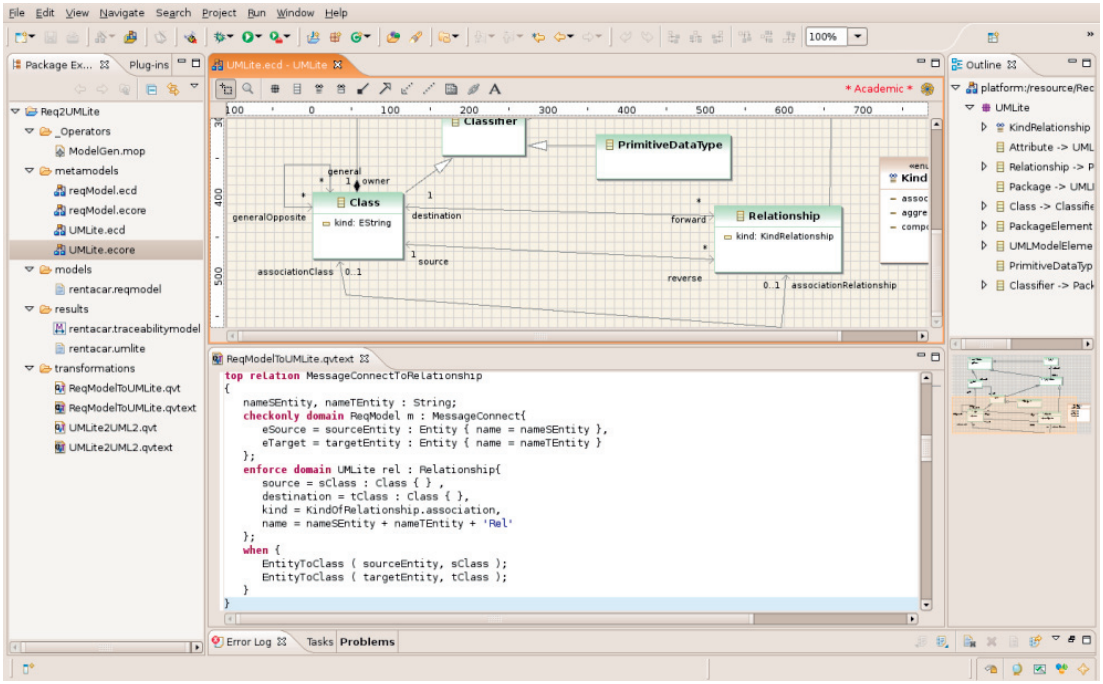


Fig. 8. MOMENT environment

Once a transformation has been executed, a traceability model is generated. The traceability model relates the elements of both models. There is a special view designed for this information. This view allows the analyst to see the transformation rules that have been executed and what the results are.

Fig. 9 shows the traceability model generated after executing the *reqModel2UMLite* transformation. The first column shows the domain model, which is the *rentacar* requirements model. The third column shows the range model, which is the generated *rentacar* UMLite model. Finally, the second column shows the traceability links (mappings) that relates the elements of the domain and range models. A traceability link, which is the result of applying the transformation rule *EntityToClass* is highlighted. This traceability link relates the *Insurance* entity with the *Insurance* class.

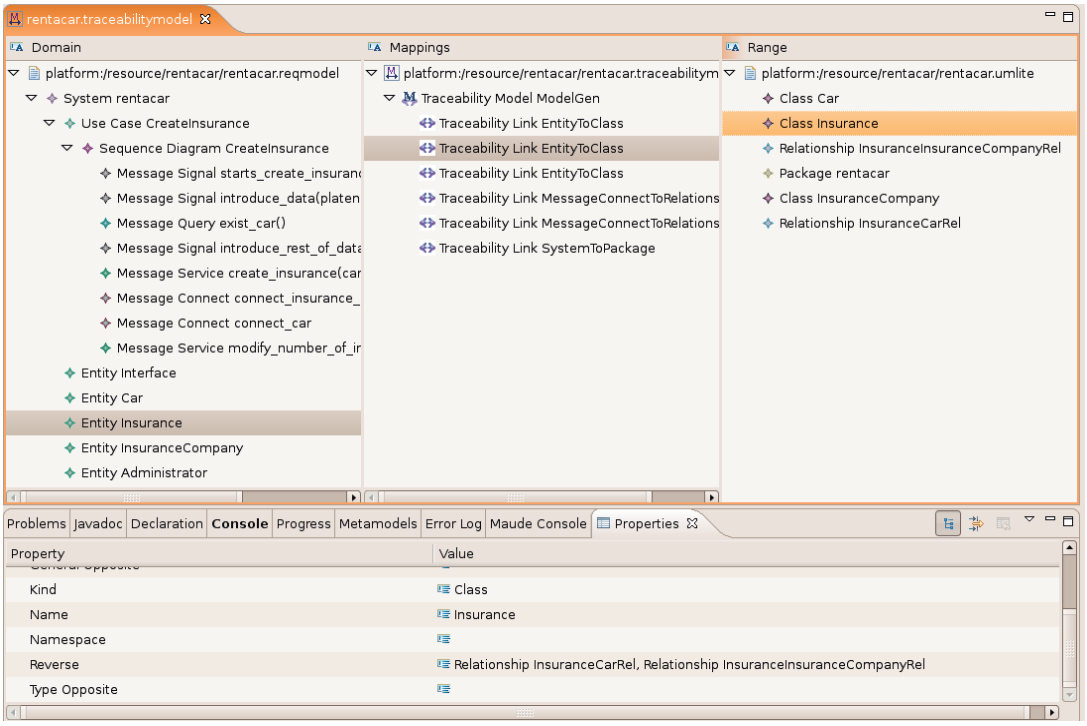


Fig. 9. Traceability model generated by applying the *ReqModelToUMLite* transformation.

5. Experiment Description

This section presents a description of the experimental process that was followed to select the best alternative transformation. The process is based on the experimental frameworks proposed by (Wohlin et al., 2000; Juristo & Moreno, 2001). This process is composed of the following activities: definition, planning, operation, and analysis and interpretation.

5.1 Definition

The main goal of this experiment is to determine which of the transformation rules for structural relationships between classes introduced in section 4.3 (A1, A2, A3, and A4) obtained the easiest to understand UML class diagram. Therefore, using the GQM (Basili & Rombach, 1988) template for goal definition, the goal of our experiment is defined as follows:

Analyze	*Alternative transformation rules for structural relationships between classes (A1, A2, and A3)*
For the purpose of	*Evaluating*
With respect to	*the Understandability of the obtained UML class diagrams*
From the point of view of	*the researchers*

In the context of	*Undergraduate students at the Department of Information Systems and Computation at the Valencia University of Technology*

5.2 Planning

The next step is planning. The definition determines *why* the experiment is conducted, while the planning prepares *how* the experiment is to be conducted. The main characteristics of the planning phase are the following:

Subjects. The participants were 39 fourth-year students in Computer Science at the Valencia University of Technology, who were taking part in the second Software Engineering course. We took a "convenience sample" (i.e. all the students in the class). The subjects had six months of experience in modeling with UML and three years of experience in the OO paradigm. The subjects were encouraged to participate by offering them an extra point in the final grade for performing the required tasks correctly.

Variable selection. The independent variables were the transformation rules for structural relationships between classes (i.e., A1, A2, and A3). The dependent variable was understandability.

Experimental material and tasks. The experimental material and tasks consisted of:

- 9 Sequence Diagrams from three different case studies, with 3 UML class diagrams each. These were obtained by applying the alternative transformation rules. An example of the experimental material is shown in Appendix A. The rest of experimental material is available at: www.dsic.upv.es/~einsfran/experiment.
- Each Sequence Diagram has a questionnaire attached consisting of 6 Yes/No questions to test the subjects' understanding of the Sequence Diagrams. The effectiveness of the subjects in answering the questionnaires (number of correct answers/number answers) was used as a criterion to exclude those observations that did not fulfill a minimum level of quality. Observations with a value less than or equal to 0.5 were excluded. If the subjects did not understand the Sequence Diagrams, their questionnaires were excluded.
- Each of the three UML class diagrams had a questionnaire attached (with 6 questions) for assessing which alternative UML class diagram was better understood by the subjects. In addition, the subjects had to write down the starting and ending times for completing the questionnaires. We obtained three measures for understandability from this understanding task:
  - *Understandability Time,* which reflects the time, in seconds, that the subjects spent answering each questionnaire (calculated by the difference between the ending time and the starting time). Each subject

completed 3 questionnaires detailing 3 alternatives (A1, A2, and A3). Three understandability time measures (A1Time, A2Time, and A3Time) were obtained.
  - *Effectiveness*, which reflects the correctness of the answers (calculated by dividing the number of correct answers by the number of answers). Three understandability effectiveness measures (A2Effec, A2Effec, and A3Effec) were obtained.
  - *Efficiency,* which reflects the correctness of the answers by time (calculated by dividing the number of correct answers by the understandability time). Three measures for understandability efficiency (A2Effic, A2Effic, and A3Effic) were obtained.
- The final task of each test consisted of asking the subjects which of the three alternative UML class diagrams best reflected the problem modeled in the Sequence Diagram. In this way, we obtained a subjective measure (*Alternative Selected*) based on the subjects' perception.

Hypothesis formulation. The following hypotheses were formulated:

- **$H1_0$**: The use of different alternative transformations (A1, A2, and A3) does not affect the Understandability Time (A1Time, A2Time, and A3Time). $H1_1 = \neg H1_0$
- **$H2_0$**: The use of different alternative transformations (A1, A2, and A3) does not affect the Understandability Effectiveness (A1Effec, A2Effec, and A3Effec). $H2_1 = \neg H2_0$
- **$H3_0$**: The use of different alternative transformations (A1, A2, and A3) does not affect the Understandability Efficiency (A1Effic, A2Effic, and A3Effic). $H3_1 = \neg H3_0$
- **$H4_0$**: There is no correlation between the Alternative Selected and the means of objective Understandability variables (Understandability Time Effectiveness, and Efficiency). $H4_1 = \neg H4_0$

5.3 Operation

The experiment started with an introductory session in which the main concepts of the Requirements Model (e.g., the notation of Sequence Diagrams) were reviewed. The goal of the experiment was not disclosed to the subjects. Then, the subjects were shown an example of the experimental material, which was similar to what they would be using during the execution of the experiment.

Each subject was given all the experimental material, including nine tests (balanced within-subject design). The diagrams were assigned in different order to limit learning effects. The alternatives were also organized in a different order across subjects in order not to favor one alternative over another. In total, eighteen types of tests were prepared. The subjects were instructed how to

develop the experimental tasks and they had a maximum of two hours to complete all the tasks.

5.4 Data Analysis and Interpretation

After the experiment took place, we collected the experiment data. It consisted of a table of 351 rows (9 diagrams x 39 subjects) and 9 columns (A1Time, A2Time, A3Time, A1Effec, A2Effec, A3Effec, A1Effic, A2Effec, A3Effec). We then performed a "data cleaning", excluding the observations that were not complete because the subjects had not written down the time or because the subjects did not selected the best alternative. Since all the questions in each questionnaire were complete, the completeness of the performed tasks was guaranteed. We also excluded the observations that had a value of effectiveness of 50% or less for each Sequence Diagram. The final data for testing the hypotheses were 325 observations.

The following statistical analyses were performed to analyze the data:

- A descriptive study was done to characterize the dependent variables.
- Hypotheses H1, H2, and H3 were tested using an ANOVA test with repeated measures.
- Hypothesis H4 was tested using the Spearman correlation coefficient.

We used SPSS (SPSS, 2002) to carry out the data analyses presented in this study.

5.4.1 Descriptive statistics

The descriptive study was performed by first analyzing the variable *Alternative Selected* and then analyzing the measures of *Understandability Time*, *Effectiveness,* and *Efficiency*.

From Table 3 (which shows the frequency of each type of alternative (A1, A2, and A3) for each diagram) and Fig. 10 (which shows the percentages of selection for each type of alternative) we can infer the following:

- A1 is the alternative transformation that was most selected by the subjects, i.e., the subjects believed that the use of associations allowed to obtain the best UML class diagram (the easiest to understand).
- A3 is the alternative transformation that was least selected by the subjects, i.e., the subjects believed that it was the least appropriate alternative transformation.

Table 3. Frequency of transformation alternatives per diagram

Diagrams/ Alternatives	D1	D2	D3	D4	D5	D6	D7	D8	D9	Total
A1	6	12	13	25	18	14	16	18	7	129
A2	8	4	7	8	9	12	6	5	13	72
A3	19	20	10	5	11	12	15	14	18	124
Total	33	36	30	38	38	38	37	37	38	325

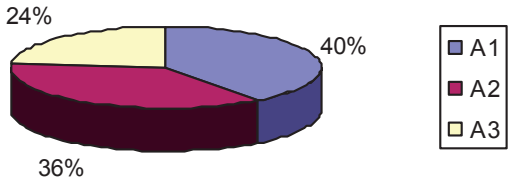


Fig. 10. Percentages for Alternative Selected

The descriptive statistics for the *Understandability Time*, *Effectiveness*, and *Efficiency* are shown in Table 4. They are ranked in ascendant order by the value of the mean.

Table 4. Descriptive statistics for the measures for Understandability Time, Effectiveness and Efficiency.

	Min.	Max.	Mean	St. Dev.
A2TIME	23	449	93.4338	57.9701
A1TIME	12	611	101.9046	79.1776
A3TIME	15	734	106.5077	72.2032
A2EFFEC	0.167	1	0.8701	0.1528
A3EFFEC	0.167	1	0.8844	0.1821
A1EFFEC	0.167	1	0.9111	0.1548
A3EFFIC	0.006	0.4	0.0677	0.0419
A2EFFIC	0.011	0.24	0.0757	0.0436
A1EFFIC	0.005	0.5	0.0803	0.0531

Table 4 reveals that, on average, the subjects spent less time performing the tasks related to alternative A2. However, the difference with the other tasks was not very significant (approximately 8 seconds for A1 and A3). The subjects

were more effective and efficient performing the tasks related to alternative A1; but the difference in effectiveness with the other alternatives was not very significant.
In summary, the descriptive statistics show a slight tendency in favor of A1, which is the transformation based on associations.

5.4.2 Testing Hypotheses

To test the hypotheses H_1, H_2, and H_3, we carried out an ANOVA for repeated measures. The results show hypotheses $H1_0$, $H2_0$, and $H3_0$ can be rejected (with a significance level = 0.05). This means that each alternative transformation really does affect the *Understandability Time*, *Effectiveness,* and *Efficiency*.

Moreover, we compared the means for each measure by pairs of alternatives. There was a significant difference between the following pairs:

- The pairs A1-A2, A1-A3 in the values of the *Understandability Time*.
- The pairs A1-A2, A1-A3 and A2-A3, in the *Understandability Effectiveness*.
- The pairs A1-A3, A1-A3 and A2-A3, in terms of the *Understandability Efficiency*.

This comparison shows that there is a significant difference between A1 (related to associations) and the other alternatives (A2 and A3).

To test hypothesis H_4, we carried out a correlation analysis using the Spearman Correlation, separately per each diagram. We did not find any correlation between the subjective measure (*Alternative Selected*) and the mean of the objective measures (*Understandability Time, Effectiveness,* and *Efficiency*). This reveals that the use of an alternative transformation is not dependent on how effective or efficient the subjects are (i.e., the performance of the subjects did not affect their perception).

In summary, the main findings of the experimentation show that there exists a slight tendency in favor of using associations. In other words, the subjects are a slightly more effective and efficient when performing tasks related to association relationships (instead of aggregations or association classes). Assuming that the three alternatives are alternatives, this indicates that transformations related to association relationships are the most appropriate when the understandability quality attribute is selected.

5.5 Threats to validity

This section discusses several issues that can affect the validity of the empirical study and how we attempted to alleviate them.

In order to control the risk that the variation due to individual differences is larger than due to the treatment, we selected a homogeneous group of subjects. In addition, to attempt to control the internal validity of the study, the following issues were considered:

- *Differences among subjects.* Using a within-subjects design, error variance due to differences among subjects was reduced. In addition, we randomly assigned the tests to the subjects in different order. This procedure cancels out a possible learning effect (due to similarities in the treatments) and a confounding effect (due to the order in which the alternatives were presented).
- *Knowledge of the universe of discourse.* We used the same requirement specification document for all subjects. It specifies the requirements of a Car Rental System for a company. This is a well-known universe of discourse.
- *Fatigue effects.* On average, each subject took two hours to solve the experimental tests, so fatigue was not very relevant.
- *Persistence effects.* In order to avoid persistence effects, the experiment was carried out by subjects who had never done a similar experiment.
- *Subject motivation.* We motivated students to participate in the experiment by offering them an extra point in the final grade of the course.

One limitation to the external validity of this study is the fact that the three alternative transformation rules cannot be applied simultaneously to all modeling situations. For instance, to establish an association class relationship (A4), at least one «service/new» message and two «connect» messages are needed in the source model. The goal of this experimentation was to gather empirical evidence for the specific case when the three alternative transformations can be applied to obtain a relationship between classes. We are aware that, more alternatives may be possible to represent structural relationships between classes. More experimentation is needed to validate these other combinations.

Another limitation is the use of only students as participants. In general, our students have no working experience in conceptual modeling. The use of student participants may present a threat to the study's external validity. However, the students who participated in the experiment were fourth-year students in Computer Science. Therefore, they can be considered as representative of novice users of conceptual modeling approaches. To increase external validity, the current study needs to be replicated using experienced practitioners from the industrial sector who are experienced in UML and/or students with higher levels of training in order to confirm our results.

6. Conclusion

This chapter has presented an approach for quality-driven model transformations. Specifically, it described a controlled experiment to investigate the selection of alternative QVT transformations to obtain UML class diagrams from a Requirements Model. The goal of the experiment was to gather empirical evidence about which alternative transformation produces the UML class diagram that is easiest to understand.

The results show that there is a slight tendency to favor the use of association relationships when the three alternatives can be applied. This indicates that transformations related to association relationships are the most appropriate when the understandability quality attribute is selected. A possible reason for this could be that this relationship has less semantic strength than the other kinds of relationships. When an aggregation relationship is chosen instead of an association relationship, analysts know that they are defining a *part-of* relationship. However, when an association class is chosen, the same relationship can be represented using two association or aggregation relationships.

These results provided first evidence about the understandability of a model obtained through a model transformation process. The study was conducted in the context of the UML class diagram that is the most-used specification for model-driven software development in industry. Although this evidence is specific to this domain, and in particular, to the relationships among classes, it should be generalized to other elements in a UML class diagram, other UML diagrams, and also to other domains. Therefore, more experimentation is needed to verify the generalizability of our approach.

The results that we have obtained through experimentation are promising. However, they must be considered as preliminary results. We plan to replicate this experiment with students from the University of Castilla-La Mancha in Spain and also with more experienced practitioners from the industrial sector in order to confirm these results. We believe that the level of experience in UML modeling can considerably influence the performance of the subjects.

7. Future Research Directions

Our literature review has shown that there are very few studies that deal with quality in model-driven development. As far as we know, there is no study encompassing the empirical validation of model transformations. The study of quality for model-driven development is of great relevance to software development organizations faced to the adoption of this technology in industry.

The empirical study presented in this chapter shows how model transformations can be empirically validated with regard to a given quality attribute. This work is part of a project on quality-driven model transformations whose goal is the

definition of a quality metamodel to drive the selection of model transformations according to multiple quality attributes. Therefore, our current research efforts are focused on the validation of the remaining transformations of the Transformation Rules Catalog (Insfran 2003). We also plan to study other quality attributes (i.e., efficiency, understandability, usability, modifiability) and possible conflicts that could arise when more than one quality attribute is chosen. Our ultimate goal is to build an empirically validated quality metamodel to drive the selection of model transformations in different domains (e.g., Bioinformatics and Data Warehouse). This quality metamodel will be fully integrated in the MOMENT environment.

There is an urgent need for more research studies of this type to complement and extend the current empirical study. Empirical evaluation of model transformations will help software developers assess the usefulness of different sets of transformations according to the quality of the resulting target model and/or transformation needs.

While several studies (including this one) have studied model transformations and its properties, it would be interesting to survey or interview domain-specific engineers and ascertain the importance of certain model transformations to define heuristics to drive the transformations. In our study, a heuristic could be the type of traceability (i.e., strong and weak) assigned to each transformation rule. This could be an additional criterion to be used during the transformation process.

Another area of future research needs to examine other quality perspectives (i.e., syntactic quality, semantic quality). Syntactic quality in the context of model-driven development is trivial due to all the models are compliant to their respective metamodel. However, assessing semantic quality will allow to verify which alternative transformation will produce a target model that is more correct and relevant to the problem domain.

Finally, the applicability or dependence of model transformations to the type of domain of the application being developed would be an interesting study.

References

Basili, V. & Rombach, H. (1988). The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6), 728-738.

Boronat, A., Carsí J.A., & Ramos I. (2006). *Algebraic Specification of a Model Transformation Engine*. Proceedings of the Fundamental Approaches to Software Engineering (FASE'06). ETAPS'06. Vienna, Austria, 262–277.

Boronat, A., Carsí, J.Á., Ramos, I. (2005). *MOMENT: a formal MOdel manageMENT tool.* School on Generative and Transformational Techniques in SE. Braga, Portugal.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., & Talcott, C. (2005). Maude 2.2 manual and examples, from http://maude.cs.uiuc.edu/maude2-manual

Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal,* 45(3), 621–645.

Genero, M., Manso, M., Visaggio, A., Canfora, G. & Piattini, M. (2007) Building measure-based prediction models for UML class diagram maintainability. *Empirical Software Engineering*, (to appear).

Genero, M., Moody, D., & Piattini, M. (2005). Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation. *Journal of Software Maintenance and Evolution: Research and Practice*, 17, 225-246.

Insfran, E. (2003). *A Requirements Engineering Approach for Object-Oriented Conceptual Modeling*, PhD Thesis, DSIC, Valencia University of Technology, Spain.

Insfran, E., Pastor, O. & Wieringa, R. (2002). Requirements Engineering-Based Conceptual Modelling. *Journal of Requirements Engineering*, 7 (2), 61–72, Springer-Verlag.

ISO, ISO/IEC 9126-1, (2001). Software Engineering – Product quality – Part 1: Quality model.

Ivkovic, I., & Kontogiannis, K. A. (2006). *Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations*, Proc. of the Conference on Software Maintenance and Reengineering (CSMR'06), 135–144.

Jurista, N. & Moreno, A. M. (2001). *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers.

Kerhervé, B., Nguyen, K. K., Gerbé, O., & Jaumard, B. A. (2006). Framework for Quality-Driven Delivery in Distributed Multimedia Systems, Proc. of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006), 195–205.

Kurtev, I. (2005). *Adaptability of Model Transformations*. PhD Thesis, University of Twente, The Nederlands.

Lindland, O. I., Sindre G., & Sølvberg A. (1994). Understanding quality in conceptual modeling. *IEEE Software*, 11(2), 42–49.

Markovic, S., & Baar, T. (2005). *Refactoring OCL annotated UML class diagrams*. In Proc. of the 8th Int. Conference on Model Driven Engineering Languages and Systems, 280–294.

Merilinna, J. (2005). *A Tool for Quality-Driven Architecture Model Transformation*. Espoo, VTT Electronics, VTT Publications.

OMG, (2006). OMG, UML 2.1 Unified Modeling Language™

OMG, (2005). OMG, MOF 2.0 Query/Views/Transformations Final Adopted Specification, Object Management Group, from http://www.omg.org/cgibin/apps/doc?ad/05-11-01.pdf

OMG, (2004). Meta Object Facility (MOF) 2.0 Core Specification, ptc/04-10-15.

OMG, (2003). MDA Guide, from http://www.omg.org/docs/omg/03-06-01.pdf. Version 1.0.1.

Otero, M. C., & Dolado, J. J. (2004). Evaluation of the Comprehension of the Dynamic Modeling in UML. *Information and Software Technology*, 46(1), 35-53.

Reinhartz-Berger, H. & Dori, D. (2005). OPM vs. UML—Experimenting with Comprehension and Construction of Web Application Models. *Empirical Software Engineering*, 10, 57–79.

Rottger S., & Zschaler, S. (2004). Model-Driven Development for Non-functional Properties: Refinement through Model Transformation, In LNCS Volume 3273, The Unified Modelling Language (UML) Conference, pp. 275–289.

Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE Software*, 20 (5), 19-25.

SPSS, SPSS 11.5, Syntax Reference Guide. 2002, SPSS Inc.: Chicago, USA.

Sottet, J. S., Calvary, G., & Favre, J. M. (2006). Mapping Model: A First Step to Ensure Usability for sustaining User Interface Plasticity, In: Proc. of the MODELS 2006 Workshop on Model Driven Development of Advanced User Interfaces.

Wohlin C., Runeson P., Höst M., Ohlson M., Regnell B. and Wesslén A. (2000). *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers.

Zou, Y., Kontogiannis, K. (2003). Quality Driven Transformation Framework for OO Migration. In. *Proc. 2nd ASERC Workshop on Software Architecture*, Banff, Canada, pp. 18–24.

Additional Reading

Endres, A. & Rombach, D. (2003). *A Handbook of Software and Systems Engineering: Empirical Observations*, Laws and Theories. Addison Wesley.

Frankel, D. S. (2003) *Model driven Architecture: Applying MDA to Enterprise Computing*, Wiley.

Juristo, N. & Moreno, A. M. (2001). *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers.

Maxwell, K. (2002). *Applied Statistics for Software Managers*. Software Quality Institute Series. Prentice Hall.

Stahl, T., Voelter, M., & Czarnecki, K. (2006) *Model-Driven Software Development: Technology, Engineering, Management, Wiley.*

Unhelkar, B. (2005) *Verification and Validation for Quality of UML 2.0 Models*, Wiley-Interscience.

Wohlin C., Runeson P., Höst M., Ohlson M., Regnell B. and Wesslén A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.

Appendix A. An Example of the Experimental Material

TEST R1

The following Sequence Diagram represents the creation of a *Car* for a car rental company. All the cars of the company have an assigned *Rate.* In addition, they must have an *Insurance* policy from an *Insurance Company*.

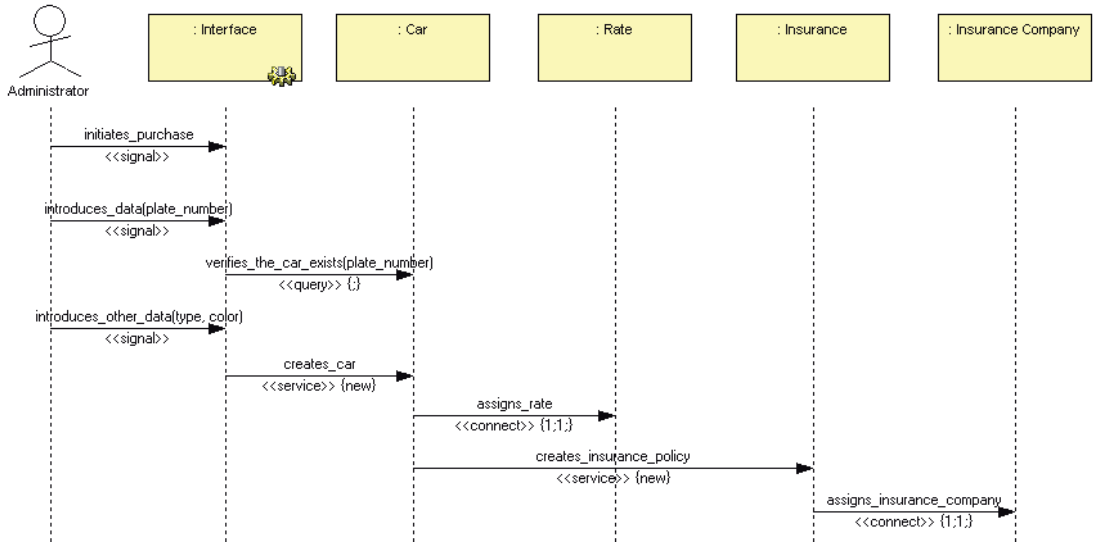


Fig. 11. Sequence diagram "creation of a car"

SECTION A: Understandability

WRITE DOWN THE CURRENT TIME (HH: MM: SS) _____

Answer the following Yes/No questions:

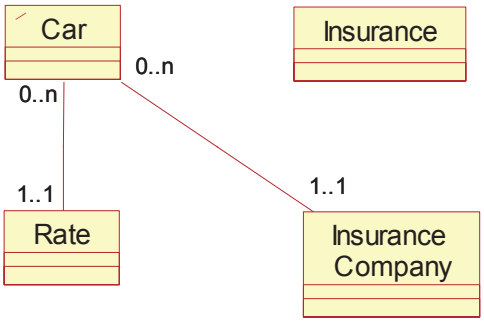
1. Is it possible in this scenario to create several *Cars*? _____
2. Can a *Car* have several *Insurance* policies in addition to the obligatory one for accidents?; for example, a policy for theft _____
3. Are there four classes in this Sequence Diagram? _____
4. If an appropriate *Rate* for a *Car* does not exist, can a new type of *Rate* be created and then assigned to the *Car*? _____
5. Can a *Car* be created without an obligatory *Insurance* policy? _____
6. Is it possible to associate an *Insurance* policy of another *Car* to the *Car* being created? _____

WRITE DOWN THE CURRENT TIME (HH: MM: SS) _____

Section B: Alternatives of Representation

Note: As an example of the three alternatives obtained from the Sequence Diagram shown in Fig. 11, we include the one based on the association class relationship.

Alternative 1:



WRITE DOWN THE CURRENT TIME (HH: MM: SS) _____

Answer the following Yes/No questions:

1. Does the *Insurance* policy exist because there is a relationship between the *Car* and *Insurance Company* classes? _____
2. Can an *Insurance* policy be related to an *Insurance Company* without being related to the *Car*? _____
3. If the relationship between the *Car* and the *Insurance Company* is destroyed, can the *Insurance* policy continue to exist? _____
4. If the *Insurance* policy is destroyed, must the relationship between the *Car* and the *Insurance Company* also be destroyed? _____
5. If the *Insurance* policy is destroyed, must the *Rate* be destroyed too? _____
6. Can a *Car* have several *Insurance* policies? _____

WRITE DOWN THE CURRENT TIME (HH: MM: SS) _____

…

Note: For reasons of brevity, only Alternative 1 is shown. See www.dsic.upv.es/~einsfran/experiment for the alternatives 2-3 (in Spanish).

SECTION C: Rating tasks

In your opinion, which one of the UML class diagrams presented in Section B best represents the scenario illustrated in the Sequence Diagram of Fig. 11? (Mark your choice with an "X")

Alternative 1 ( ) Alternative 2 ( ) Alternative 3 ( )

Towards Quality-Driven Model Transformations: A Replication Study

Emilio Insfran[1], José Ángel Carsí[1], Silvia Abrahão[1], Marcela Genero[2], Isidro Ramos[1], Mario Piattini[2]

[1] ISSI Group, Department of Information Systems and Computation
Universidad Politécnica de Valencia
Camino de Vera, s/n, 46022, Valencia, Spain
{einsfran, pcarsi, sabrahao, iramos}@dsic.upv.es

[2] ALARCOS Group, Department of Information Systems and Technologies,
University of Castilla-La-Mancha
Paseo de la Universidad Nº 4, 13071, Ciudad Real, Spain
{Marcela.Genero, Mario.Piattini}@uclm.es

Abstract. Commonly, there are several ways to transform a source model into a target model. These alternative target models may have the same functionality but can differ in their quality attributes. One of the key challenges of an automated transformation process is to identify the transformations that will produce a target model with the desired quality attributes. In this paper, we present a replica of a controlled experiment to investigate the selection of alternative transformations to obtain UML class models from a Requirements Model. This is a concrete instantiation of a wider domain-independent approach for quality-driven model transformation. Specifically, we focus on a set of transformations related to structural relationships between classes (association, aggregation and association class) and the understandability quality attribute. Although, some results could be foreseen even by a superficial analysis of the alternatives, the goal of this work is to use experimentation to gather empirical evidence about which alternative transformation produces the UML class model that is the easiest to understand. The empirical results support the original results showing that there is a tendency to favor the use of association relationships to drive these transformations when understandability is chosen.

Keywords: Model transformations, MDA, Software Quality, Requirements, UML class model, Empirical Software Engineering.

1 Introduction

Model-Driven Architecture (MDA) is an emerging approach to software development. It promotes the use of models and model transformations as the primary artefacts to be built and maintained. In essence, an MDA development process transforms a Platform-Independent Model (PIM) into one or more Platform-Specific Models (PSM), which are then transformed into code (Code Model – CM). Therefore,

Insfrán, E., Carsí, J. A., Abrahao, S., Genero, M., Piattini, M., Ramos, I. Towards Quality-Driven Model Transformations: A Replication Study. Proc. of the International Workshop on Empirical Studies of Model-Driven Engineering (ESMDE 2008) at MODELS 2008. 2008.

in this context, a model is no longer simply a means for describing software, but rather an essential piece of the software development process. Consequently, the quality of the models built throughout this process is of great significance since these models will determine the quality of the software product that is finally deployed.

Usually, in an MDA development process there are several ways to transform a source model into a target model. Alternative target models may have the same functionality but differ in their quality attributes. One of the key challenges for an *automated transformation process* is to identify which transformations will produce a target model with the desired quality attributes (e.g., understandability, modifiability).

In the last few years, some approaches that deal with quality in Model-Driven Engineering (MDE) have been proposed [15] [14] [12] [13] [8] [9]. One disadvantage of these approaches is that the practical applicability of model transformations is often reported based on the intuition of the researcher. As pointed out by Czarnecki and Helsen [4], there is a lack of controlled experiments to fully validate the observations made by the researchers in the field of MDE. Therefore, more systematic approaches to ensure quality in MDE processes are needed.

In this paper, we present a replica of a controlled experiment to investigate the selection of alternative transformations to obtain UML class models from a Requirements Model [5] [6]. This work is part of a project on *quality-driven model transformations* whose overall goal is the definition of a quality metamodel to drive the selection of alternative model transformations according to different quality attributes.

Specifically, we focus on a sub-set of transformations that are related to structural relationships between classes (association, aggregation, and association class) and the 'understandability' quality attribute. These transformations have been selected because the determination of structural relationships has a great impact on the UML class model. Understandability has been selected since it is recognized as the main quality attribute that influences maintainability. A UML class model must be understood before any desired change to it can be identified, designed or implemented.

The goal of the experimentation is to gather empirical evidence about which alternative transformation produces the UML class model that is the easiest to understand. The empirical evaluation of the best transformation is particularly important when the transformations are automatically applied, which is the main reason for adopting MDA [11].

This paper is organized as follows. Section 2 gives an overview of our approach to transform a Requirements Model into UML class models. This section also shows the analysis of alternative model transformations. Section 3 describes the design of the original experiment and its replica to empirically validate the selection of alternative transformations with regard to the understandability. Section 4 presents the data analysis and the interpretation of the results. Finally, section 5 presents our conclusions and future work.

2 Transforming Requirements into UML Class Models

Following an MDA development approach, it is important to ensure quality in every step of the development process. In this context, automated model transformation plays a key role for success. We propose to empirically validate model transformations with regard to quality attributes and use this information to drive the selection of alternative transformations. A quality attribute is a measurable physical or abstract property of an entity (e.g., conceptual model) [7].

A model transformation is executed taking a transformation definition as input. A transformation definition contains transformation rules that relate constructs in the source model to constructs in the target model. We use another input for the transformation process that is the definition of the quality attributes together with the corresponding empirical evidence gathered from controlled experiments. This information will feed the transformation process with the criteria to choose the alternative transformation that maximizes the selected quality attribute. The rationale of this approach is to be able to automatically select the alternative transformation that an experienced software developer would select if the transformation process were manually applied. In this section, we present a concrete application of the quality-driven model transformation approach to transform software requirements into UML class models (see Fig. 1).

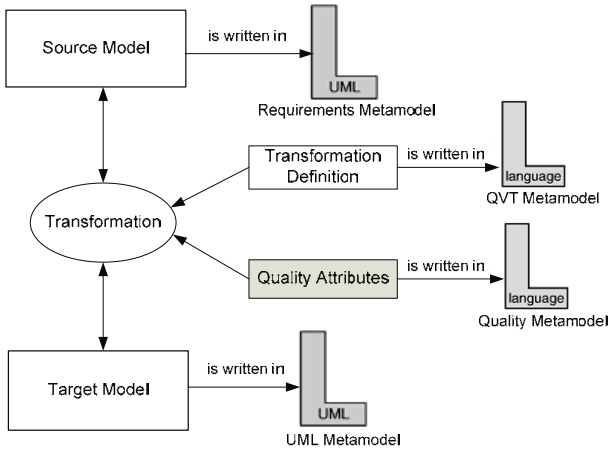


Fig. 1. The quality-driven model transformation approach

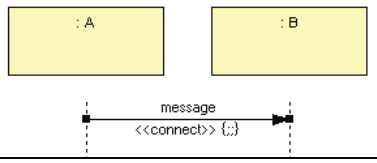
The Requirements Model [5] [6] defines the structures and the process followed to capture the software requirements following an MDA approach. It is composed of a *Functions Refinement Tree (FRT)* to specify the hierarchical decomposition of the system functionality, a *Use Case Model* to specify the system communication and functionality, and *Sequence Diagrams* to specify the required object-interactions necessary to realize each Use Case. This Requirements Model is supported by a Requirements Engineering TOol – RETO (http://reto.dsic.upv.es). Once the Requirements Model has been specified, a conceptual model including a UML class model can be obtained by applying a set of transformations from the Transformation Rules Catalog (TRC) [5]. These transformation rules establish traceability relationships between the Requirements Model and the UML class models.

The application of a Transformation Rule (TR) implies that a certain structural pattern match in the Requirements Model and that a resultant structure in the UML class model can be generated while establishing a traceability relationship between them. Some transformations are easy to apply once the transformation pattern has been matched (a *one-to-one* relation). For example, the generation of classes for the UML class model is based on the analysis of participating classes in all the Sequence Diagrams. However, other transformations are not easy to identify or apply for three main reasons: the complexity of the transformation pattern, the non-disjoint condition pattern of the transformation, and the multiple valid representation of a conceptual model for a given requirement pattern.

2.1 Analyzing Alternative Transformations

Following, we explain with examples some transformations where multiple alternatives arise because of non-disjoint condition patterns and multiple possible representations of the same pattern. Fig. 2 shows the Sequence Diagram used to specify the necessary object interactions to realize the Use Case *Create Insurance* of a Car Rental system. This Use Case represents the creation of a car *Insurance* policy that must be bought from an *Insurance Company* and assigned to the *Car* before using the car for rentals. The actor *Administrator* initiates the interaction (message 1). After introducing the necessary data and checking the existence of the corresponding car (messages 2 to 4), a new *Insurance* object is created (message 5). In addition, an *Insurance Company* object and a *Car* object must be connected to the newly created *Insurance* policy object (messages 6 and 7).

After analyzing the requirement specification provided by the previous Sequence Diagram, the analyst of the system could possibly determine that the partial Class Model that best represents this requirement is defined by two association relationships that relate the newly created class *Insurance* (message 5) to the *InsuranceCompany* and *Car* classes as shown in Fig. 3(a). This partial Class Model can be obtained by applying twice the following transformation rule TR15 (Association) from the Transformation Rule Catalog to the messages 6 and 7:

TR15 (Association): *For every message between two classes labeled with the stereotype «connect», THEN an association relationship between these classes will be generated.*	
Pattern condition:	**Result:**
	

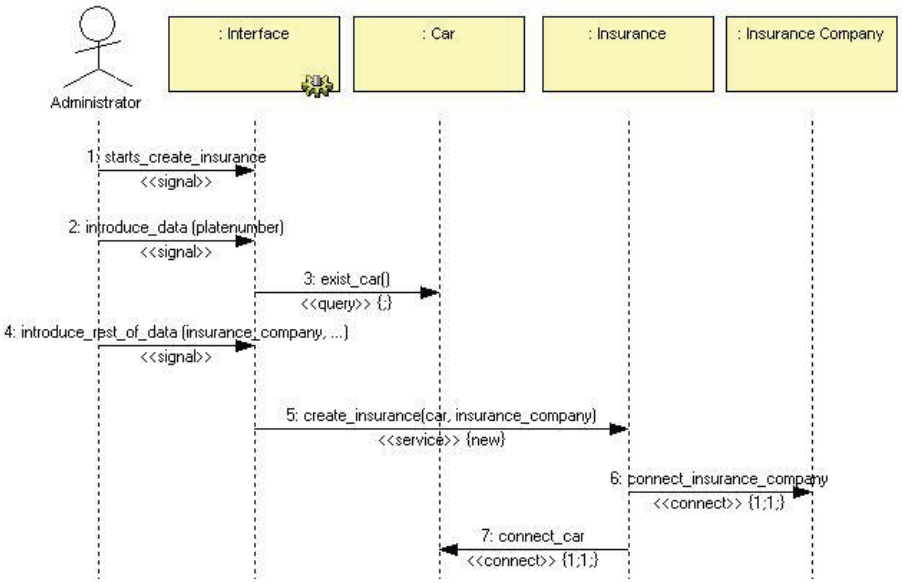


Fig. 2. Sequence Diagram showing the required interactions for the Use Case *Create Insurance*

As an alternative solution to the same requirement specification, the analyst could prefer a solution that uses an association class (named *Insurance*) to relate the *Car* rented with the *Insurance Company* as shown in Fig. 3 (b). This partial Class Model can be obtained by applying the transformation rule TR39 (AssociationClass) to the messages 5, 6 and 7. Finally, another possibility is the definition of two aggregation relationships, one between *Insurance Company* and *Insurance* classes and another between *Car* and *Insurance* classes as shown in Fig. 3 (c). This partial Class Model can be obtained by applying twice the transformation rule TR28b (Aggregation) to the messages 6 and 7.

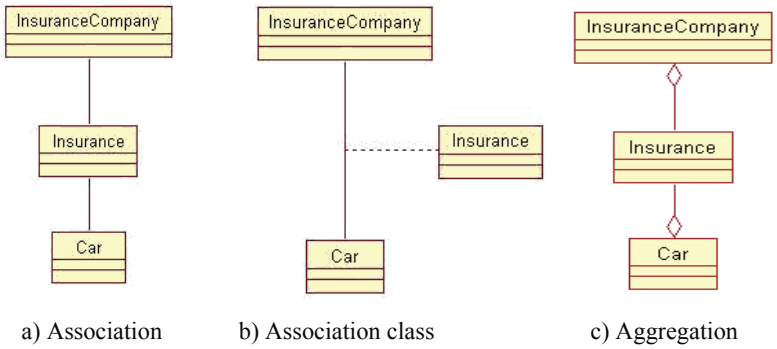


Fig. 3. Two partial Class Models for the Car Rental system

An important issue derived from these examples is that different alternative structural relationships can be derived from the analysis of the requirements specifications described using Sequence Diagrams. And what is more important is the fact that the part of the information used in the Sequence Diagrams for deriving these

structural relationships follows the same pattern: a message with the stereotype *new*, followed by two messages with a stereotype *connect*.

In the examples, we have shown that TR15, TR39, or TR28b transformation rules could be applied, depending on the interpretation of the analyst if the transformation process is performed manually. In an automated transformation process, the information provided by the stereotyped Sequence Diagrams is not sufficient to automatically determine which structural relationship of a Class Model best realizes the specified requirement. Because these alternatives exist, it is the human analyst who should decide which alternative better represent a solution in the corresponding problem domain. In this work, considering that alternatives exist, we use controlled experiments to discover which alternative transformation maximizes a given quality attribute of the resulting target model (e.g., understandability).

3 A Controlled Experiment and its Replica

In a previous work [1], we presented a controlled experiment to determine which of the transformation rules for structural relationships: association (A1), aggregation (A2), or association class (A3), obtained the easiest to understand UML class model. The results show a slight tendency to favor the transformations related to associations (A1). However, as Basili et al. [2] suggested relevant and credible results can only be obtained by replicating the experiments. In other words, single studies rarely provide definitive answers. Therefore, we carried out an internal strict replication (changing only the subjects) to corroborate the findings. To provide an overall view of the experimentation, we explain both the original experiment and the replica.

3.1 Planning

The participants in the original experiment were 39 fourth-year students in Computer Science at the Universidad Politécnica de Valencia, who were taking part in the second Software Engineering course. We took a "convenience sample" (i.e., all the students in the class). The subjects had six month of experience in modeling with UML and three years of experience in the OO paradigm. The subjects that participated in the replica were a different group of 37 students from the same course.

The independent variables for the experiments were the transformation rules for structural relationships between classes (A1, A2 and A3). The dependent variable was understandability. The experimental material and tasks consisted of:

- 9 Sequence Diagrams from three different case studies (a car rental system, a hotel management system, and a singer contest system), with 3 UML class models each. These were obtained by applying the alternative transformation rules. The material used is available at: www.dsic.upv.es/~einsfran/experiment.
- Each Sequence Diagram had a questionnaire attached consisting of 6 Yes/No questions to test the subjects' understanding of the Sequence Diagrams. The effectiveness of the subjects in answering the questionnaires (number of correct

answers by number of answers) was used to exclude those observations that did not fulfill a minimum level of quality.

- Each of the three UML class models had a questionnaire attached (with 6 questions) for assessing which UML class model was best understood by the subjects. In addition, the subjects had to write down the starting and ending times for completing the questionnaires. For this purpose we used a wall clock. From this understanding task, we obtained three measures for understandability:
  Understandability Time, which reflects the time, in seconds, that the subjects spent answering each questionnaire (calculated by the difference between the ending time and the starting time). Each subject completed 4 questionnaires detailing 3 alternatives (A1, A2 and A3). Three understandability time measures (A1Time, A2Time and A3Time) were obtained.
  Effectiveness, which reflects the correctness of the answers (number of correct answers by number of answers). Three understandability effectiveness measures (A2Effec, A2Effec and A3Effec) were obtained.
  Efficiency, which reflects the correctness of the answers by time (number of correct answers by understandability time). Three measures for understandability efficiency (A2Effic, A2Effic and A3Effic) were obtained.
- The final task of each test asked the subjects which of the three alternative UML class models best reflected the problem modeled in the Sequence Diagram. It is a subjective measure (*Alternative Selected*) based on the subjects' perception.

The following hypotheses were formulated:

- **$H1_0$**: The use of different alternative transformations does not affect the Understandability Time (A1Time, A2Time and A3Time). $H1_1 = \neg H1_0$
- **$H2_0$**: The use of different alternative transformations does not affect the Understandability Effectiveness (A1Effec, A2Effec and A3Effec). $H2_1 = \neg H2_0$
- **$H3_0$**: The use of different alternative transformations does not affect the Understandability Efficiency (A1Effic, A2Effic and A3Effic). $H3_1 = \neg H3_0$
- **$H4_0$**: There is no correlation between the Alternative Selected and the means of objective Understandability variables (Understandability Time, Effectiveness, and Efficiency). $H4_1 = \neg H4_0$

We selected a balanced within-subject design, i.e., each subject received the same experimental material.

3.2 Execution

Both the original experiment and the replica started with an introductory session in which we reviewed the main concepts of the Requirements Model (e.g., the notation of Sequence Diagrams). The goal of the experiment was not disclosed to the subjects. Then, we showed an example of the experimental material, which was similar to the material they would use during the execution of the experiment.

Each subject was assigned all the material, with the nine tests (balanced within-subject design). The models were assigned in different order to limit learning effects. We showed them how to develop the experimental tasks, and they had a maximum of two hours to complete all the tasks.

After the experiment took place, we collected the experiment data, which consisted of a table of 351 rows (9 models x 39 subjects) and 9 columns (A1Time, A2Time, A3Time, A1Effec, A2Effec, A3Effec, A1Effic, A2Effec, A3Effec). The replica had the same structure, but with 331 rows (9 models x 37 subjects). In both samples, we performed a "data cleaning", excluding the observations that were not complete because the subjects had not written down the time or because the subjects did not select the best alternative. All the questions were answered in each questionnaire, thereby assuring the completeness of the performed tasks. We also excluded the observations that had a value of effectiveness of 50% or less for each Sequence Diagram. We considered that if the level of correct answers was low in relation to the Sequence Diagram, the subjects had not really understood the model, and they would probably not perform well in the following tasks, so we discarded them. Therefore, the final data for testing the hypotheses were 325 observations for the original experiment and 293 for the replica.

4 Data Analysis and Interpretation

The following statistical analyses were performed to analyze the data: (1) a descriptive study was done to characterize the variables Alternative Selected, Understandability Time, Effectiveness, and Efficiency; (2) Hypotheses H1, H2, and H3 were tested using an ANOVA test with repeated measures; (3) Hypothesis H4 was tested using the Spearman correlation coefficient.

We used SPSS to carry out the data analyses presented in this study. The transformation most selected by the subjects was A1, i.e., the subjects believed that the use of associations obtained the easiest to understand UML class model. Association class (i.e., alternative A3) was the transformation that was least selected, which reveals that it could be the least appropriate transformation.

The descriptive statistics we carried out for Understandability Time, Effectiveness and Efficiency suggest the following:

- **Original Experiment.** On average, the subjects spent less time performing the tasks related to alternative A2; however, the difference with the others is not very significant (≈ 8 seconds for A1 and A3). The subjects were more effective and efficient performing the tasks related to alternative A1; however, the difference in effectiveness with the other alternatives is not very significant.
- **Replica.** The measures related to A1 have the best values, which means that, on average, the subjects spent less time and were more effective and more efficient performing the tasks related to the class model that was obtained via associations.

In summary, the descriptive statistics show a slight tendency in favor of A1, the transformation based on associations. Surprisingly, the difference between the minimum and maximum time values is significant. This may be due to the fact that the subjects were novice modelers.

To test the hypotheses presented in section 3.1, we carried out an ANOVA for repeated measures, which is the appropriate statistical test for analyzing the collected data [10]. Due to space constraints, we will briefly present the main findings obtained through the ANOVA for each data sample:

- **Original Experiment.** We can reject hypotheses $H1_0$ ($p = 0.0002$), $H2_0$ ($p = 0.0001$), and $H3_0$ ($p = 0.0005$), with a significance level = 0.05. This means that the use of different alternative transformations really affects understandability time, effectiveness, and efficiency.
- **Replica.** We can reject $H1_0$ ($p = 0.0028$) and $H2_0$ ($p = 0.0003$), which means that the use of one alternative or another does not affect efficiency but does affect time and effectiveness when the subjects understand the class models.

When planning the experiment, we designed it in such a way as to alleviate the threats to the internal validity.

One limitation to the external validity (i.e., the generalization of the findings) of this study is the fact that the three alternative transformation rules cannot be applied simultaneously to all modeling situations. For instance, to establish an association class relationship (A3), at least one «service/new» message and two «connect» messages are needed in the source model. The goal of this experimentation was to gather empirical evidence for the specific case when the three alternative transformations could be applied to obtain a relationship between classes. We are aware that, more alternatives may be possible to represent structural relationships between classes. More experimentation is needed to validate these other combinations. Another limitation to the external validity might be the use of students as experimental subjects. However, the students who participated in the experiment can be considered to be representative of novice users of conceptual modeling approaches. To increase external validity, the current study needs to be replicated using experienced practitioners.

5 Conclusions and Future Work

This paper has presented an analysis of alternative model transformations and how controlled experiments can be used to provide useful information to guide the selection of transformations in an automated transformation process. In particular, we presented the results of an experiment to gather evidence about which alternative transformation produces the UML class model that is easiest to understand.

The main findings obtained from the experimentation are the following: (a) the transformation that was most selected in the original experiment and the replica was the association transformation (A1); (b) the results of the replica confirm the results of the original experiment for effectiveness. The subjects were more effective when they understood the class models with association relationships; (c) the fact that the hypothesis related to efficiency could not be confirmed in the replication has no great impact on our approach since the transformations are automatically executed in a Model Management framework (MOMENT) [3].

These results show that there is a slight tendency to favor the use of association relationships as part of an automated transformation process. A possible reason for this could be that this relationship has less semantic strength than the other kinds of relationships. When an aggregation relationship is chosen instead of an association relationship, analysts know that they are defining a part-of relationship. In the case of an association class, it is possible to represent almost the same relationship using two

association relationships. Although the results obtained can be quite obvious the important point is the systematic approach presented to validate this 'obvious' results. That the association relationship is more understandable than the aggregation relationship or association classes is something that almost all the people can say but until this moment no one has the data to confirm that result but merely by intuition.

These preliminary results provided empirical evidence that can be further used to define a domain-independent quality metamodel to drive the execution of model transformations. Nevertheless, more replication is needed for building a body of knowledge. We plan to replicate this experiment with practitioners. Future work also includes the evaluation of the remaining transformations of the Transformation Rules Catalog taking into account other quality attributes (e.g., usability, modifiability).

Acknowledgments. This research is part of the META project TIN2006-15175-C05-05, the MECENAS project PBI06-0024, and the IDONEO project PAC08-0160-6141.

References

1. Abrahão, S., Genero, M., Insfran, E., Carsí, J.A., Ramos, I., Piattini, M.: Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In: Model-Driven Software Development: Integrating Quality Assurance, IGI Publishing, 2008.
2. Basili, V., Shull, F., Lanubile F.: Building Knowledge through Families of Experiments. In: IEEE Transactions on Software Engineering, 25(4): 435–437, 1999.
3. Boronat, A., Carsí, J.A., Ramos, I.: Algebraic Specification of a Model Transformation Engine. In: Fundamental Approaches to Software Engineering (FASE'06). ETAPS'06. Vienna, Austria, 2006, pp. 262–277.
4. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. In: IBM Systems Journal 45(3): 621–645, 2006.
5. Insfran, E.: A Requirements Engineering Approach for Object-Oriented Conceptual Modeling, PhD Thesis, DSIC, Valencia University of Technology, 2003.
6. Insfran, E., Pastor, O., Wieringa, R.: Requirements Engineering-Based Conceptual Modelling. In: Journal of Requirements Engineering 7(2): 61–72, 2002.
7. ISO, ISO/IEC 9126-1, (2001). Software Engineering – Product quality P1:Quality model.
8. Ivkovic, I., Kontogiannis K.: A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations. In: Conf. on Software Maintenance and Reengineering, 2006, pp. 135–144.
9. Kerhervé, B., Nguyen, K.K., Gerbé, O., Jaumard, B.: A Framework for Quality-Driven Delivery in Distributed Multimedia Systems. In: AICT/ICIW 2006, 2006, pp. 195–205.
10. Kirk, R.E.: Experimental design. Procedures for the behavioural sciences. Brooks/Cole Publishing Company, 1995.
11. Kontio, M.: Architectural Manifesto: The MDA adoption manual, (2005), Accessible at http://www-128.ibm.com/ developerworks/wireless/library/wi-arch17.html
12. Kurtev, I.: Adaptability of Model Transformations. PhD Thesis, Univ. of Twente, 2005.
13. Markovic, S., Baar, T.: Refactoring OCL Annotated UML Class Diagrams. In: 8th Int. Conf. on Model Driven Engineering Languages and Systems, 2005, pp. 280–294.
14. Merilinna, J.: A Tool for Quality-Driven Architecture Model Transformation. In: Espoo, VTT Electronics, VTT Publications, 2005.
15. Rottger, S., Zschaler, S.: Model-Driven Development for Non-functional Properties: Refinement Through Model Transformation. In: The Unified Modelling Language (UML) Conference, LNCS Volume 3273, 2004, pp. 275–289.

A GENERIC SOLUTION FOR THE CONSTRUCTION OF DIAGNOSTIC EXPERT SYSTEMS BASED ON PRODUCT LINES

Ma Eugenia Cabello Espinosa, Isidro Ramos Salavert
Polytechnic University of Valencia,Camino de Vera s/n, 46022 Valencia,Spain
mcabello@dsic.upv.es, iramos@dsic.upv.es

Keywords: Expert Systems, Medical Diagnosis, Software Architectures, Reusability of Software, Software Product Lines, Variability, Domain Engineering, Domain Application Engineering, Conceptual Models.

Abstract: This paper presents a generic solution for the construction of diagnostic expert systems using aspect-oriented-software architectures and product line techniques. The approach is shown by specifying a case study using CIMs, and automatically generating a PIM. The case study presented is a medical diagnosis system for the detection of infantile infectious diseases. PRISMA models are used as PIMs. We follow the Model Driven Architecture (MDA) initiative of the Object Management Group (OMG) for building domain models (CIMs), which are automatically transformed into PIMs and compiled to a .NET executable application (PSM). The Software Product Line techniques have been used to capture the variability of systems of this kind.

1 INTRODUCTION

In the last few years, there has been an increase in interest in expert systems that perform diagnostic tasks. The main objective of systems of this kind is to capture the state of an entity from a series of data (observation variables) and produce a diagnosis. The domain of expert systems for diagnosis includes systems for medical and education diagnoses, among others. Since systems for medical diagnosis have become more relevant, the need of techniques for their development has also become more important. Additionally, expert systems introduce a difference in the decision making process: they store expert knowledge in a Knowledge Base.

In order to capture complex software requirements, it is also necessy to increase the PIM abstraction level. The PRISMA framework (Pérez, 2006) provides expresivity for specifying software architectures with aspects at a design level. It offers properties and advantages in the construction of complex, distributed, evolutionary, and re-usable architectural models that can be used in the domain of expert systems for medical diagnosis.

Furthermore, the development of these complex systems is becoming more elaborate due to a series of factors. These factors are the emergence of new technologies (Internet and intranet), the interconnection of the different systems and platforms, and to integrate Legacy Systems that are still valid. Also, the need to develop custom software for each type of user complicates the specific aspects of systems in different implementation platforms. This state of affairs requires having multiple versions of the same application in order to deal with all of this variability.

In order to cope with this variability problem, Software Product Lines (SPL) (Clements et al., 2002) have emerged in an effort to control and minimize the high costs of the software development process and to reduce the time to market of these new products. This approach is based on having a base design that is shared by all the product family members. Thus, a specific product can benefit from the common parts of the software. The base design can be re-used in different products by adding different features that caracterize them.

We have built BOM (Base-Line Oriented Modeling), wich is a framework that automatically generates diagnostic systems based on software product lines, to achieve the following goals:

- to create new diagnostic systems in different domains,
- to decrease production costs by reusing software packages or assets,

Cabello, M. E., Ramos, I. A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines. Proc. of the International Conference of Health Informatics- HEALTHINF´08. 2008.

- to generate automatic code to increase the productivity and quality of software and to decrease the time to market,
- to construct systems in a simpler way by using the ontologies of the diagnosis and the application domains. The models will be closer to the problem domain, which will facilitate user interaction,
- to develop platform independent systems from the problem perspective and not the solution perspective, which will provide generality in the development approach and applicability in different domains.

The products of our Diagnostic Software Product Line (DSPL) have been designed using the PRISMA model and include the architecture and operations of a rule-based expert system.

In this paper the development process of our Diagnostic Software Product Line is presented using a case study of infantile infectious diseases.

The structure of the paper is the following: Section 2 introduces the relevant concepts of Expert Systems, Model Driven Development, the PRISMA Model, and Software Product Lines. Section 3 introduces the variability dimensions of our domain. Section 4 provides an in-depth description of our approach. Section 5 presents a brief summary of the case study. Section 6 presents related works. Section 7 presents our conclusions and provides some ideas for future work.

2 FOUNDATIONS

Our work integrates different technological spaces in order to cope with the complexity of the problem. These are the following:

2.1 Expert Systems

Expert systems capture the knowledge of experts and try to imitate their reasoning processes when the experts solve problems in a certain domain.

These systems usually have a basic architecture that includes a knowledge base, an inference motor, a working memory or facts base, and the user interface. These are the four main components of the architecture of a rule-based expert system.

These components are independent and are composed of separates units. The data are grouped into the working memory (temporary storage of dynamic information). The representation of the knowledge is captured by means of rules of the type: IF <antecedent > THEN <conclusion>, which make up the knowledge base. The control aspect is independent and is performed by the inference motor during the inference processes using different reasoning strategies. The input and output of the information of the systems are done through the user interface.

2.2 Model Driven Development

The Model Driven Development approach (MDD) for software system development is based on the separation of the functionality of the system from its implementation on specific software platforms. MDD increases the abstraction level of the software production process by emphasizing the importance of the conceptual models - Computer Independent Models (CIM) or Platform Independent Models (PIM) - and their role in the software development process.

There are currently two main initiatives in MDD. One of them, which is promoted by the Object Management Group (OMG) is called Model Driven Architecture (MDA) (http://www.org.mda). The other one, promoted by Microsoft, is called Software Factories (SF) and Domain Specific Languages (DSL) (Greenfield et al., 2004).

The key idea of MDA is to focus on the models as first class citizens in the software development process. MDA proposes defining and using models at different abstraction levels. These models can automatically generate code by means of mappings or by applying transformation rules to executable Platform Specific Models (PSM).

2.3 The PRISMA Model

The PRISMA architectural model integrates two approaches: Component-Based Software Development (CBSD) (Szyperski, 1998), and Aspect-Oriented Software Development (AOSD) (http://aosd.net). This integration is obtained by defining the architectural elements through their aspects.

The PRISMA model consists of three types of architectural elements: components, connectors, and systems. A component captures the functionality of the system, whereas a connector acts as a coordinator among other architectural elements. A system is an architectural element of great granularity that allows the encapsulation of a set of components, connectors, and other systems. This, in turn, allows the system to correctly connect them.

PRISMA defines the architectures in two abstraction levels: the type level and the

configuration level. In the type level, the types of the architectural artifacts are defined (all of which can be reused): interfaces, aspects, components, connectors, and systems. In the configuration level, the types are instantiated and the topology of the model (its configuration) is specified.

2.4 Software Product Lines

The Software Product Line (SPL) approach, from a practical point of view, is one of the most successful since it combines systematic development and the reuse of reusable components or assets; i.e., the products are different in some features but share a basic architecture. SPL provides an industrial approach to the software development process.

In the SPL approach, rather than a single application, the development process produce a series or family of them. This implies changing the existing engineering process by introducing a distinction between the domain engineering process and application engineering process. In general, the domain engineering process defines the shared architecture and the variability of the SPL. More than creating products, it is a question of putting assets together in a Base-Line warehouse. For each SPL there is a well defined production plan that specifies the process to obtain each of the individual products. The construction of the assets and their variability (domain engineering process) is separate from the application production (application engineering process).

One of the most important points (or perhaps the most important) of a SPL is the definition of the basic architecture of the Products Line (also called domain architecture). This is due to the fact that this architecture determines the scope of the SPL and the features of the products that can be developed.

One of the key elements for a SPL is how to represent and manage variability. In the context of BOM variability appears in the construction of the domain model (which is represented as a decision tree with different variation points). The base assets are selected by the decision tree. These assets are enriched by the specific features (given in the application model) by a process that results in PRISMA architectural model types. In BOM, the assets are automatically transformed by inserting the instances of the Feature Model (Batory et al., 2006), which gives one executable application.

2.5 Feature Oriented Programming

Feature Oriented Programming (FOP) is the study of the modularity of the features of a domain and their use. The features are considered as first-class citizens in the design process. FOP is an approach to SPL where the programs are built by means of feature composition. These features are considered as building blocks of the programs. They are units that increase monotonically the functionally of the application by providing different products. Each one of the features can be included in the different software artifacts. In general, a SPL is characterized by the set of features being used, which is called the feature model of the SPL.

3 THE VARIABILITY

3.1 Variability in the diagnostic domain

After a field analysis of different systems in the diagnostic domain, we can conclude that a diagnosis consists of an interpretation of the involved entity states (as a set of properties), which is followed by the identification of the problem or disfunction by means of its properties. We have detected seven features (or variability sources) that are present in these systems. They are the following:

- a) property views: an entity can have the same properties (the same view), or have different properties (different views) during the diagnostic process,
- b) property levels: the properties of the entities can have *n* different abstraction levels. The rules that relate the proprieties of the entities have *n-1* levels, where n is the level of the proprieties of the entities.
- c) number of hypotheses: the goal of the diagnosis is a single validated hypothesis. There can be one or several candidate diagnostic hypotheses which must all be validated in order to select the appropriate one.
- d) reasoning types: reasoning shows the way in which the rules are applied by the inference motor in order to infer a final diagnosis. The reasoning types can be: deductive reasoning (driven by data), inductive reasoning (driven by goals), and differential reasoning (establishing the difference between two or more diagnostical possibilities),
- e) use case numbers: a use case indicates the division of the system based on its functionality; i.e., the different operations of

the systems and how the system interacts with the environment (final users),

- f) number of actors: represents the number of final users of the system,
- g) use cases per actor: a final user can access different use cases.

The variability shown in the use cases, actors, and use cases per actor is reflected in the construction of the architectural elements assets (skeleton) and in the final PRISMA architecture in the following way:

- there is one connector that connects all the architectural component assets for each use case,
- the number of ports of the Inference Motor component is the number of use cases,
- the number of ports of the Knowledge Base is the number of use cases,
- the number of User Interfaces is the number of actors of the use cases,
- the number of ports of the User Interface is the number of use cases that can be accessed by an actor.

The variability of the diagnostic domain is expressed by means of an explicit Features Model.

3.2 Variability in the application domain

The features selected from the Features Model must be defined following the case study in which the application is developed. These features are:

- a) name and type of the properties by abstraction level,
- b) rules by abstraction levels: the rules relate the entity properties (name and type) with the properties that are used in the head part and the body part of each one of these rules,
- c) name and type of the hypothesis (and the pre-hypothesis) that are used in the diagnostic process.

4 THE BOM APPROACH

Our work is based in the Software Product Lines approach, on two OMG standards: the Reusable Asset Specification (RAS), which identifies, describes and packs assets in an standard way; and the Software Process Engineering Metamodel (SPEM) (http://www.omg.org/docs/ad/06-06-02.pdf), which defines the standard language for modeling the software process.

In BOM a clear separation is made between the domain engineering and application engineering. This partition is the basis for the reuse and the automation of the software process (Czarnecki et al, 2000). In domain engineering phase, a set of assets and processes are created. In the application engineering phase, these assets are used to produce software products of high quality with a minimal cost and time by executing the stored processes.

4.1 1st Phase: Domain Engineering

In this first phase, the following software artifacts are created by the diagnostic domain engineer. These artifacts are necessary to generate the product plan of our software product line: the artifacts a and b are Computation Independent Models (CIMs), and the artifacts c, d, e, f, g, h, i, j, and k are Platform Independent Models (PIM).

- **a) The (CIM) Features Model**.- the Features Model identifies the DSPL in terms of the variability in the domain.

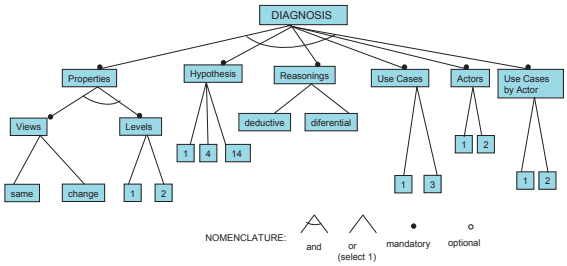


Figure 1: The diagnosis features model

- **b) The (CIM) Decision Tree**.- the sources of variability observed in the Features Model are shown in the variation points of the Diagnostic Decision Tree.

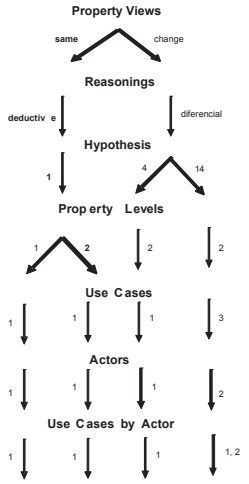


Figure 2: The diagnostic decision tree

- **c) The (PIM) Domain Conceptual Model**.- this model captures the variability of the diagnostic domain.

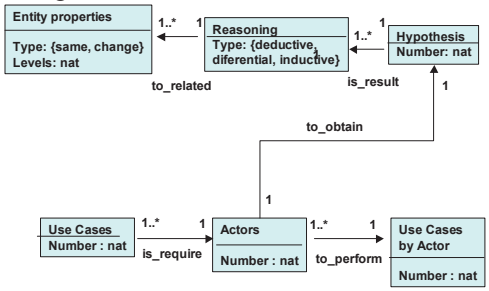


Figure 3: The domain conceptual model

- **d) The (PIM) Application Domain Conceptual Model**.- this model captures the variability of the application domain. (Figure 4 shows the case study of the medical diagnosis).

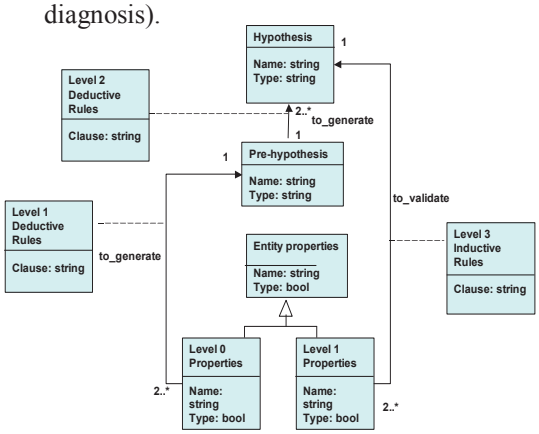


Figure 4: An application domain conceptual model

- **e) The (PIM) Skeletons or Template Assets.-** there are different classes of skeletons or templates for: components, connectors, aspects, and interfaces; they follow the PRISMA Model. The aspects that are necessary for the definition of these architectural elements are: the functional aspects of each one of the components, and the coordination aspect of the connector. These aspects use interface services. These architectural elements are:
  - **The Inference Motor Component-** it establishes the system's control and makes decisions. In addition, the Inference Motor Component provides the general resolution strategy to obtain the diagnosis. It is independent of the system knowledge. This component has a functional aspect that defines the inference process of the system.
  - **The Knowledge Base Component-** it contains the domain knowledge of the case study in rules of inference (Horn clauses) and facts (information that remains unchanged). This component is a temporary warehouse of dynamic information, since the number of facts can be increased as they relate to the inference rules of the domain. When a diagnosis process has concluded, the contents of the work memory is cleared so that the memory is empty before initiating a new diagnosis. The Knowledge Base Component has a functional aspect that defines the domain rules.
  - **The User Component-** it establishes man-machine interaction by allowing communication between the users and the system. Through it, the user offers initial data to the system or answers questions formulated by it. This component has a functional aspect.
  - **The Diagnostic Connector-** this connector synchronizes or requests component's services that are sent/received through its ports. It has a coordination aspect. This diagnostic connector choreographs the diagnostic process.
- **f) The (PIM) Features Insertion Process**.- this process inserts the features into the software artefacts. These artefacts represented as XML documents are transformed using XSLT templates.
- **g) The (PIM) RAS Models of the Assets**.- its goal is to store information from each one of the assets: ID asset identifier, asset classification, description of the different asset artifacts, variability points of the asset artifacts.
- **h) The (PIM) Assets**.- an asset is composed by a skeleton, its RAS Model, and its corresponding insertion process (to be executed in the application construction phase).
- **i) The (PIM) Base-Line**.- the Base-Line is the repository that contains all the assets and all the application domain conceptual models, which are used to capture the specific application features.
- **j) The (PIM) Process for selecting the assets and the application domain conceptual**

models.- This process computes the paths of the decision tree, which is used to select the software artifacts.

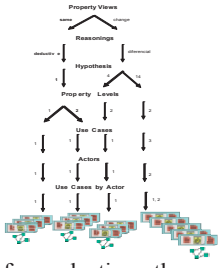


Figure 5: Process for selecting the assets and the application domain conceptual models

- **k) The (PIM) Process of the production plan of our DSPL**.- this process is described using the SPEM.

4.2 2nd Phase: Application Engineering

The production plan of our Software Product Line is described using SPEM. The SPEM Metamodel allows several aspects and problems of the development process to be modeled. In this work, we focus on modeling the tasks, using the SPEM sequence relations without priority. The tasks performed by the application engineer consume input artifacts and produce output artifacts. A task can have associated elements that guide and help in the task execution.

Each one of these tasks is described below:

- **Task 1: To create a configuration of the domain features**.- The engineer introduces the domain information of the case study. BOM captures the variability information by means of the domain features detailed in the Domain Conceptual Model. This model allows the engineer (by means of a GUI) to introduce the information of the Model by selecting the product's features using check boxes and pull-down menus.

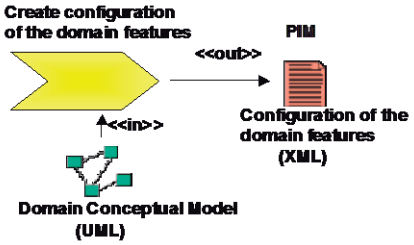


Figure 6: Create a configuration of the domain features

- **Task 2: To select Assets and the Application Domain Conceptual Model**.- BOM selects the assets and the Application Domain Conceptual Model from the Base-Line (by means of the decision tree).

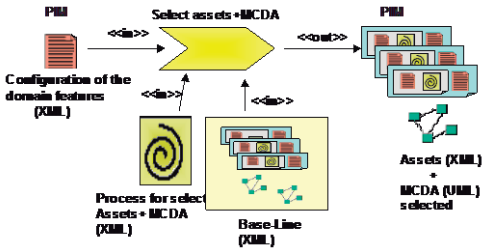


Figure 7: Select assets and application domain conceptual model

- **Task 3: To create a configuration of the application domain features**.- The engineer introduces the application domain information of the case study. BOM captures the variability information by means of the application domain features contained in the Application Domain Conceptual Model. This model allows the engineer (by means of a GUI) to introduce the information present in that Model using check boxes and pull-down menus.

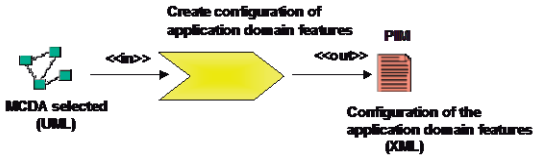


Figure 8: Create a configuration of the application domain features

- **Task 4: To create PRISMA software artifact types**.- BOM fills the selected skeletons with the data of the specific features of the case study that were defined by the engineer, thereby creating the PRISMA software artifact types. The transformation is represented as XSLT documents to apply on the XML pages representing the software artifacts.

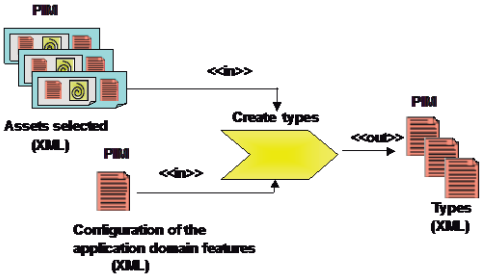


Figure 9: Create PRISMA software artifacts types

- **Task 5: To configure the Architectural Model.-** BOM produces the configuration program, that is used by the PRISMA-CASE tool (Cabedo et al., 2005) in order to configure the PRISMA software architecture by instancing the PRISMA types. These instances will configure the software architectures of our Product Line. Therefore our DSPL are the diagnosis systems of each one of the specific domains.

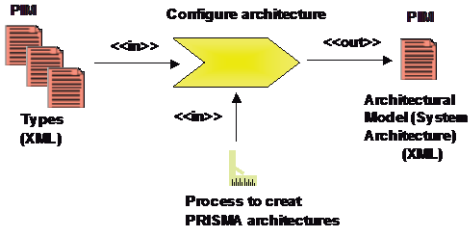


Figure 10: Configure the architectural model

BOM uses the PRISMA-MODEL-COMPILER tool (Cabedo et al., 2005) to automatically generate the code (in .NET, C#) of the software architecture of the preceding task. The final diagnostic system, i.e. an instance of the DSPL, is executed on top of the PRISMANET middleware (Costa et al., 2005).

5 THE CASE STUDY

We have selected the field of infantile infectious diseases as an application domain from the field of medical diagnosis in order to know the requirements necessary to obtain the final software product in our Product Line.

The software system of infantile infectious diseases contains knowledge that uses a set of rules made of the signs and symptoms of diseases of the patient. These signs, symptoms and diseases have been provided by a paediatrician. The system proposed in this work, makes the medical diagnosis by introducing patient data into the system. This data is made up of sign and symptom values and is input by the final users of the system. As a result, the system obtains a diagnosis of the patient's disease. The final diagnosis is made from two types of diagnoses: clinical and laboratory. The objective of this is to provide a highly accurate diagnosis.

Medical diagnosis is understood as the process of the identification or recognition of a disease on the basis of the signs and symptoms (including laboratory studies) of the patient. The medical diagnosis represents the research process performed on the patient, and the diagnosis is based on the observations and reasoning of the doctor.

In medical diagnosis, the entity to be diagnosed is the patient, and the result is the disease that a patient has. The properties are signs and symptoms. These are classified in two abstraction levels: coarse grain and fine grain.

The process of medical diagnosis is the following: A syndrome is inferred from sign and symptom values of coarse grain. Two or more possible diseases are inferred from the syndrome. Deductive reasoning is applied in this part of the process. These hypotheses must be validated. A disease (validated hypothesis) is inferred from sign and symptom values of fine grain. Inductive reasoning is applied in this part of the process.

We present examples of the properties, rules, pre-hypotheses and hypotheses of the case study.

```
properties of level 0:  cough, fever
properties of level 1: dry_cough,
constant_fever
pre-hypotheses: warth, parotiditis
hypotheses: pneumonia, bronchitis
rules:
IF (cough=true and fever=true and
  respiratory_dificulty=tue)
  THEN syndrome=warth
```

The variability sources or points in the diagnostic domain of our Product Line for this case study are the following.

- The hypotheses are inferred by means of different properties of the entities.
- The system exhibits a type of behaviour or reasoning strategy: differential reasoning. This reasoning strategy is the most widely used in solving medical diagnostic problems because it is suited to this kind of task.
- The entity properties of level 0 are the signs and symptoms of coarse grain (e.g. cough and fever). The entity properties of level 1 are the signs and symptoms of fine grain (e.g. dry cough and constant fever).
- The pre-hypotheses are syndromes (e.g. parotiditis) that are inferred by means of rules of level 1. The hypotheses are the diseases (e.g. pneumonia) that are inferred by means of rules of level 2. The diagnostic result is inferred by means of rules of level 3.
- In this case study, several hypotheses are generated. These must be validated in order to obtain only one validated hypothesis, which is the diagnostic result.
- The system offers the user the following functionalities (use cases): clinical diagnosis,

laboratory diagnosis, and the visualization of the results of the final diagnosis.

- The use cases are used by two final users (actors from the use case diagram): doctors or members of the laboratory. Two use cases can be invoked by the doctor, and only one use case can be invoked by a member of the laboratory. In this case study, there are three use cases and two actors, where one actor (the doctor) is associated to two use cases, and the other actor (the member of the laboratory) is associated to only one use case. Figure 11 shows the case use diagram for the medical diagnosis.

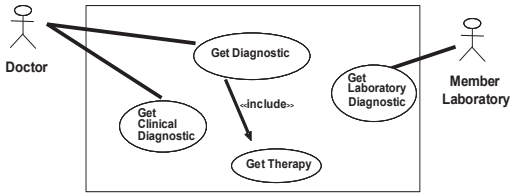


Figure 11: Use case diagram for a medical diagnosis

The skeletons of the architectural elements are: Inference Motor (with three ports), Knowledge Base (with three ports), User Interface 1, i.e., the doctor (with two ports), User Interface 2, i.e., the member of laboratory (with one port), Diagnostic Connector 1 (with three ports), Diagnostic Connector 2 (with three ports), and Diagnostic Connector 3 (with three ports). The architectural model represents a product of our DSLP and is shown in Figure 12

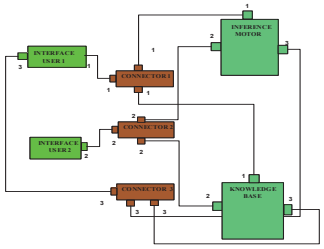


Figure 12: Visual metaphor of the architectural model of the medical diagnosis system

We present the (partial) code that is automatically generated by the PRISMA-CASE tool. This code corresponds to the Knowledge Base Component of the case study.

```
namespace KBMD
{   [Serializable]
    public class
    KnowledgeBaseMedicalDiagnosis:
    ComponentBase
  { public class
     KnowledgeBaseMedicalDiagnosis
     string name:  base (name)
    { AddAspect (new FBaseMD ());
    InPorts.Add
    ("KnowledgeClinicalPort",
    "IDomainMD", "KNOWLEDGE_CLIN");
    OutPorts.Add
    ("KnowledgeClinicalPort",
    "IDomainMD", "KNOWLEDGE_CLIN");
    InPorts.Add
   ("KnowledgeLaboratoryPort",
   "IDomainMD", "KNOWLEDGE_LAB");
    OutPorts.Add
   ("KnowledgeLaboratoryPort",
   "IDomainMD", "KNOWLEDGE_LAB");
    InPorts.Add
   ("KnowledgeResultsPort",
   "IDomainMD", "KNOWLEDGE_RES");
    OutPorts.Add
   ("KnowledgeResultsPort",
   "IDomainMD", "KNOWLEDGE_RES");
}   }}
```

6 RELATED WORKS

There are a great number of works that are related to our approach. The methodologies and applications on this subject have produced a wide variety of research products, offering suggestions and solutions in specific domains.

A study made by (Liao, 2005) examines the methodologies of expert systems and classifies them into eleven categories. Two of these categories correspond to the systems based on rules and the systems based on knowledge. These categories have been taken into account in our work when using knowledge represented in the form of rules (Horn clauses) and facts (observable variables). Likewise, (Liao, 2005) mentions that the applications of the expert systems are built as specific domain problem oriented systems. In our work we present a case study in the medical diagnostic domain.

(Liao, 2005) also mentions that the development of expert systems has been characterized by the separation of knowledge and processes as independent units. In our architectural model, the elements in the type level are defined taking into account this concept, specifically when there is a component that contains the domain knowledge and another component that executes the inference process of the diagnosis.

(Giarratano et al., 2004) and other authors in the field of expert systems considered that the architectures of these systems are based only on components. The architecture of our system has

integrated two approaches combining both components and aspects. This increases the reusability and the maintenance of the system.

Expert systems have also been implemented in the development of different programming paradigms such as structured, logic, and object-oriented paradigms. These paradigms are oriented toward fourth-generation languages and visual programming methods to provide user-friendly communication. PRISMA provides an Architecture Description Language to define an architectural model that follows the MDA approach (http://omg.org.mda), which allows the automatic generation of code.

The integration of the DSBC and DSOA approaches are introduced by (Constantinides et al, 2000). The concerns and requirements described in this work are contemplated in our architectural model. The advantages of each one of these approaches are used to define the architectural elements with their aspects.

MDA proposes the definition of models at high abstraction level, which are independent of the technology (PIM). In our work, we have considered this line of research focusing on experts systems that are based on product lines.

Our work also applies the detection of the components based on the functional decomposition of the problem, which is compatible with the Architecture Based Design Method (ABD) methodology (Bachman, 2000). In ABD software architectures of the application domain are designed. This methodology has been applied in the building of our architectural model for medical diagnosis.

The work by (Garlan, 2001) is a very important reference in establishing the elements of a complex software system. In his work, he introduces the component, connector, system, input port, and output port concepts. These concepts have been included in our model.

Another work that is related to our approach is based on the contract concept of (Andrade et al., 1999). We have defined the connectors of the architectural models of the DSPL, incorporating the choreography concept in the connectors, which are specified by the protocol of the coordination aspect of the connectors.

There are many Architectural Definition Languages (ADL), which have advantages and disadvantages. This study has been done by (Medvidovic et al., 2000). The language proposed by (Loques et al., 2000) in their model R-RIO is the one that is the closed to the PRISMA-ADL. Their model has re-configuration capabilities like PRISMA; however their work does not incorporate the notion of aspect.

Software Product Lines have been an important discussion topic in the last decade. There are many works on this subject. Our research is related to the following works:

- (Batory et al., 2006) express the domain features in the Features Model, and they use Feature Oriented Programming as a technique for inserting the features.
- (González et al, 2006) applied the MDA proposal and Requirements Engineering for Product Lines.
- (Clements et al, 2002) use the SPL development approach, considering a division between domain engineering and application engineering for the reuse and the automation of the software processes.
- (Trujillo, 2007) has developed the XAK tool to insert features into XML documents by means of XSLT templates.
- (Ávila-García et al, 2006) has developed a MDA tool with functionalities of metamodeling over MOF and the transformations in ATC. The authors integrate the functionalities of the process modeling in SPEM, and RAS to package reusable assets .
- (Santos, 2007) proposes the development of a technique based on MDA for variability management in Software Product Lines.
- In (ACM, 2006) several works, related with to the Software Product Line Engineering have been published.

7 CONCLUSIONS

This paper presents BOM (Base-Line Oriented Modeling), wich is a framework that automatically generates diagnostic systems based on software product lines.

BOM has been designed to improve the development of diagnostic systems in following ways:

- To use the advantages of Expert Systems: incorporate several reasoning strategies in order to solve a problem by applying the most efficient one, and separate the inference process of the knowledge information from the application domain.
- To apply techniques from the field Software Product Lines by building a design that shares all the members of a program family. In this way, a specific design can be used in different products. Since we obtain a specific product

from a series of previous models, the costs, time, effort, and complexity can be reduced.

- To construct Product Line Architectures in the PRISMA framework, is order to have the advantages of distributed systems, which will facilitate the management of complexity.
- To create an integrated and flexible approach to describe (medical) diagnosis architectural models that are complex, distributed, and re-usable by improving the development of expert systems for (medical) diagnosis following the PRISMA model (Pérez, 2006) to integrate the components and aspects.
- To apply MDA techniques to implement the systems on different platforms, and to automatically transform them and incorporate the features of the Features Model instances to obtain an executable application.

In the future, we want to extended the analysis of the diagnostic field in other application domains in order to increase variability and our Base-Line. Our Products Line will be able to offer more products. In addition, we plan to validate our approach in other case studies, and compare the performances of the generated Expert Systems with other obtained using other approaches.

Acknowledgements. This work is funded by the Spanish Department of Science & Technology, under the National Program for Research, Development and Innovation, META project (TIN2006-15175-C05-01), and by the European Regional Development Fund

REFERENCES

Andrade L. and Fiadeiro J., 1999. Interconnecting Objects via Contracts. OOPSLA´99.

Ávila-García O., García A. E., Rebull V. S., y García J. L. R., 2006. Integrando modelos de procesos y activos reutilizables en una herramienta MDA, *en XI Jornadas de Ingeniería de Software y Bases de Datos JISBD'2006*, Barcelona, España.

Bachman F., Bass L., Chastek G., Donohoe P. and Peruzzi F., 2000. The Architecture Based Design Method. *Technical Report CMU/SEI-2000-TR-001, Carnegie Mellon University,* USA.

Batory D., Benavides D., and Ruiz-Cortés A., 2006. Automated Analyses of Feature Models: Challenges Ahead. *ACM on Software Product Lines*.

Cabedo R., Pérez J., Carsí J.A. y Ramos I., 2005. "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", *en Actas del IV Workshop DYNAMICA*, Archena, Murcia, España.

Clements P. and Northrop L.M., 2002. *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering, Addison Wesley.

Constantinides C.A., and Errad T., 2000. On the Requeriments for Concurrent Software Architectures to Support Advanced Separation of Concerns. In Proceedings of The OOPSLA 2000, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems*.

Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I., 2005. "PRISMANET: Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", *en Actas de las X Jornadas de Ingeniería del Software y Bases de Datos,* Granada, España.

Czarnecki K., and Eisenecker U., 2000. *Generative Programming: Methods, Tools, and Applications*. Addisson-Wesley. ISBN 0-201-30977-7.

Garlan D., Cheng S. and Kompanek A. J., 2001. *Reconciling the Needs of Architectural Description with Object Modeling Notations*. Science of Computer Programming Journal, Special UML Edition, Elsevier Science.

Giarratano, J., and Riley, G., 2004. *Expert Systems: Principles and Programming.* Fourth Edition: (Hardcover), ISBN: 0534384471.

González-Baixauli B. y Laguna M. A., 2005. MDA e Ingeniería de Requisitos para Líneas de Producto. *Taller sobre Desarrollo Dirigido por Modelos. MDA y Aplicaciones. (DSDM´05*), Granada, España.

Greenfield J., Short K., Cook S, Kent S., and Crupi J., 20004. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley.

Liao S.-H., 20005. "Expert Systems Methodologies and Applications- a Decade Review from 1995-2004", *in Expert Systems with Applications*, Vol. 28, Issue 1.

Loques O., Sztajnberg A., Leite J., and Lobosco M., 2000. On the Integration of Meta-level Programming and Configuration Programming. *In Reflextion and Software Engineering (special edition*), Lectures Notes in Computer Science, Springer-Verlag, Heidelberg, Germany

Medvidovic N., and Taylor R.N., 2002. A Classification and Comparison Framework for Software Architecture, *in Proceedings of IDEAS*, Cuba.

Pérez J., 2006. *PRISMA: Aspect-Oriented Software Architectures*. PhD. Thesis of Philosophy in Computer Science, Polytechnic University of Valencia, Spain.

Santos A.L., Koskimies K., and Lopes A., 2005. *Using Model-Driven Architecture for Variability Management in Software Product Lines*. Ph Thesis Proposal Facultade de Ciencias de la Universidade de Lisboa, Portugal.

Software Product Line Engineering Communications of the ACM. 2006, Vol 49, Number 12, pp 28-88.

Szyperski C., 1998. *"Component software: beyond object-oriented programming",* ACM Press and Addison Wesley, New York, USA.

Trujillo S., 2007. *Feature Oriented Model Driven Product Lines.* PhD. Thesis, The University of the Basque Country, San Sebastian, Spain.

Variability Management in Software Product Lines for Decision Support Systems Construction

María Eugenia Cabello[1], Isidro Ramos[2]

Polytechnic University of Valencia,Camino de Vera s/n, 46022 Valencia, Spain
{mcabello[1], iramos[2]}@dsic.upv.es

Abstract. This paper presents software variability management in complex cases of Software Product Lines where two kinds of variabilities emerge: domain variability and application variability. We illustrate the problem by means of a case study in Decision Support Systems. We have deal with the first one by using variability points that are captured using decision-tree techniques in order to select base architectures and the second one by decorating the base architectures with the features of the application domain. In order to present this variability management, we focus on the diagnostic domain, a special case of Decision Support Systems. A generic solution for the automatic construction of systems of this kind is given using our approach: Baseline Oriented Modeling (BOM).

1. Introduction

The development of Decision Support Systems is complex since the elements that form their software architecture vary. These architectural elements change in their behaviour and in their structure.

In these kinds of systems, the variability management cannot be performed through a unique Feature Model, and the variability must be treated in two phases: the first phase through different base architectures derived from a unique generic architecture; and the second phase by means of a more classic treatment using a Feature Model and decorating these base architectures with different features.

We have illustrated the variability management in the domain of Decision-Oriented Systems, and an application field: the diagnosis of educational programs using BOM (Baseline Oriented Modeling). BOM is a framework that semi-automatically generates DSS in a particular domain, based on product lines techniques.

Our work integrates different technological spaces to cope with the complexity of the problem. These are the following: a) The generic architecture of Decision Support Systems (DSS) [1] to capture the knowledge of experts and to try to imitate their reasoning processes when they solve problems in a certain domain; b) Model-Driven Architecture (MDA of OMG) [2] at the abstract modelling level (PIM); c) The PRISMA Architectural Framework [3] as the target software level (PSM); d) The Software Product Lines (SPL) [4] as a technique for systematic reuse in software products, and e) Feature-Oriented Modeling (FOM) [5] to capture the different variabilities of the problem.

Cabello, M. E., Ramos, I. Variability Management in Product Lines for Decision Support Systems. Proc. of the 10th International Conference on Enterprise Information Systems (ICEIS´08). 2008.

Several methodologies and applications on SPL have produced a wide variety of research products, offering suggestions and solutions in specific domains. Our research is related to the following works: i) Batory et al. [6] express the domain features in the Feature Model, and uses Feature Oriented Programming as a technique for inserting the features. ii) González et al. [7] apply the MDA proposal as well as Requirements Engineering for Product Lines. iii) Clements et al. [4] use the SPL development approach, considering a division between domain engineering and application engineering phases for the reuse and the automation of the software process. iv) Trujillo [5] has developed the XAK tool for inserting features into XML documents by means of XSLT templates. v) Santos [8] proposes the development of a technique based on MDA for variability management in Software Product Lines. vi) The AMPLE project [9] deals with variability management in different ways. This project mentions that Bachman et al. [10] propose to separate the variability declaration from the affected artifacts.

The structure of the paper is the following: Section 2 presents the variability management in the DSS. Section 3 introduces the variability dimensions of our domain (diagnosis). Section 4 introduces the variability in the application domain. Section 5 describes the design of the Decision Software Architecture. Section 6 presents our conclusions and provides some ideas for future work.

2. Variability management in Decision Support Systems

The development process of a specific application (member of the product line) starts from a unique generic architecture of the domain. The DSS domain unique generic architecture is presented in Figure 1(a). This implies the existence of additional particular features that are represented as variability points, and their choice configure the final product. However, another variability emerges, in complex application domains. This variability is represented by the existence of several base architectures for the same generic architecture (see Figure 1(b)). We use a Modular view for describing the generic architecture conforming a Modular view metamodel (MM MODULAR VIEW) and a Component-Connector view for describing the Base Architectures Skeletons conforming a Component-Connector view metamodel (MM C-C SKELETON). Figure 1 shows a unique generic architecture and two base architectures.

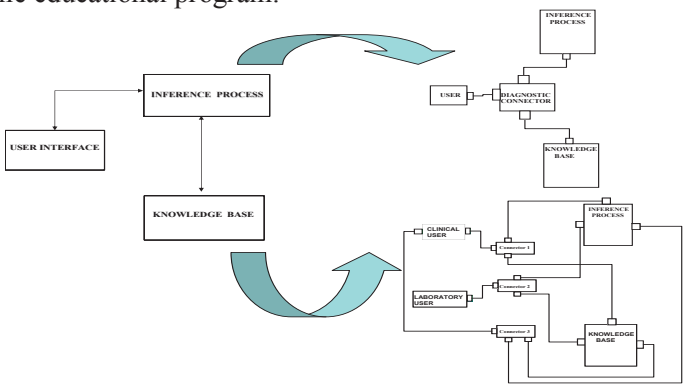
In these cases, there are two variability types which are orthogonal: one provided by the domain (e.g. diagnosis) and another one provided by the application domain (e.g. medical diagnosis, educational diagnosis).

Therefore, we have treated variability in two phases. The initial variability is treated through variability points in a Decision Tree (DT). This DT selects the right base architecture. The second variability is managed by decorating this base architecture with the features of the application domain.

To illustrate our approach, we have selected a case study: the diagnosis of educational programs within the diagnosis domain. Our research considers the following ontology:

- The diagnosis consists of interpreting the status of an entity through its properties.

- In the application domain of the diagnosis, the entity to be diagnosed is a postgraduate academic program, and the diagnosis result is the level of development of this program. The properties of the entities belong to the features and sub-features that are evaluated to obtain the validation of only one hypothesis (appropriate/inappropriate) as a result of the diagnosis.
- The scenario for the academic diagnosis process is as follows: the system takes data that are given as input by the final user and relates it to the features of the educational program. The system infers other properties by deduction and subsequently does the same for the hypothesis. The result is the development level of the educational program.



(a) Modular View (b) Component-Connector View

Fig. 1. Visual metaphor of (a) a unique Generic Architecture, and (b) two Base Architectures

3. The variability in the domain: V_1

The first type of variability involves a family of base architectures. This SPL must be constructed and stored on the Baseline. The Baseline is then itself a SPL, formed by all the assets necessary to construct the base architectures.

Some assets are templates (or architectural element skeletons) and the configuration information for the base architectures. Thus, when we configure the base architectures, we obtain the first SPL.

This first type of variability represents the variability information of the domain (properties of the entity that participate in the diagnostic process, property levels, types of the reasoning, and hypotheses), and the variability in the final user requirements (number of use cases, actors, and use cases by actor).

The features of the first variability type are present in the Feature Model (FM) of our SPL (Figure 2). These features are represented as variability points in a DT. This DT allows the acces to the assets necessary to build base architectures (i.e. the architectural elements, their feature insertion processes, the base architectural model configuration, the feature insertion main process, the Application Domain Conceptual Model (ADCM), and the RAS model [11] of assets). These assets are the leaves of the DT (Figure 2). The FM and the DT act as variability mechanisms that manage the variability at the artefact level.

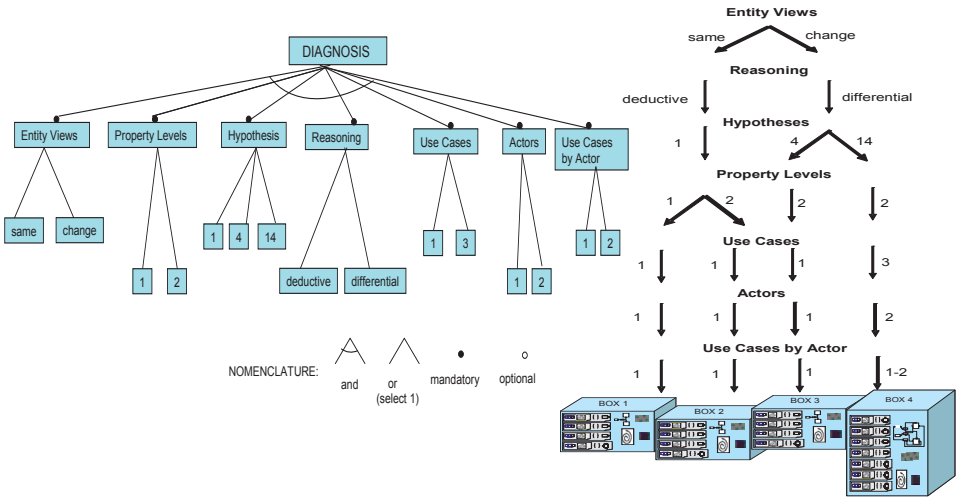


Fig. 2. Diagnostic Feature Model and its Decision Tree

We have detected seven types of features (or variability points) that are present in the FM and the DT: 1) *entity views*: an entity can be characterized by the same properties (the same view), or have different properties (different views) during the diagnostic process. 2) *property levels*: the properties of the entities can have *n* different abstraction levels. The deduction rules relate these properties between levels. 3) *number of hypotheses*: the goal of the diagnostic process is a single hypothesis. There can be one or several candidate hypotheses, which must all be validated in order to obtain the unique and correct hypothesis. 4) *reasoning types*: reasoning shows the way in which the rules are applied by the inference motor in order to infer a final diagnosis. The reasoning types can be: deductive reasoning (driven by data), inductive reasoning (driven by goals), and differential reasoning (establishing the difference between two or more diagnostic possibilities). 5) *number of use cases*: a use case indicates the division of the system based on its functionality; i.e., the different operations of the systems and how the system interacts with the environment (final users). 6) *number of actors*: represents the number of final users of the system. 7) *use cases per actor*: each final user can access different use cases.

An example of these features applied to our educational program diagnosis is: the entity views are the same during the diagnostic process, there are two property levels, one hypothesis, deductive reasoning, one use case, one actor, and one use case per actor. With this information, the assets that will form the base architecture of this case study are selected (Figure 2).

The variability shown in the use cases, actors, and use cases per actor is reflected in the construction of the assets of the architectural element skeletons and in the base architecture in the following way: i) there is one connector that joins all the architectural component assets for each use case, ii) the number of ports of the Inference Process component is the same as the number of use cases, iii) the number of ports of the Knowledge Base is the number of use cases, iv) the number of User Interfaces is the number of actors of the use cases, v) the number of ports of the User Interface is the number of use cases that can be accessed by an actor.

The domain variability management could be viewed as a transformation taking as input the Domain Conceptual Model (DCM) and the Decision Tree Model, and producing as output the corresponding Base Architecture Model (we call it T_1).

The structure of the architectural elements involves features on three variability points in the DT. These features are used to select assets that let to configure the base architectures.

The behaviour of the architectural elements involves the features of four variability points in the DT: entity views, property levels, number of hypotheses, and reasoning types. These features are related to the services, the protocols and the played_roles of the aspects from the PRISMA architectural elements (the information of state, process, and roles, respectively).

To optimize the feature insertion process, instead of repeating this information in each PRISMA type, we have placed it in the skeltons of the Baseline.

4. Variability in the application domain: V_2

The second type of variability involves the SPL of the application in a specific field. This variability allows to the base architectures to be enriched or decorated with the application domain features.

In the process of the application variability management, the variations of the specific requirements of the application domain should be selected. This selection is made by means of an ADCM. The features are inserted (by means of QVT-Transformations [12]) in the base skeletons in order to generate the types of the PRISMA software artifacts. These PRISMA architectural elements will be used to configure the PRISMA architectural model of the application. That model is automatically compiled into code (C#) using the PRISMA Model Compiler [13], and it is executable over the PRISMA NET Middleware [14].

The features that are used to fill the skeletons to obtain PRISMA types that make up the PRISMA base architectures of our SPL are:

- name and type of the entity properties, that will be diagnosed,
- name and type of the hypotheses,
- the rules that relate the entity properties.

In the following, we present an example of these features (F) in our educational program diagnosis:

- Features of the properties of level 0 (FP.0) = {student_population, graduation_rate, graduation_time, graduate_quality, control_school, critical_mass, academic_training, scientific_productivity, groups_lines_research, general_design, pupil_control, research_experience_students, number_courses, scheduled_duration, library, computer_equipment, laboratories, building&facilities, general_services }
- Features of the properties of level 1 (FP.1) = { student_body, faculty, curriculum, infrastructure&services }
- Features of the hypotheses (FH) = developmental_stage
- Features of the rules of level 1 (FR.1) = {student_population=”good” and graduation_rate=”good” and graduation_time=”good” and graduate_quality=”good” and control_school=”good” } student_body =”good”

- Features of the rules of level 2 (FP.2) = { student_body =" good" and faculty =" good" and curriculum =" good" and infrastructure&services=" good" } developmental_stage = "consolidated"

The features considered as constants represent the base models (in our case, the skeletons). For example, S-IP-EPD, wich means the "S-IP-EPD program". It is a constant feature. S-IP-EPD is the nomenclature used for Inference Process Skeleton of the Educational Program Diagnosis.

The skeletons selected will be filled or decorated with the features of the application domain in order to create the respective types. These types will include the features as functions, which are refinements of the base models. These models are extensions of the base model, which is taken as input. For example, F ● S-IP-EPD, wich means: "insert the F feature into the S-IP-EPD model", where ● is the application of the F function. The successive decoration process with features will be represented as:

$S\text{-IP-EPD}_0 = FP.0 \bullet S\text{-IP-EPD}_1 = FP.0 \bullet (E\text{-MI- DPE}_1)$
$S\text{-IP-EPD}_1 = FP.1 \bullet S\text{-IP-EPD}_2 = FP.1 \bullet (E\text{-MI- DPE}_2)$
$S\text{-IP-EPD}_2 = FH \bullet \cdot S\text{-IP-EPD}_3 = FH \bullet (E\text{-MI- DPE}_3)$
$S\text{-IP-EPD}_3 = FR.1 \bullet S\text{-IP-EPD}_4 = FR.1 \bullet \cdot (E\text{-MI- DPE}_4)$
$S\text{-IP-EPD}_4 = FR.2 \bullet S\text{-IP-EPD}_5 = FR.2 \bullet \cdot (E\text{-MI- DPE}_5)$

Figure 3 represents a visual metaphor of the feature insertion process in a skeleton selected from the Baseline ($S\text{-IP-EPD}_0$) to create its PRISMA type.

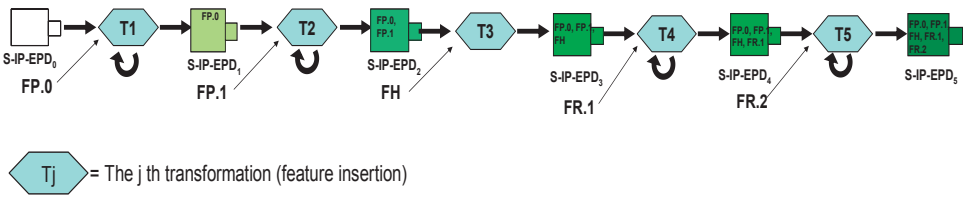


Fig. 3. Feature insertion process in a skeleton to create its PRISMA type

It is important to state that a Skeleton-Base Architecture can be instantiated to one or more PRISMA-Base Architecture types (i.e. several products of our SPL), when the decorating features are different. In two case studies performed by the authors (the diagnosis of the developmental_stage of educational programs, and the diagnosis of TV video quality), they shared the same skeleton, but each one of them has different PRISMA types. This was due to the fact that different properties of the application domain were inserted in each case.

As an example of this, we present the functional aspect skeleton of the Knowledge Base and the PRISMA type. Both the skeleton and the type are specified in PRISMA-ADL (Architecture Description Language).

Table 1. PRISMA-ADL of the Functional Aspect of the Knowledge Base (The different section holes are depicted in bold type)

Functional aspect of the Knowledge Base of a skeleton	**Functional aspect of the Knowledge Base of a PRISMA type**
Functional Aspect FBaseEPD using IDomainDPEDT Attributes	Functional Aspect FBaseEPD using IDomainDPEDT Attributes

Variables **<Features FP.0>,**	Variables **laboratories:string,** **library:string,** **critical-mass:string,** **scientific_productivity:string;**
Derived **<Features FP.1>,** **<Features FH>,**	Derived **infrastructure&services:string,** **faculty:string** **developmental_stage:string;**
Derivations **<Features FR.1>** **<Features FR.2>**	Derivations **{laboratories="good" and library="good"}** **infrastructure&services:="good"** **{critical_mass="good" and scientific_productivity ="good"} faculty:= "good"** **{infrastructure&services="good" and faculty= "good"}** **developmental_stage:= "consolidated";** .....
Services ……. Played_Roles ..... Protocols ……. End_Functional Aspect FBaseEPD	Services ……. Played_Roles ..... Protocols ……. End_Functional Aspect FBaseEPD

5. Designing Decision Software Architectures: Product Engineering Phase

Figure 4 shows the transformations involved in the design of the decision software architectures. This figure illustrates how the two transformations performed at the model level are applied in the transformation process. Transformations T_1 and T_2 are executed at the model level (M_1 in OMG-MOF [15]) shown in Figure 4, they are defined as relations in QVT-Relational at the metamodel level (M_2 in OMG-MOF).

In transformation T_1, the Component-Connector (C-C) Skeleton model (c) is obtained from the DSS modular model (a) and the DCM (b). In transformation T_2, the PRISMA architecture model (e) is obtained from the Skeleton C-C model generated in T_1 (c) and the ADCM (d).

The relations R1 and R2 that specify the corresponding transformations T_1 and T_2 are: R1 $\underset{\text{def}}{\subseteq}$ MM MODULAR VIEW $\times$ MMV_1 $\rightarrow$ MM C-C SKELETON ;

R2 $\underset{\text{def}}{\subseteq}$ MM C-C SKELETON $\times$ MMV_2 $\rightarrow$ MM PRISMA VIEW ;

Where MMV_1 and MMV_2 are the metamodels (MM) of the DCM and the ADCM. We use the UML metamodel [16] for both.

The relations R_1 and R_2 have been specified using QVT-Relational. One of them is (R_1 relations): MODULAR VIEW $\longrightarrow$ C-C VIEW

In BOM, the stakeholders only introduce the variability data by means of the Conceptual Models: In the fisrt step, the stakeholders introduce the variability of the

domain by means of the DCM capturing the domain variability $V_{1.}$. T_1 will obtain a base architecture "ad hoc" to the case study using the Generic Architecture Modular View. In the second step, the stakeholders introduce the variability of the application domain by means of the ADCM capturing the application variability $V_{2.}$. T_2 will generate a PRISMA architecture model as a product of our SPL (application engineering), using the Skeleton C-C- model generated in the first step. The two conceptual models conform to their respective metamodels (MM): the UML metamodel.

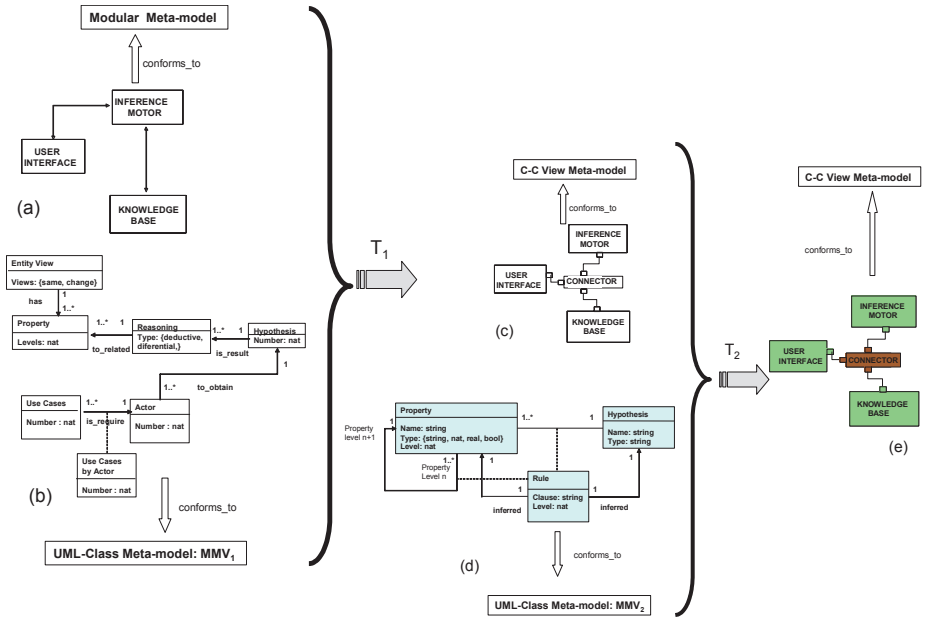


Fig. 4. The transformation of models in BOM

5.1 Specification of the relations between the software views

The relations between the software views are specified by means of a MOF diagram, and the code for these relations is written in the QVT-relational. Code and diagrams are shown in [17]. In order to gain clarity the prefix "skeleton" will be omitted from now, i.e "skeleton component" will be written as "component", and "skeleton connector" will be written as "connector". The relations between the software views are the following:

i) Relation moduleToComponent. This relation maps each module with a component. Relation moduleToComponent has two types of relations: the checkonly type and the enforce type. The object of the Module domain is of the checkonly type. In contrast, the object of the Component domain is of the enforce type. This creates an object of the Component class that is related to the Module class. The where clause indicates a call to the functionToService relation, which relates an object of the Module class with an object of the Component class.

ii) Relation functionToService. This relation implies that a function of the module metamodel will generate a service of the Component class. When this occurs, the type of the Function will generate a port. In the where clause, the name of the port is

obtained by calling the function typePort. When the relation is executed, the classes that are in the source metamodel can only be verified and the classes that are in the target meta-model will be created (only the ServiceToPort class).
iii) Relation rUseModToConnector. The 'uses' relation is transformed from the modular meta-class to a link between a connector and two components in the C-C meta-class. The creation of objects for this relation is from 1 to n because a relation of two components is generated through a connector. In [17] is shown part of the code to illustrate how the relations are invoked in the 'where' clause to create the relations among module, component, and functionToService.
iv) Relation rCompositionModToComp. The set of modules will also generate a set of components. However, in this case when a component is created (container) a subordinate is created inside it.

6. Conclusions

The development of DSS is complex because there is variability in the architectural elements that conforms them as well as variability in their final architecture. This situation produces several base architectures in our SPL, sharing a unique generic architecture.

Futhermore, the complexity problem of these kind of systems is not solved by means of a unique Feature Model and the insertion of its features. Since the variability management is the essence of SPL, we have taken a new approach. This approach manages the variability in two stages, wich correspond to the development of two SPL: 1) the base architectures SPL that shares a generic architecture, and 2) the application SPL in a specific domain, that shares a base architecture.

In this context, we describe how the variability is managed in our SPL by means of our BOM framework. BOM automatically generates DSS in a specific domain using SPL.

BOM captures the data that characterize the domain variability and the application variability in conceptual models. DT and FOM exploit this data in the domain engineering and product engineering phases in order to obtain a specific application by means of Model Transformation techniques. In BOM, the variability appears in the construction of the DCM (which is represented as a DT showing the different variation points). The base assets are selected by the DT to configure a base architecture. These assets are enriched by the specific application features (given in the ADCM) by a process that results in PRISMA types (a product of our LPS).

In the domain engineering phase, the user constructs the different assets and stores them in the Baseline. This Baseline is used in the application engineering phase by a production plan in order to obtain the final product. The production plan is one of the assets stored in the Baseline.

We can conclude that the main characteristics of BOM are the following: i) Variability is managed at the model level rather than at the program level. ii) Systems variability is modeled using conceptual models independently of their functional models. The DSLs for expressing the variability are suited for the domain, instead of adding tangled variability annotations directly to the functional models (UML, ADLs) as other approaches have proposed [9]. iii) Variability is operated by two orthogonal

types: one provided by the features of the domain (e.g. diagnosis), and another one provided by the features of the application domain. iv) Various technological spaces are integrated to cope with the complexity of the problem.They are the current trends in Software Engineering. v) BOM implements a generic approach to SPL developement, that can be applied to different domains, application domains and systems. In this paper, the BOM framework is applied to the diagnostic domain. Other domains will be considered in the near future, e.g. interpretation, and prediction.

A prototype of the BOM framework: ProtoBOM has been implemented and will be used in real case studies. In the future, we will use benchmarks in order to compare BOM results with other approaches.

References

1 Turban, E., and Aronson, J.E., Decision Support Systems and Intelligent Systems, Prentice Hall, ISBN: 0-13-089465-6 (2001) 865 pages.

2. Model Driven Architecture. http://www.omg.org/mda.

3. Pérez J., PRISMA: Aspect-Oriented Software Architectures. PhD. Thesis of Philosophy in Computer Science, Polytechnic University of Valencia, Spain (2006).

4. Clements P. and Northrop L.M., Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison Wesley (2001).

5. Trujillo S., Feature Oriented Model Driven Product Lines. PhD. Thesis, The University of the Basque Country, San Sebastian, Spain (2007).

6. Batory D., Benavides D., and Ruiz-Cortés A., Automated Analyses of Feature Models: Challenges Ahead. ACM on Software Product Lines (2006).

7. González-Baixauli B. y Laguna M. A., "MDA e Ingeniería de Requisitos para Líneas de Producto", Taller sobre Desarrollo Dirigido por Modelos. MDA y Aplicaciones, Granada, España (2005).

8. Santos A.L., Koskimies K., and Lopes A., "Using Model-Driven Architecture for Variability Management in Software Product Lines", Ph Thesis Propasa, Facultade de Ciencias de la Universidade de Lisboa, Portugal (2005).

9. Aspect-Oriented, Model-Driven, Product Line Engineering Specific Targeted Research Project: IST-33710. AMPLE Project. http://www.ample-project.net

10. Bachman F., Goedicke M., Leite J., Nord R., Pohl K., Ramesh B., and Vilbig A, "A meta-model for representing variability in product family development", The 5th International Workshop on Product Family Engineering (2003) 66-80.

11. Reusable Asset Specification. http://www.omg.org/technology/documents/formal/ras.htm

12. Query/View/Transformations OMG Document ad/2005-07-1.

13.Cabedo R., Pérez J., Carsí J.A. y Ramos I., "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", IV Workshop DYNAMICA, Murcia, España (2005).

14. Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I., "PRISMANET: Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", X Jornadas de Ingeniería del Software y Bases de Datos, Granada, España (2005) 27-34.

15. Meta-Object Facility: http://www.omg.org/mof.

16. Unified Modeling Language 2.0: http://www.omg.org/docs/formal/05-07-04.pdf.

17. Limón Cordero R., Cabello Espinosa M.E., and Ramos Salavert I., "Establish Relations among Software Architecture Views through MDA for SPL". In Proceedings of the 14th International Congress on Computer Science Research, CIICC'07, Veracruz, México.

Acknowledgements. This work is funded by the Spanish Department of Science & Technology, under the National Program for Research, Development and Innovation, META project (TIN2006-15175-C05-01), and by the European Regional Development Fund.

The Baseline: the milestone of Software Product Lines for Expert Systems automatic development

Maria Eugenia Cabello, Isidro Ramos
Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain
{mcabello, iramos}@dsic.upv.es

***Abstract*—In this paper we present the Baseline of our Software Product Line. We explain the construction of the Baseline at the domain engineering stage, and demonstrate how it is used at the application engineering phase. We describe how to create the stored software assets (represented by models). In our approach, the Baseline contains both the assets and the processes for building applications in a specific domain. In order to exemplify our Baseline Oriented Modeling framework and show how to automatically generate Expert Systems, we have chosen the diagnosis as a case study.**

***Index Terms*—Baseline, Expert systems, Software architectures, Software product lines.**

I. Introduction

NOWADAYS Software Engineering promotes automation in software development and mechanisms to avoid complexity in software production. However, the automatic generation of software systems is only possible if there is a framework that supports the process. The Model-Driven Architecture (MDA) [1] strategy advocates the use of standards and platform independence in the software development process as a new way of generating applications automatically.

Furthermore the development of Expert Systems (ES) [2] applications has acquired great importance in recent years and there is a need to provide proper support for them. The ES captures the knowledge of experts and attempts to imitate their reasoning processes when the experts solve problems in a specific domain. Such systems usually have a generic architecture with components that are independent and in separate units. Control is data-independent. The input and output of the information is carried out through the user interface. However, these systems are complex because their architectural elements vary.

In order to cope with the variability problem, Software Product Lines (SPL) [3] have emerged in an effort to control and minimize the high costs of the software development process and to reduce the time to market of these new products. This approach is based on having a generic design that is shared by all product family members.

Our target platform is the PRISMA framework [4] providing expressivity for the specification of software architectures with aspects at a design level. It offers good properties and advantages in the construction of complex, distributed, evolutionary, and re-usable architectural models and can be used in the ES domain.

In this context, we have developed the Baseline Oriented Modeling (BOM) framework. BOM automatically generates ES (as PRISMA architectural models) based on SPL, by using the MDA approach.

Our work integrates various technological spaces. We use SPL techniques in order to control and minimize the time to market and high costs of the software development process. We use the PRISMA framework in order to obtain the advantages of integrating aspects and reusable architectural elements. ES has been our case study. ES implementation separates the inference process from the knowledge domain, and incorporates several reasoning strategies in order to solve a problem by applying the most efficient strategy. We use the MDA approach in order to automatically generate code from models by means of transformation rules.

However not everything is automated; user interaction is required to provide information to the system. BOM's user develops a specific ES when he/she inputs the variability information of a particular domain into the system using domain conceptual models.

The engineering process of our SPL, according to [5], has two phases: the domain engineering phase and the application engineering phase. In the domain engineering phase we create the models and processes that are stored in a repository: the

Manuscript received May 15, 2008. This work has been funded under the Models, Environments, Transformations, and Applications: META project TIN20006-15175-605-01.

M. E. Cabello is with the Department of Information Systems and Computation, Universitat Politècnica de València, Camino de Vera s/n, E 46022 Valencia, Spain (phone: 34-963879350, fax: 34-963879359, e-mail: mcabello@dsic.upv.es).

I. Ramos is with the Department of Information Systems and Computation, Universitat Politècnica de València, Camino de Vera s/n, E 46022 Valencia, Spain (phone: 34-963879350, fax: 34-963879359, e-mail: iramos@dsic.upv.es).

Cabello, M. E., Ramos, I. The Baseline: the milestone of Software Product Lines for Expert Systems automatic development. Proc. of the 9th Mexican International Conference on Computer Science. 2008.

Baseline (as reusable resources or assets). The assets and processes will be used in the application engineering phase to produce the final application.

The objective of this paper is to present the construction and use of the Baseline modeled at a high abstraction level.

The paper is structured as follows: Section 2 presents the related work. Section 3 presents our case study (the ES and the diagnostic domain). Section 4 introduces our Baseline, and provides a description of how to construct and use in the Production Plan of our SPL. Section 5 presents our conclusions and provides some ideas for future work.

II. Related Work

SPL is nowadays a hot topic and there are a wide variety of research results offering suggestions and solutions in specific domains. Our research is related to the following work:

Batory et al. [6] capture the domain features in the Feature Model (FM). In BOM we capture the FM's features in a decision tree and in conceptual models.

Trujillo [7] uses Feature Oriented Programming (FOP) as a technique for inserting features into XML documents by means of XSLT templates. In BOM we use this technique but at the model level, i.e. we use Feature Oriented Modeling (FOM). The features are inserted in the skeleton aspects of the architectural elements that make up the PRISMA models.

Clements et al. [5] use the SPL development approach, and establish a division between domain engineering and application engineering phases for the reuse and the automation of the software process. In BOM, we used this approach in the development of our SPL.

Ávila-García et al. [8] use the OMG standards SPEM and RAS to package reusable assets. In our approach we use both SPEM and RAS when we model our Baseline as repository of the assets, and the Production Plan of the SPL.

Bachman et al. [9] propose to separate the variability and the functionality description of the affected artifacts. In BOM, variability and functionality are specified in separate conceptual models. Using instances of such conceptual models the user inputs the domain features as variants to select assets of the Baseline or to define the application domain features in order to configure the final application.

III. Our field study: diagnostic Expert Systems

We have carried out a field study to learn about the structure and functionality of ES and we have selected diagnosis as the application domain. A number of case studies (medical diagnosis, educational diagnosis, video diagnosis, disaster victim diagnosis) have been studied.

A. Our case study: diagnosis

The analysis carried out on the diagnostic process has caused the emergence of domain variability. We have elicited and modeled the user requirements as UML-Use Case diagrams. Based on these results, we can conclude the following: diagnosis consists of an interpretation of the states of the entities involved (viewed as a set of properties), followed by the identification of the problems using rules.

Variability in the diagnostic process can be described in terms of its features as follows: i) Entity views: an entity can be considered to participate with the same properties (the same view), or to have different properties (different views) during the diagnostic process. ii) Property levels: the properties of the entities can have n different abstraction levels. The rules that are used to relate the properties of the entities have n-1 levels, where n is the level of the entity properties, iii) Number of hypotheses: the goal of the diagnosis is a single validated hypothesis. There can be one or several candidate diagnostic hypotheses which must be evaluated in order to select the valid hypothesis. iv) Reasoning types: show the ways in which the rules are applied by the inference motor in order to infer a final diagnosis.

There is variability in user requirements too, and this can be described in terms of its features as follows: i) Number of use cases: operations of the system and how it interacts with the environment (final users). ii) Number of actors: number of final users of the system. iii) Use cases per actor: an actor can access different use cases.

However, the features of the application domain are also necessary. Therefore another variability emerges: the application domain variability The features that correspond to the specific application domain are: i) Name and type of the entities' properties by abstraction level. ii) Rules by abstraction levels (where the entity properties are related through these rules). iii) Level, name and type of the hypotheses used in the diagnostic process.

B. Our domain functionality: Expert Systems

Based on the field study carried out in the ES, we have detected that a unique generic architecture captures the functionality of these systems. This generic architecture has three basic modules [16]: i) the module that contains the inference process that resolves a problem in a specific domain, ii) the module that contains the knowledge about the domain, and iii) the module that allows communication between the user and the system.

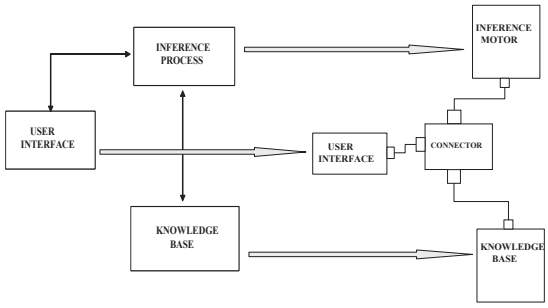
But as we implement the SPL using PRISMA software architectures, the modules of the ES generic architecture must be mapped into the following architectural elements: The Inference Process Component which establishes system control and provides the general resolution strategy for taking a decision; The Knowledge Base Component which contains the domain knowledge of the case study using rules that define the application domain rules (Horn clauses) and facts (information unchanged); The User Interface Component which establishes the man-machine interaction by allowing communication between the users and the system. However to

be consistent with the PRISMA metamodel, it is necessary to incorporate a new architectural element (the connector), to establish the communication among the components; therefore the other architectural element is the Coordinator Connector, which contains the choreography of the decision process.

This implementation is a mapping between the two models with a small semantic gap. As is shown in Figure 1 (using gray arrows), there is a correspondence among the modules and their respective components. The connections between components and connectors are the attachments.

However, we have detected that the base architecture model implementing the generic architecture model is not unique; it varies in its behavior, and since we use PRISMA models, in its structure too. This implies the creation of different base architecture models, every time we start to develop a product line.

Furthermore, we conclude that variability in structure in the ES is related to the features associated with user requirements, and variability in the behavior of the ES is related with the associated process.



(a) (b)

Fig. 1. Mapping between the generic architecture (a), and a base architecture (b).

C. BOM: The framework

After the conclusion of the field study, the variability management in BOM includes two types of orthogonal variabilities: The first variability (V1) has as variants the diagnostic process features and the end user requirement features, which determine a specific base architecture. The second variability (V2) corresponds to the application domain features, which determine the final product of our SPL.

Since BOM is a modeling oriented approach, we manage the variabilities as follows:

The features of the first variability are represented in the Feature Model. These features are represented as variability points and their variants define a Decision Tree. The variants shown in the Feature Model and the Decision Tree are captured as instances of the Domain Conceptual Model (DCM).

The features of the second variability, i.e. variability of the application domain, are captured as instances of the Application Domain Conceptual Model (ADCM).

For this reason, the modeling of the system is separated into two stages:

A) Constructing the DCM (capturing in its instances the variants of the domain: V1) and the model of the functionality of the system (i.e. a unique generic architecture associated with the ES). This information allows the obtaining (through decision tree techniques) of the corresponding base architectural model (as a template or skeleton of the system in the ADL used: ADL-PRISMA).

B) Constructing the ADCM conceptual model capturing in its instances the application variants. The ADCM's instances together with the functional model of the system (i.e. the base architecture selected in 1) are transformed using FOM to a PRISMA architectural model, i.e. the final product of the SPL, i.e. the ES, which is compliant with the two variabilities defined.

All these models and the processes (as reusable resources or assets) are stored in a repository: the Baseline. The Baseline and its assets are also modeled at a high abstraction level.

IV. The Baseline

Usually the Baseline assets capture pieces of the functionally of the application domain. However, in BOM we store these functional assets together with the processes required to produce a product of the SPL, defining how to build applications in a domain, i.e. our Baseline is a repository where we store the assets and the "know-how" of how to use them.

In this section we present the construction and use of the Baseline according to [5], in two phases: the domain engineering and the application engineering phases. In the domain engineering phase we create the Baseline that will be used in the application engineering phase to obtain the final application.

We use two OMG standards: the Reusable Asset Specification (RAS) [10] which identifies, describes and packs assets in an standard way, and the Software Process Engineering Metamodel (SPEM) [11] which defines the standard language for modeling the software process.

A. Domain engineering: building the Baseline

We present below all the software artifacts created by the domain engineer in order to obtain the Baseline. These software artifacts are built as Computation Independent Models (CIM) or Platform Independent Model (PIM). The artifacts a) and b) are CIMs, and other artifacts are PIMs.

a) The Feature Model: The Feature Model identifies the first variability in BOM.

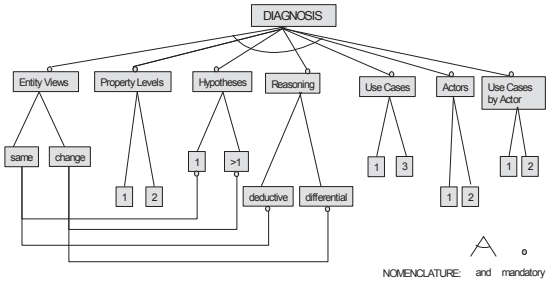


Fig. 2. Diagnostic feature model.

b) The Decision Tree.- The features included in the Feature Model are shown in the variability points of the Decision Tree. The variability points of the first variability are represented in the nodes of the Decision Tree, and its leaves points to the asset families of the SPL.

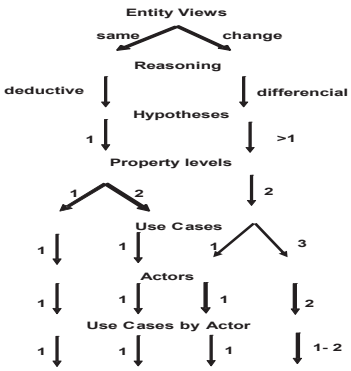


Fig. 3. Diagnostic decision tree.

c) The Domain Conceptual Model (DCM).- The DCM captures in its instances the variants of the Feature Model. Decision Tree techniques are used to query the Baseline and obtain the Base Architecture software assets. The instances of this model are created by the domain engineer capturing the domain variability during the Application Engineering phase.

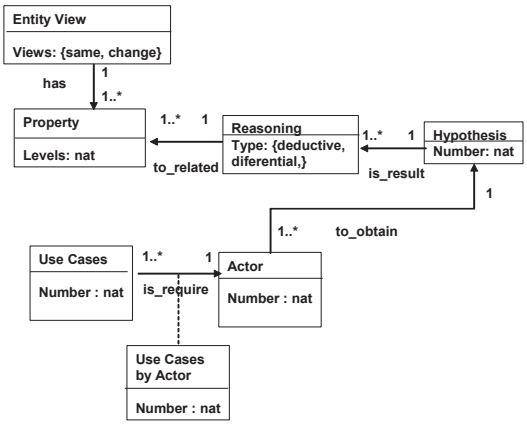


Fig. 4. Domain conceptual model.

d) The Application Domain Conceptual Model (ADCM).- This model captures the application domain variability. The instances of this model are created by the application engineer and they capture the specific application domain variants. The features are glued in the base architecture skeleton assets.

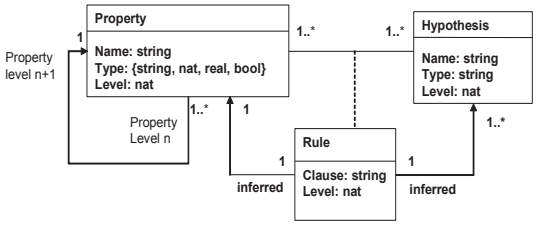


Fig. 5. Application domain conceptual model.

e) The Asset Selection Process.- This process computes the paths in the Decision Tree which are used to select the assets in the Baseline. This process is created by the domain engineer and uses the Decision Tree and the assets of the Baseline. This process is executed by BOM in the application engineering phase, when the application domain engineer instantiates the DCM (i.e. inserting the specific domain information into the system as variants of the domain variability points).

f) Skeleton or Template Assets.- These software artifacts are PRISMA ADL's aspect templates (i.e. incomplete PRISMA aspect with feature holders). The skeleton aspects are used in the application engineering phase when BOM glues the selected skeletons with the application domain features (i.e. the ADCM instances). A skeleton (or template) is a PRISMA type with feature holders (holes). These holes will be filled with application domain features (i.e. domain entity properties, hypotheses, and rules) mapped as attributes (variable and derived) of the PRISMA aspect type.

Below we present the functional aspect skeleton of the Knowledege Base component part of the educational program diagnosis architecture. It is specified in the PRISMA-ADL.

```
Functional Aspect FBaseED using IDomainED
  Attributes
    Variables
      <FP.0>
    Deriveds
      <FP.1>,    Derivation   <FR.1>
      <FH>,      Derivation   <FR.2>

  Services
   ...

  Played_Roles
   …

  Protocols
   …
End_Functional Aspect FBaseED,
```

Where <Fx.i> are the features place holders. These features are the following: FP.0 = Features of the properties

of level 0, FP.1 = Features of the properties of level 1, FH = Features of the hypotesis, FR.1 = Features of the rules of level 1, FR.2 = Features of the rules of level 2,

g) The Interface, Architectural Element, and Architectural Model PRISMA types.-The interfaces, the architectural elements, and the architectural model, do not have features. For this reason these artifacts are directly PRISMA types. These PRISMA artifacts are used in the application engineering phase when BOM configures the architectural model.

h) The Main Feature Insertion Process.- ´This process is the main process controlling the insertion of the application domain features, and it invokes the individual feature insertion processes associated with each skeleton aspect.

i) The Feature Insertion Processes.- Each one of these processes is used to insert the application domain features in their specific skeleton aspect. These processes are invoked by the main feature insertion process.

We present the algorithm to insert the application domain features in a skeleton aspect (in order to obtain the corresponding PRISMA type aspect). The features are modeled as functions, representing refinements of the input model. In the educational program diagnosis, we have:

$$Fx.i \cdot \bullet \ S\text{-}KB\text{-}EDj,$$

that means "add feature Fx.i to the S-KB-ED model", where $S\text{-}KB\text{-}ED_0$ is the Skeleton-KnowledgeBase Educational Domain, j= 0,...n and "•" denotes the application of the function.

The algorithm is the following:

$S\text{-}KB\text{-}ED_1 = FP.0 \bullet \cdot S\text{-}KB\text{-}ED_0$
$S\text{-}KB\text{-}ED_2 = FP.1 \bullet S\text{-}KB\text{-}ED_1$
$S\text{-}KB\text{-}ED_3 = FH \cdot \bullet \cdot S\text{-}KB\text{-}ED_2$
$S\text{-}KB\text{-}ED_4 = FR.1 \bullet S\text{-}KB\text{-}ED_3$
$S\text{-}KB\text{-}ED_5 = FR.2 \bullet S\text{-}KB\text{-}ED_4$

j) The Aspect-Process Skeletons.- Each one of these is a skeleton aspect and its feature insertion process. These assets are executed in the application engineering phase to decorate the skeletons and create the PRISMA types.

k) The Packaged Hybrids.- A packaged hybrid is a new packaged asset, formed by an aspect-process skeleton, its interface, and its PRISMA type architectural element. These assets are stored in the Baseline.

l) The Base Architecture Model Configuration.- This asset captures the manner in which to configure the selected base architecture. i.e. the PRISMA configuration program, used to configure the PRISMA software architecture by instancing the PRISMA types. This artifact is stored in the Baseline. In the application domain phase, BOM takes this artifact as input in order to compile the final architectural model

m) The RAS Models of the Assets.- An RAS Model stores the information of each one of the assets contained in the Baseline: the identifier of each asset, its type, the description of each asset, and the variability points associated with the assets.

n) The Kit-Boxes.- The assets selected by the domain engineer to build the SPL families sharing common features (from the domain variability) are packaged as a new composite asset: the Kit-Box asset. Therefore, the Kit-Boxes have all assets and know-how for building a specific ES and are the recovery units. A visual metaphor of a Kit-Box is shown in Figure 6.

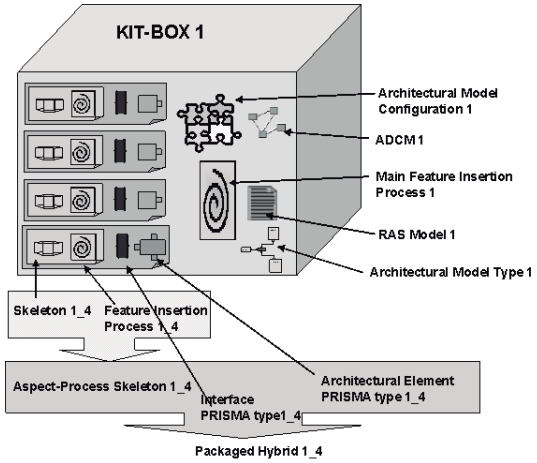


Fig. 6. An example of kit-box.

o) The Production Plan of the SPL.- This process shows the production life cycle of our SPL. In the domain engineering phase this process is specified in SPEM, and it is inserted in the Baseline as an asset. In the application engineering phase, the Production Plan is executed to generate an ES as one SPL product.

p) The Baseline.- The Baseline as repository of all the assets necessary to obtain a SPL product, is structured as a set of Kit-Boxes and the Production Plan (asset) of the SPL. Therefore, the Baseline has the necessary know-how to produce the SPL. The Baseline is used in the application engineering phase to produce the assets (Kit-Boxes) that correspond to the specific domain products. A visual metaphor of the Baseline is shown in Figure 7.

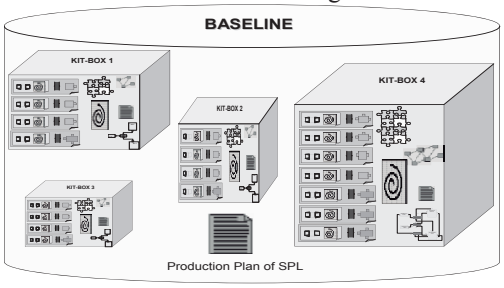


Fig. 7. The baseline.

B. Application engineering: using the Baseline

In the application engineering phase the user executes the Production Plan through BOM. The Production Plan describes the process by means of tasks or activities to obtain a final product of the SPL. This process is modeled in the SPEM notation. It is shown in Figure 8.

The Production Plan starts when BOM obtains (from the application engineering) the features expressed in the DCM's instances of the variability points of the first variability.

Next, BOM selects the assets from the Baseline, i.e. one Kit-Box asset that corresponds to the specific product.

The Kit-Box that is selected by the engineer using BOM must be unpackaged in order for each asset to be used.

One of the assets recovered from the Kit-Box is the ADCM, which is used by BOM in order to obtain (from the application engineer) the features of the application domain considered as variants of the second variability.

Other assets recovered from the Kit-Box are the Packaged Hybrids. These assets are in turn unpacked to produce the PRISMA type aspects. BOM applies the Main Feature Insertion Process (which invokes the individual features insertion processes) to fill the selected skeletons (aspects) with the specific features of the case study defined by the engineer, thereby creating the PRISMA type aspects.

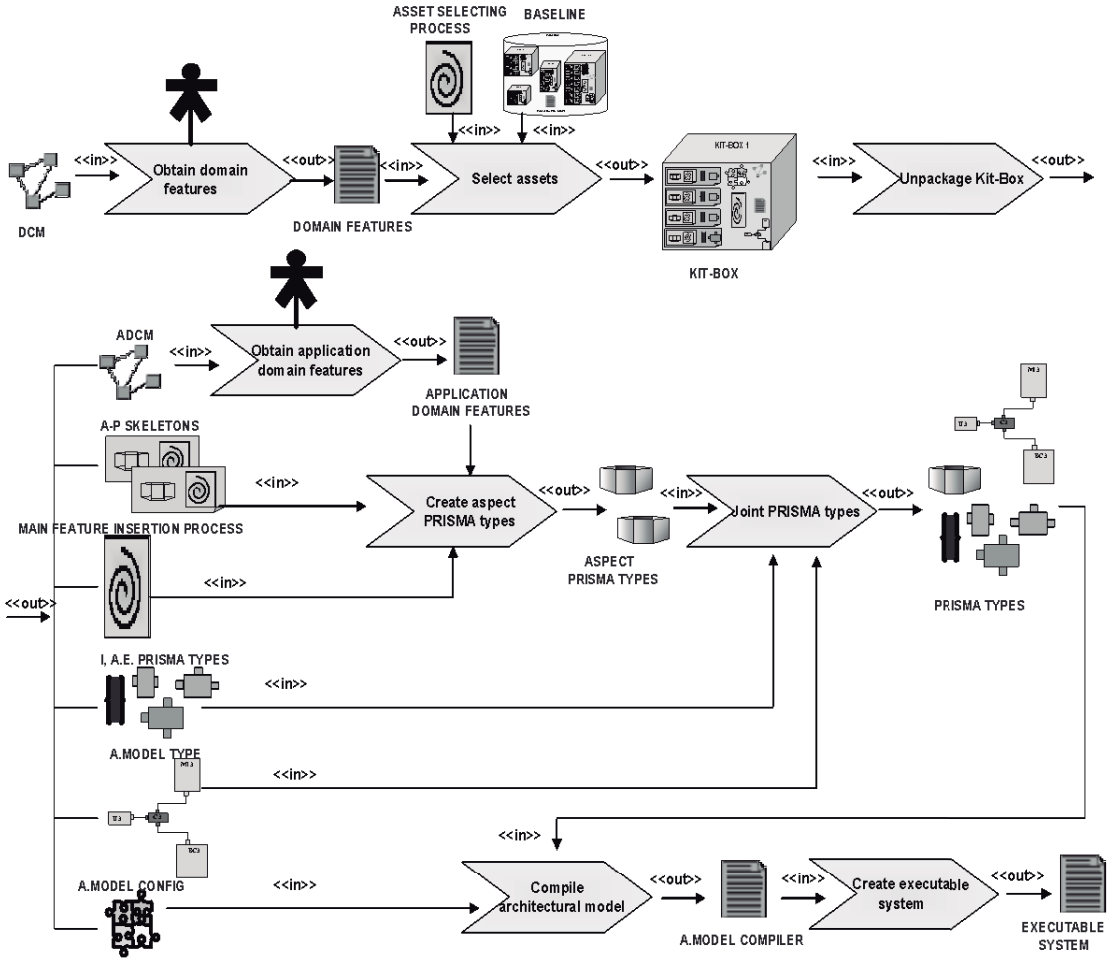


Fig. 8. Production plan of our SPL

Below we present an example of the functional aspect type of the Knowledege Base component from the educational program diagnosis, it corresponds to the functional aspect skeleton described in section 4.1 It is specified in the PRISMA-ADL.

```
Functional Aspect FBaseED using IdomainED
  Attributes
    Variables
   laboratories: string,
   library: string,
   critical-mass: string,
   scientific_productivity: string;
    ……….
  Deriveds
   infrastructure&services: string,
    Derivations
   {laboratories=”good” and
    library=“good”}   infrastructure&services:=“good”,

   faculty: string,
    Derivations
    {critical_mass=”good” and scientific _productivity =“good”}
   faculty:= “good”,
……

   developmental_stage: string;
    Derivations
    {infrastructure&services=“good” and faculty= “good”}
   developmental_stage:= “consolidated”,
.....
Services
……
 Protocols
…….
End_Functional Aspect FBaseED
```

BOM takes the aspects PRISMA types and the other PRISMA type artifacts (interfaces architectural elements, and architectural model) to joint all the PRISMA types to be used in the next task.

Another asset that has been unpacked from the Kit-Box is the Base Architectural Model Configuration. This artifact and the PRISMA type artifacts are sent to the PRISMA-MODEL-COMPILER tool [12]. BOM works over this tool to automatically generate the code (in .C# .NET) from the software architecture obtained from the preceding task.

An example of (part of) the generated C# code which corresponds to the Knowledge Base component of the educational program diagnosis case study is as follows:

```
namespace KBMD
{  [Serializable]
   public class KnowledgeBaseEducational: ComponentBase
    {public class
       KnowledgeBaseEducationalDiag  string name:base(name)
       { AddAspect (new FBaseED ( ) );
         InPorts.Add(“KnowPort”,“IDomainED”,“KNOW”);
         OutPorts.Add(“KnowPort”,“IDomainED”,“KNOW”);
    }
      }
}
```

At the end, BOM creates the final diagnostic system as an executable application, i.e. an instance of the SPL. This application is executed on top of the PRISMANET Middleware [13].

In BOM, the user (application engineer) builds an ES (product of the SPL), by giving as input the features of the variabilities V1 and V2 using instances of the two conceptual models: DCM and ADCM.

We have recently developed a BOM prototype, called ProtoBOM [14]. Figure 9 shows the interface generated by ProtoBOM in order to obtain the domain features (instances of the DCM).

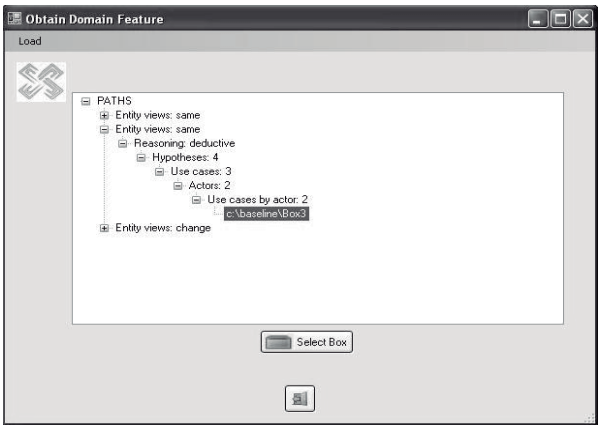


Fig. 9. Obtaining domain features through ProtoBOM.

Figure 10 shows the interface generated by ProtoBOM in order to obtain the properties of the application domain features (instances of the ADCM).

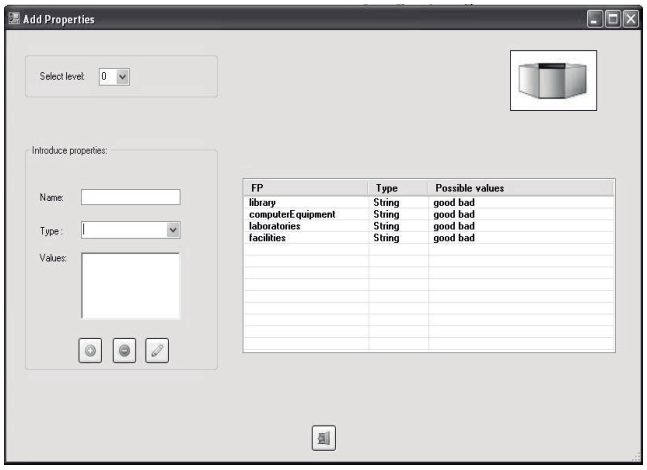


Fig. 10. Obtaining properties of the application domain features through ProtoBOM.

Figure 11 shows the interface generated by ProtoBOM in order to obtain the hypotheses of the application domain features (instances of the ADCM).

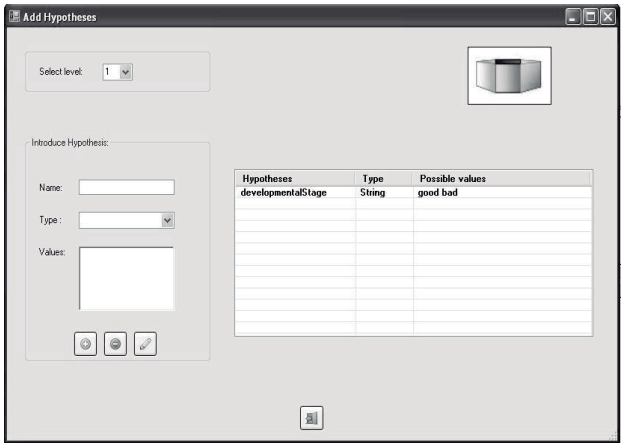


Fig. 11. Obtaining hypotheses of the application domain features through ProtoBOM.

Figure 12 shows the interface generated by ProtoBOM in order to obtain the derivation rules sets of the application domain features (instances of the ADCM).

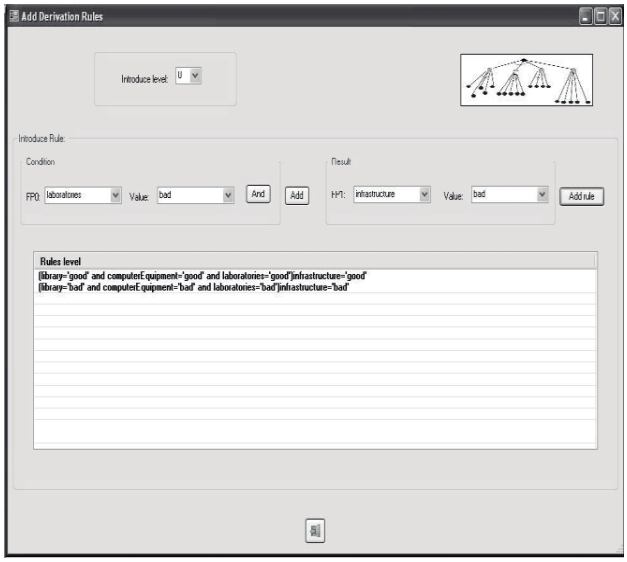


Fig. 12. Obtaining derivation rules sets of the application domain features through ProtoBOM.

V Conclusions and Future Work

Our BOM approach automatically generates ES in a specific domain following an engineering process in two phases: the domain engineering and application engineering phases. The domain engineer creates the Baseline as a repository of all the necessary assets to build an ES. The application engineer provides information on the specific domain in the Production Plan execution, thus obtaining the final ES. The Baseline is used in the application engineering phase when BOM selects the assets (within the production plan) to build the architectural model of a specific domain application.

In this paper we have presented as an example the diagnostic domain. In the future, other domains will be considered. A prototype of the BOM framework (ProtoBOM) has been implemented and will be used in real life case studies and different application domains in order to validate the approach and to obtain feedback.

References

[1] MDA: http://www.omg.org/mda
[2] Turban, E., and Aronson, J.E., Expert Systems and Intelligent Systems, Prentice Hall, 2001, 865 pages, ISBN: 0-13-089465-6.
[3] SPL: http://www.softwareproductlines.com
[4] Pérez J., PRISMA: "Aspect-Oriented Software Architectures", PhD. Thesis of Philosophy in Computer Science, Universitat Politècnica de València, Spain, 2006.
[5] Clements P., and Northrop L.M., Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison Wesley, 2002.
[6] Batory D., Benavides D., and Ruiz-Cortés A., "Automated Analyses of Feature Models: Challenges Ahead", ACM on Software Product Lines, 2006.
[7] Trujillo S., "Feature Oriented Model Driven Product Lines". PhD. Thesis, The University of the Basque Country, San Sebastian, Spain, 2007.
[8] Ávila-García O., García A. E., Rebull V. S., y García J. L. R., 2006. Integrando modelos de procesos y activos reutilizables en una herramienta MDA, XI JISBD´06, Barcelona, España, 2006. (in spanish).
[9] Bachman F., Goedicke M., Leite J., Nord R., Pohl K., Ramesh B., and Vilbig A, "A meta-model for representing variability in product family development", The 5th International Workshop on Product Family Engineering, 2003, pp. 66-80.
[10] RAS: http://www.omg.org/technology/documents/ formal/ras.htm
[11] SPEM: http://www.omg.org/docs/ad/06-06-02.pdf
[12] Cabedo R., Pérez J., Carsí J.A. y Ramos I., "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", IV Workshop DYNAMICA, Archena, Murcia, España, 2005.(in spanish)
[13] Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I., "PRISMANET: Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", X JISBD´05, Granada, España, 2005, pp. 27-34. (in spanish)
[14] Cabello M.E., Gómez M., LLavador M, and Ramos I. "ProtoBOM: a framework that semi-automatically generates Expert Systems based on Software Product Lines". Technical report: DSIC II/02/08, Universitat Politècnica de València, 2008, pages 68.



María Eugenia Cabello Espinosa (México City, 1958) received her Bachelor degree in Physics Sciences from the Universidad Nacional Autónoma de México (México City), Master in Sciences of the Computation from the Fundación Arturo Rosenblueth (México City), and Master in Software Engineering, Formal Methods and Information Systems from Universitat Politècnica de València (Valencia, Spain).

She is currently working on her PhD degree in Declarative Programming and Engineering of the Programming in the Universitat Politècnica de València (Valencia, Spain). Her research interests include: expert systems, diagnosis systems, component-based and aspect-oriented software architectures, software product lines, and model-driven architectures. Her.previous publications are:

Limón R., Cabello M. E., and Ramos I., "Establish Relations among Software Architecture Views through MDA for SPL". In Proceedings of the 14th International Congress on Computer Science Research-CIICC'07, Veracruz, México, pp- 175-187, nov. 2007, ISBN 13 978-970-95771-0-5.

9

Cabello M. E. and Ramos I., “A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines”. In Proceedings of HEALTHINF 2008, International Conference of Health Informatics-INSTICC, Madeira, Portugal, Vol. 2, pp. 237-246, jan. 2008, ISBN 978-989-8111-16-6.

Cabello M. E. and Ramos I., “Variability Management in Product Lines for Decision Support Systems”. International Conference on Enterprise Information System-ICEIS, jun. 2008, Barcelona, Spain (accepted).

M.C. Cabello is membership of the Engineering Software and Information Systems research group in the Universitat Politècnica de València (Valencia, Spain), and membership of the Software Product Lines Spanish Network (ESLEPES).



Isidro Ramos Salavert (Valencia, Spain, 1943) received his Bachelor degree and PhD degree in Physics Sciences from the Universidad Complutense de Madrid (Madrid, Spain), and received his Bachelor in Computer Science from the Universidad Complutense de Madrid (Madrid, Spain).

He has been teaching during 41 years in several universities. Currently, he is a full professor at the Universitat Politècnica de València (Valencia, Spain) and the leader of the Software Engineering and Information Systems research group of the same university. He is currently working in MDE and in Software Architectures fields.

Dr. Ramos has been a key researcher in 21 Research Projects, member and president of different Programme Committees of relevant conferences. Dr. Ramos has also been the International Coordinator of the VII CYTED Subprogram on Applied Electronics and Computing, the Founder President of the spanish Society of Software Engineering (SYSTEDES), and the President of the University of Castilla la Mancha. Dr. Ramos is membership of the Software Product Lines Spanish Network (ESLEPES). Dr. Ramos has recently received the Spanish Computer Science National Award 2006.

BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA

María Gómez[1], Abel Gómez[1], Mª Eugenia Cabello[2], Isidro Ramos[1]
[1] Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, España
[2] Universidad de Colima, Ave. Universidad 333, 28040 Colima, México
magolac@fiv.upv.es, agomez@dsic.upv.es, ecabello@ucol.mx, iramos@dsic.upv.es

Resumen. Este documento presenta BOM–Lazy, una aproximación para desarrollar Sistemas Expertos mediante la utilización de técnicas de Desarrollo de Software Dirigido por Modelos y Líneas de Producto Software. Se ha realizado un estudio sobre la variabilidad de los Sistemas Expertos para determinar las características generales y particulares de dicho dominio. La variabilidad de tal dominio se gestiona mediante una transformación de modelos que permite obtener automáticamente diferentes arquitecturas base a partir de la arquitectura genérica de la Línea de Productos Software.

Palabras clave: Líneas de Productos Software, Gestión de Variabilidad, Transformación de Modelos, Arquitecturas Software, Sistemas Expertos.

1 Introducción

Una clase de sistemas que está cobrando gran interés en los últimos tiempos son los Sistemas Expertos (SE) [1]. Un SE es un programa software que permite simular el comportamiento de un especialista humano frente a un problema de su competencia en un determinado campo. Estos sistemas intentan codificar los conocimientos y reglas de decisión de los especialistas, incrementando la calidad y rapidez en las respuestas, dando así lugar a una mejora de la productividad del experto. La aplicación de los SE ha sido extensa en muchos ámbitos de la vida humana, algunos de los campos de aplicación son: medicina, educación, inteligencia militar, transportes, aeronáutica, agricultura, arqueología, derecho, geología, industria electrónica, informática y telecomunicaciones. Por ello, los SE empiezan a tener cada vez mayor auge, hasta el punto de ser un punto de referencia importante en la toma de decisiones, siendo de notable interés en sistemas que realizan tareas de diagnóstico. Pero el desarrollo de este tipo de sistemas es complejo, dado que los elementos básicos que conforman su arquitectura varían tanto en su comportamiento como en su estructura. Por consiguiente, existe la necesidad de soportarlos adecuadamente.

La naturaleza cambiante de la tecnología lleva a necesitar en cortos periodos de tiempo múltiples versiones de la misma o parecida aplicación. Por ello, la Ingeniería del Software debe proporcionar herramientas y métodos que permitan desarrollar una familia de productos con distintas capacidades y adaptables a situaciones variables, en contraposición a desarrollar un único producto. Ante esta situación, surge el concepto de Línea de Productos Software (LPS) [2] con la finalidad de controlar y minimizar los altos costes del desarrollo de software. Esta aproximación se fundamenta en la creación de un diseño que puede ser compartido por todos los miembros de una

Gómez, M., Gómez, A., Cabello, M. E.,Ramos, I. BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. Actas del VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). 2009.

familia de programas (la arquitectura genérica). De esta manera, un diseño hecho explícitamente para un producto se beneficia del software común que puede ser usado en diferentes productos, reduciendo los costes y el tiempo para construirlos. Para el desarrollo de SE se considera más adecuado el enfoque de LPS que el enfoque tradicional, puesto que este tipo de sistemas involucran una amplia gama de áreas de aplicación, con características que difieren de un caso de estudio a otro. Además las LPS facilitan el desarrollo de los productos en distintas plataformas y su uso en distintas tecnologías.

Este trabajo integra diferentes espacios tecnológicos [3]: Sistemas Expertos (SE), Líneas de Producto Software (LPS) y la iniciativa *Model-Driven Architecture* (MDA) [4] propuesta por el *Object Management Group* (OMG) como soporte tecnológico para integrarlos. MDA se basa en la separación de la descripción de la funcionalidad del sistema (*modelo independiente de plataforma*, PIM) de su implementación en plataformas software específicas (*modelo específico de plataforma*, PSM). MDA propone definir y usar modelos a diferentes niveles de abstracción como los principales activos del proceso de desarrollo de software. Sobre estos modelos se pueden realizar manipulaciones (transformaciones de modelos), para refinarlos de forma sucesiva y obtener el sistema software final. El estándar propuesto por el OMG para la definición de estas transformaciones se denomina *Query / View / Transformations (QVT)* [5]. De esta manera, en nuestra aproximación, que llamamos BOM–Lazy emplearemos todo el utillaje tecnológico proporcionado por MDA para representar y gestionar la variabilidad de la LPS mediante la construcción de modelos de dominio y su transformación en modelos arquitectónicos.

La estructura de este documento es la siguiente: la sección 2 presenta la variabilidad de los SE a través de diferentes casos de estudio. La sección 3 describe cómo es gestionada la variabilidad en nuestra aproximación. La sección 4 explica la implementación realizada y la sección 5 presenta las conclusiones.

2 Variabilidad en los Sistemas Expertos

Para mostrar nuestra aproximación se ha elegido un subconjunto dentro de los sistemas expertos: aquellos que se emplean para realizar diagnósticos. El diagnóstico de una entidad consiste en interpretar el estado de ésta a través de sus propiedades. Se ha realizado un estudio de campo [6] sobre los sistemas de diagnóstico para identificar las características de su variabilidad. Este estudio ha permitido conocer el comportamiento y la estructura de los SE en varios dominios específicos. A lo largo del artículo se emplearán dos casos que resultarán paradigmáticos: los sistemas empleados para el *diagnóstico médico* y los sistemas para el *diagnóstico educativo.*

En el caso del diagnóstico médico la entidad a diagnosticar es el paciente y el resultado es la enfermedad que padece. En el proceso se realiza un diagnóstico clínico, que debe coincidir con el diagnóstico de laboratorio, para finalmente obtener un diagnóstico integral con la participación de ambos. Así, se distinguen tres funcionalidades básicas del sistema «realizar diagnóstico de laboratorio» (1), «realizar diagnóstico clínico» (2), y «obtener resultados del diagnóstico» (3). La primera de ellas (1) es empleada por el llamado «técnico de laboratorio», mientras que las dos

últimas (2 y 3) se emplean por parte del «médico». En este caso, las propiedades de las entidades cambian durante el proceso del diagnóstico,lo que conlleva la existencia de varias hipótesis que deberán ser validadas para obtener la más adecuada, mediante un razonamiento diferencial.

En el caso del diagnóstico educativo la entidad a diagnosticar es un programa educativo de postgrado en el que se evalúan diversos criterios de calidad, y el resultado del diagnóstico es el nivel de desarrollo que tiene dicho programa. Las propiedades de las entidades no cambian durante el proceso del diagnóstico por lo que se genera una sola hipótesis aplicando un razonamiento deductivo. En este caso, el sistema experto sólo realiza la acción «obtener nivel de desarrollo» invocada por el usuario que emplee la herramienta.

2.1 Arquitectura genérica de los Sistemas Expertos

En las LPS existen partes comunes y partes variables de los productos. La parte común está representada por la arquitectura genérica, que capta la funcionalidad compartida. La parte variable involucra las características particulares adicionales que definen los productos concretos, representados por las arquitecturas base.

La arquitectura genérica de los Sistemas Expertos viene dada por un modelo modular compuesto tradicionalmente por tres módulos básicos (Fig. 1):

- El módulo «*Inference Motor*», queestablece el proceso de inferencia y toma decisiones.
- El módulo «*Knowledge Base*» que contiene el conocimiento del dominio de aplicación.
- El módulo «*User Interface*» que establece la interacción hombre-máquina.



Fig. 1. Arquitectura genérica de los Sistemas Expertos

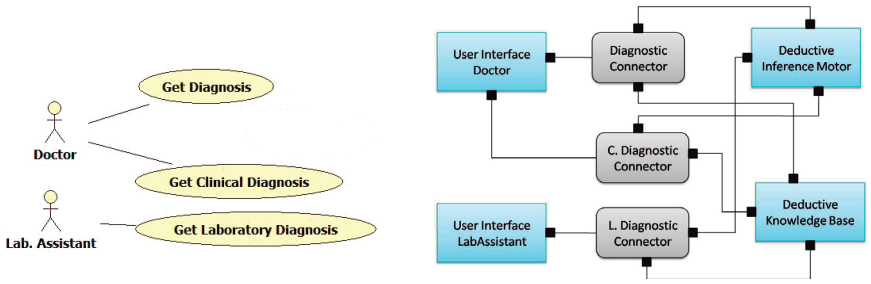
En el proceso de desarrollo de una aplicación concreta miembro de la LPS, ésta se deriva a partir de la arquitectura genérica, pero existen características particulares adicionales que definen el producto concreto. Esto implica la creación de una arquitectura base específica cada vez que se desarrolla un producto de la línea. Sin embargo se ha detectado que la arquitectura base que implementa una arquitectura genérica no es única puesto que este tipo de sistemas varían tanto en estructura como en comportamiento como se explica a continuación.

2.2 Variabilidad en la estructura de los Sistemas Expertos

Para mostrar que los elementos arquitectónicos de un SE varían en su estructura, se han modelado los requisitos funcionales que el producto final ha de satisfacer mediante un diagrama de casos de uso de UML. Estos diagramas muestran las

distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno. En particular, la estructura de los elementos arquitectónicos varía según el número de casos de uso considerados, así como el número de actores y de los casos de uso a los que accede un actor.

En la Fig. 2 se muestra el diagrama de casos de uso del dominio del diagnóstico médico junto con su correspondiente arquitectura base.



(a) (b)

Fig. 2. Diagrama de casos de uso del diagnóstico médico (a) y arquitectura base (b).

Como se observa en la Fig. 2.a, el sistema experto para el diagnóstico médico de nuestro caso de estudio tendrá dos actores, *doctor* y *asistente de laboratorio*. El primero de ellos empleará la aplicación para obtener un diagnóstico clínico así como un diagnóstico final; y el segundo de ellos empleará la aplicación para obtener el diagnóstico de laboratorio. Estos casos de usos inciden en la arquitectura base final de la aplicación (Fig. 2.b) donde dispondremos, por ejemplo, de un módulo de interfaz de usuario por cada actor de nuestro diagrama.

2.3 Variabilidad en el comportamiento de los Sistemas Expertos

La variabilidad en el comportamiento entre los elementos arquitectónicos de un SE depende del dominio de aplicación y el tipo de razonamiento usado. Como se presentó en el apartado anterior, el razonamiento puede ser *deductivo* o *diferencial* y determinará el proceso de inferencia a seguir. El proceso de inferencia es estático si las entidades tienen las mismas propiedades durante todo el proceso del diagnóstico y una sola hipótesis. Por el contrario si las propiedades de una entidad cambian durante el proceso del diagnóstico y existen diferentes hipótesis, el proceso es dinámico. De

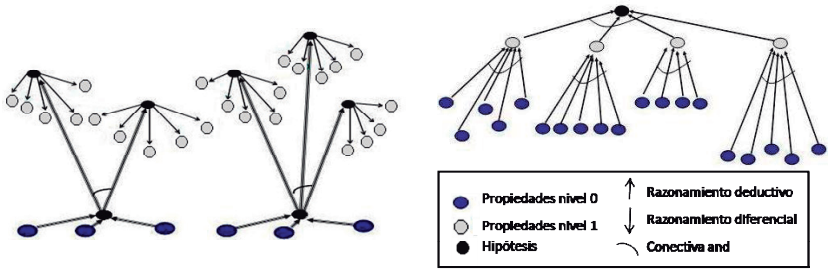


Fig. 3. Grafos descriptivos de los procesos de inferencia del diagnóstico médico (izquierda) y diagnóstico educativo (derecha).

esta manera, el diagnóstico médico requiere un razonamiento diferencial y un proceso de inferencia dinámico (Fig. 3.a); mientras que el diagnóstico educativo requiere un razonamiento deductivo y un proceso de inferencia estático (Fig. 3.b) [7]. Así, por ejemplo, la arquitectura base para los sistemas expertos del dominio del diagnóstico médico incorporarán un componente «*Differential Inference Motor*» cuyo comportamiento diferirá del de los sistemas expertos de la familia del diagnóstico educativo, que incorporarán un componente «*Deductive Inference Motor*». En la Fig. 3. se muestran los procesos de inferencia del diagnóstico médico y educativo a través de un grafo.

3 Gestión de la Variabilidad

La gestión de la variabilidad es la base para el desarrollo de los SE. La variabilidad entre los productos de una LPS puede ser expresada en términos de características [8]. En BOM–Lazy (Baseline Oriented Modeling–Lazy) las características observadas en el proceso del diagnóstico y los requisitos del usuario son considerados como puntos o fuentes de una primera variabilidad, y adicionalmente deben de ser consideradas las características del campo de aplicación a través de una segunda variabilidad, tratando así la variabilidad en dos fases. La primera variabilidad es tratada a través de puntos de variabilidad plasmados en un *Modelo de Características*, el cual permite diseñar la arquitectura base concreta. La segunda variabilidad es manejada decorando una determinada arquitectura base con las características del dominio de aplicación, lo cual determina el producto final. En lo que resta de documento se describirá únicamente la gestión de la primera variabilidad, que está relacionada con el proceso del diagnóstico y los requisitos del usuario (determinando el comportamiento y la estructura del SE como se describe en la sección 2), siendo la gestión de la segunda variabilidad un proceso semejante pero partiendo de artefactos más refinados.

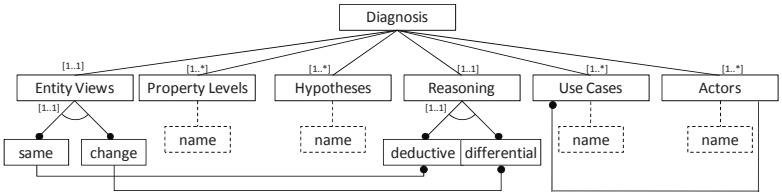


Fig. 4. Diagrama de Características de la LPS.

Las características de la primera variabilidad se representan en el *Modelo de Características* de la LPS del diagnóstico y son captadas como instancias en un *Modelo Conceptual del Dominio* (MCD). Estas características son utilizadas para configurar las arquitecturas base. La Fig. 4 muestra el *Modelo de Características* de la LPS de los SE de diagnóstico utilizando una variante de la notación extendida de Czarnecki-Eisenecker [9], donde se permite la clonación de características. En este caso, además, la notación ha sido enriquecida con el uso de atributos que sirven para identificar de forma unívoca las características que han sido clonadas.

De esta manera, el manejo de la variabilidad del dominio puede abordarse mediante una transformación de modelos, que recibe como entrada una instancia del

modelo de la variabilidad del dominio y el modelo de la arquitectura genérica de los SE, produciendo como salida el correspondiente modelo de la arquitectura base.

4 Implementación

En esta sección se presenta el desarrollo de SE mediante técnicas de transformación de modelos para el tratamiento de la variabilidad. En BOM–Lazy la transformación involucrada en la construcción de las arquitecturas base de la LPS se denomina «T1», y viene determinada por la *primera variabilidad* apuntada en la sección 3. Esta transformación genera una arquitectura base concreta (el modelo componente–conector), a partir del modelo modular (la arquitectura genérica, que describe las partes comunes de los sistemas expertos) y una instancia del modelo de variabilidad que captura la variabilidad del dominio concreto (MCD), como muestra la Fig. 5.

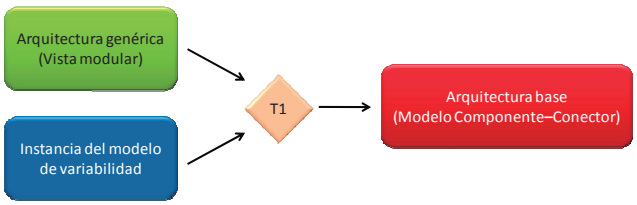


Fig. 5. Elementos involucrados en la transformación T1.

La Fig. 6 muestra el metamodelo simplificado de la vista modular empleado para representar las arquitecturas genéricas. En él se muestra que un modelo contiene un conjunto de módulos (quienes a su vez pueden contener diferentes funciones), que se vinculan a otros módulos mediante una serie de relaciones —en este caso sólo se muestra la relación de *Uso* por simplicidad—.

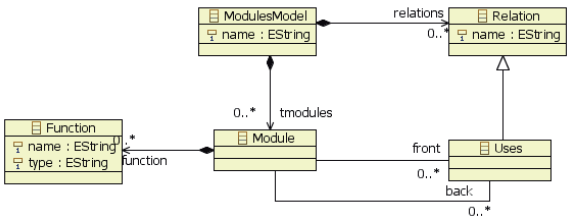


Fig. 6. Metamodelo simplificado de la vista modular.

Para desarrollar la familia de productos concretos dentro de la línea de los SE de diagnóstico debe configurarse la arquitectura genérica con las características particulares de dicho dominio. Esto se realiza mediante el modelo de características presentado en la sección 3, que finalmente se ha representado mediante un diagrama de clases de UML por simplicidad y para facilitar su uso en las herramientas de modelado. La Fig. 7 muestra cómo se representa el modelo de características del dominio de los SE para el diagnóstico (Fig. 4) según esta representación basada en los trabajos realizados en [9]. Así, una instancia de nuestro modelo de variabilidad se corresponderá con un conjunto de objetos que serán instancias de dicho modelo.

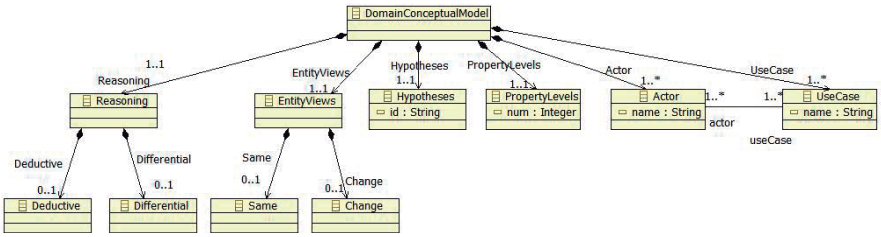


Fig. 7. Modelo de variabilidad expresado en términos de un diagrama de clases.

Por último, la Fig. 8 muestra el metamodelo simplificado de Componente–Conector, donde un modelo contiene un conjunto de componentes y un conjunto de conectores. Los componentes proporcionan una serie de servicios a través de un conjunto de puertos, y los conectores se encargan de conectar mediante sus roles los diferentes puertos entre diversos componentes.

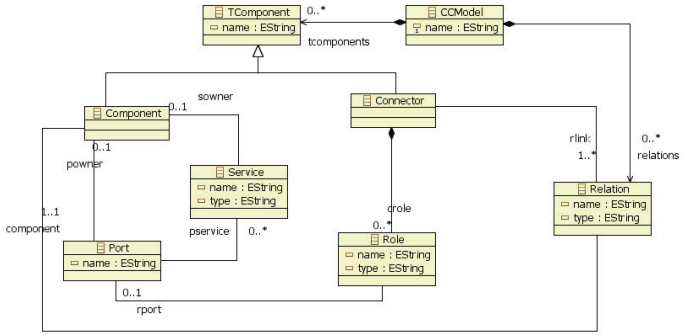


Fig. 8. Metamodelo simplificado de Componente-Connector.

4.1 Patrones de transformación

Para definir las relaciones entre metamodelos, y manteniéndonos dentro del marco de MDA, hemos empleado el lenguaje Query/Views/Transformations (QVT) [5] propuesto por el OMG. La transformación establece correspondencias entre los elementos de los modelos origen y los elementos del modelo destino. La tabla 1 muestra de forma resumida las relaciones de correspondencia identificadas junto con los elementos implicados en cada una de ellas.

Tabla 1. Correspondencias entre elementos

RELATION	*CLASES INVOLUCRADAS EN LA TRANSFORMACIÓN*		
	Modelo Variabilidad	*MM Modular*	*MM C-C*
ModulesModel2ComponentsModel	-	ModulesModel	CCModel
UseCase2Connector	UseCase	-	Connector
Module2Component	Actor EntityViews / Reasoning	Module	Component
Module2RolePort	UseCase	Module	Role, Port
Function2Service	-	Function	Service
Function2Relation	-	Function	Relation

Para establecer las correspondencias en las relaciones se han tenido en cuenta *buenas reglas de diseño software* [10] con el objetivo de incrementar la calidad del diseño de la arquitectura generada en la transformación. Algunos criterios y decisiones de diseño descritos por las reglas de transformación son los siguientes:

- *ModulesMode2ComponentsModel.— «Un diseño debe presentar una organización jerárquica que haga un uso inteligente del control entre los componentes del software».* Con el objetivo de satisfacer este criterio, la regla transforma el elemento raíz del metamodelo modular (*ModulesModel*), contenedor del resto de elementos que conforman el metamodelo, en el elemento raíz del metamodelo de Componente–conector (*CCModel*) asignándole el mismo nombre.
- *UseCase2Connector.— «El diseño debe ser modular, es decir, se debe hacer una partición lógica del Software en elementos que realicen funciones y subfunciones especificas».* Los casos de uso constituyen una división del sistema basada en funcionalidad. Por tanto, para cumplir este criterio, la regla transforma cada caso de uso en un conector coordinador que une a todos los componentes de ese caso de uso. Se estable así una relación uno a uno entre casos de uso y conectores.

  En la Fig. 9 se muestra la implementación (mediante QVT gráfico) de la regla *UseCase2Connector*. Esta regla crea, por cada uno de los casos de uso del modelo de variabilidad, un conector en el modelo Componente–Conector, tomando el nombre del caso de uso seguido de la palabra «*Connector*». El dominio modular (*mdomain*) y el dominio del modelo de variabilidad (*dcmdomain*) son de tipo «*checkonly*» (indicado con una «*C*» adjunta a la flecha del dominio). Este modificador comprueba que al menos existe algún objeto que valida el patrón definido en el dominio. Si no existe tal objeto, y puesto que la transformación se ejecuta en el sentido del dominio *ccdomain*, la aplicación de la regla no produce efectos en tales dominios. Por su parte el dominio componente–conector (*ccdomain*) es de tipo «*enforce*» (marcado con una «*E*»). Este modificador provoca que el patrón del dominio deba poderse aplicar siempre. En caso de que no haya objetos que validen el patrón, la regla deberá crear un objeto tal que sí lo cumpla, haciendo cierta la aplicación de la regla. Por último, la cláusula «*where*» indica que la relación *Module2Component* es postcondición de la regla *UseCase2Connector*, y que ésta última se debe poder aplicar correctamente tras la aplicación de *UseCase2Connector*.
- *Module2Component.—* La arquitectura genérica de los SE está formada típicamente por tres módulos. Esta regla transforma los módulos *Base de Conocimiento* y *Motor de Inferencia* en los componentes *Base de Conocimiento* y *Motor de Inferencia* respectivamente, estableciendo su tipo (deductivo o diferencial) que viene dado por los elementos *Reasoning* y *EntityViews* del modelo

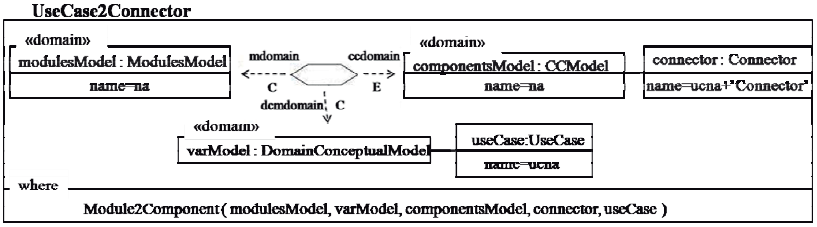


Fig. 9. Representación gráfica de la regla de transformación UseCase2Connector.

de variabilidad. Cualquier otro módulo adicional de la arquitectura genérica dará lugar a un componente que lo represente con el su mismo nombre en la arquitectura base final.

Con el objetivo de satisfacer el siguiente criterio de diseño: *«El diseño debe conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y el entorno exterior»*, el módulo *Interfaz de Usuario* será transformado en tantos componentes *Interfaz de Usuario*, como actores aparezcan en el modelo de variabilidad. De esta forma se establece una relación uno a uno entre actores en el modelo de entrada e interfaces en el modelo de salida.

- *Module2RolePort.*— Con la finalidad de satisfacer el criterio de diseño: *«Unicidad de la base de conocimiento*», que indica que el conocimiento debe ser único para todo el sistema, en nuestra arquitectura base existe un único componente *Base de Conocimiento*. Por cada caso de uso existirá una vista distinta de la *Base de Conocimiento*, se realiza la unión de las vistas en una mediante la adición de puertos a la *Base de Conocimiento* que la unen con cada conector a través de diferentes roles. Esta regla crea en cada componente un puerto y su correspondiente rol en cada conector. Por tanto, se establece una relación uno a uno entre casos de uso y puertos en la *base de conocimiento*, lo que implica una relación uno-a-uno también con los roles de los correspondientes conectores. Esta regla también crea los puertos en los componentes *Motor de Inferencia* e *Interfaces de Usuario* y sus correspondientes roles en los conectores.

5 Conclusiones

Para ilustrar la aproximación se ha elegido el dominio de los Sistemas Expertos de diagnóstico. El desarrollo de este tipo de sistemas es complejo puesto que los elementos que componen su arquitectura varían tanto en comportamiento como en estructura. Esta situación produce varias arquitecturas base en la LPS que comparten una arquitectura genérica. La aproximación presentada en este documento utiliza *QVT–Relations* como lenguaje de transformación de modelos para la gestión de la variabilidad de la LPS. Esta aproximación mejora el desarrollo de los SE aplicando técnicas de LPS al construir un diseño que comparten todos los miembros de una familia de programas. De esta manera, un diseño específico puede ser usado en diferentes productos, reduciendo costes, tiempos de producción, esfuerzo y complejidad. La aplicación de técnicas de MDA en el desarrollo, permite la construcción de sistemas independientes de plataforma, abordados desde la perspectiva del problema y no de la solución, lo que proporciona aplicabilidad en diferentes dominios. Además se ofrece una funcionalidad conjunta y coordinada de todos los espacios tecnológicos contemplados, que constituyen las técnicas actuales más avanzadas en desarrollo de software. Cabe destacar que, aunque en este artículo sólo se ha descrito la gestión de la variabilidad de la primera fase (como se comenta en la sección 3), tras la obtención de la arquitectura base el proceso de desarrollo continúa, decorándola y obteniendo una arquitectura PRISMA [11]. PRISMA es un modelo arquitectónico orientado a aspectos que dispone de la herramienta PRISMA-MODEL-COMPILER [12] que permite generar automáticamente código ejecutable en C# .NET. De esta manera, esta propuesta abarca el proceso de desarrollo completo.

Por otra parte, en aproximaciones tradicionales, el conjunto de arquitecturas base se define e implementa en la fase de diseño de la LPS (ingeniería del dominio) y permanece inmutable a lo largo del ciclo de vida de la aplicación de la LPS (ingeniería de la aplicación). En la aproximación de BOM–Lazy el empleo de la transformación T1 permite trasladar el proceso de creación de las arquitecturas base a la fase de ingeniería de la aplicación. Esto permite definir el conjunto de arquitecturas base mediante una serie de reglas que codifican patrones de buenas prácticas de diseño, así como otras decisiones de diseño de forma genérica, evitando tener que definir todas ellas de forma explícita. A medida que aumenta el tamaño de la LPS, BOM–Lazy se convierte en la aproximación más adecuada para abordar la gestión de la variabilidad, puesto que sería inviable tener construidas a priori las arquitecturas para todos los posibles productos de la LPS. Así, en nuestra aproximación, el principal esfuerzo realizado en la fase de ingeniería del dominio se realiza codificando y formalizando el conocimiento adquirido mediante un conjunto de reglas declarativas (el conocimiento se almacena de una forma explícita), y no en desarrollar de forma extensiva todas las posibles combinaciones de arquitecturas base (el conocimiento se almacena de una forma implícita). Esto redunda en una mayor eficiencia en la fase de ingeniería de la aplicación, donde partiendo del conocimiento almacenado de forma explícita se obtiene cada arquitectura precisamente en el momento en que realmente se necesita.

References

1. Giarratano, J., and Riley, G., *Expert Systems: Principles and Programming*. Fourth Edition (Hardcover). Ed. Course Technology. ISBN: 0534384471, 842 pages, Boston, 2005.
2. Clements P. and Northrop L.M., *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison Wesley, 2002.
3. Kurtev, I., Bézivin, J., Aksit, M.: *Technological Spaces: An Initial Appraisal*. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.
4. Model Driven Architecture: MDA. http://www.omg.org/mda.
5. OMG: *MOF 2.0 QVT Final Adopted Spec*., http://www.omg.org/docs/ptc/05-11-01.pdf
6. Cabello Ma. Eugenia., *BOM: Una aproximación MDA basada en Líneas de Producto para el desarrollo de aplicaciones*. Tesis doctoral, Universidad Politécnica de Valencia, 2008.
7. Cabello, M.E., Ramos, I. *Expert Systems development through Software Product Lines techniques*. In Information Systems Development: Towards a Service Provision Society. Papadopoulos, G.A., Wojtkowski, W. (et al.) (Eds.) Springer-Verlag. ISBN 978-0-387-84809-9. Due: Sept, 2009.
8. Kang K., Kim S., Lee J., Kim K., and Shin E. *FORM: A Feature Oriented Reuse Method with Domain Specific Reference Architectures*. Annals of Software Engineering, Vol. 5, 1998, pp. 143-168.
9. Czarnecki K. and Kim C. H. P. *Cardinality-based feature modeling and constraints: A progress report*. In International Workshop on Software Factories, San Diego, California, Oct 2005. http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf
10. Pressman, R.S., "*Ingeniería del Software. Un enfoque práctico*". 6ª ed. Mc Graw Hill 2006.
11. Pérez J., *PRISMA: Aspect-Oriented Software Architectures*. PhD. Thesis in Computer Science, Polytechnic University of Valencia, Spain, pages 397, Dec. 2006.
12. Cabedo R., Pérez J., Carsí J.A. y Ramos I., *Modelado y Generación de Arquitecturas PRISMA con DSL Tools*, Actas del IV Workshop DYNAMICA, Archena, Murcia, 2005.

Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for the Expert Systems development[1]

María Eugenia Cabello, Isidro Ramos, Abel Gómez, Rogelio Limón
Universitat Politècnica de València
{mcabello, iramos, agomez, rlimon}@dsic.upv.es

Abstract

This paper presents our Baseline Oriented Modeling (BOM) approach. BOM is a framework that automatically generates software applications as PRISMA architectural models using model transformations and Software Product Line techniques. We follow the Model-Driven Architecture initiative building domain models which are automatically transformed into Platform Independent Models, and then compiled to an executable application (i.e. Platform Specific Models). In order to illustrate BOM, we focus on a specific domain: the diagnostic Expert Systems.

1. Introduction

Nowadays Software Engineering promotes automation in software development and open mechanisms to cope with the complexity of software systems. However, the automatic generation of such systems is only possible if there is a framework that supports the process. The Model-Driven Architecture (MDA) [1] strategy advocates the use of models and platform independence in the software development process as a new way of generating applications automatically.

Furthermore the development of Expert Systems (ES) [2] applications has acquired great importance in recent years and there is a need to provide proper support for them. The Expert System captures the knowledge of experts and attempts to imitate the reasoning processes of such experts when they solve problems in a specific domain. Such systems can be designed as a generic architecture with components that are independent and separate units. However, these systems are complex because their architectural elements vary.

In order to cope with the variability problem, Software Product Lines (SPL) [3] have emerged in an effort to control and minimize the high costs of the software development process and to reduce the time to market of a family of products. This approach is based on having a generic design that is shared by all the product family members.

In this context, we have developed the Baseline Oriented Modeling (BOM) framework. BOM automatically generates Expert System (as PRISMA architectural models) based on SPL, by using the MDA approach.

In BOM the target platform is PRISMA [4]. The PRISMA architectural model integrates two approaches: Component-Based Software Development, and Aspect-Oriented Software Development. This integration is obtained by defining the architectural elements through their aspects. The PRISMA Architectural Description Language provides expressivity for the specification of software architectures with aspects at a design level. The architectural models can be automatically compiled to .NET executable applications.

Our work integrates these technological spaces: MDA, SPL, the PRISMA framework and Expert System as our case study. The Expert System design separates the inference process from the knowledge domain, and can incorporate several reasoning strategies in order to solve a problem by applying the most efficient of them. We use the MDA approach in order to automatically generate code from models by means of transformation rules and SPL to minimize the variability impact on the cost of the software production.

[1] **Acknowledgments**. This work has been funded under the Models, Environments, Transformations, and Applications: META project TIN20006-15175-605-01, and the scholarship of the Training University Faculty Program, ref. AP-2006-00690.

Cabello, M. E., Ramos, I., Gómez, A., Limón, R. Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for Expert Systems development. Proc. of the 1st Asia Conference on Intelligent Information and Database Systems (ACIIDS´09). 2009.

However not everything can be automated; user interaction is required to provide information to the system. BOM's user produces a specific Expert System when he/she inputs the variability information of a particular domain into the system by means of domain conceptual models.

The objective of this paper is to present the design and use of BOM at a high abstraction level, by means of Expert System development.

The paper is structured as follows: Section 2 presents our field study (diagnostic Expert Systems). Section 3 comments on variability management in BOM. Section 4 describes the software views, metamodels, models and transformations used in BOM. Section 5 presents the related works. Section 6 presents our conclusions and provides our ideas for future work.

2. Our field study: diagnostic Expert Systems

We have carried out a field study to learn about the variability in the Expert System and we have selected diagnosis as our case study. A number of cases have been studied: medical diagnosis, educational diagnosis, video diagnosis, and disaster victim diagnosis.

2.1 The Diagnosis: domain variability

Based on the analysis carried out in the diagnostic process we can conclude the following:

Diagnosis consists of an interpretation of the states of the entities involved (viewed as a set of properties), followed by the identification of the problems using rules.

There is variability in the diagnostic process (system behavior) and such variability can be described in terms of its features as follows: i) Entity views: an entity can be considered to participate with the same properties (the same view) or have different properties (different views) during the diagnostic process. ii) Property levels: the properties of the entities can have *n* different abstraction levels. iii) Number of hypotheses: the goal of the diagnosis is a single validated hypothesis. There can be one or several candidate diagnostic hypotheses which must be evaluated in order to select the valid one. iv) Reasoning types: show the ways in which the rules are applied by the inference motor in order to infer a final diagnosis.

There is variability in user requirements too. We have elicited and modeled as UML-Use Case diagrams [5] the user´s interaction requirements because they impact in the software architecture structure. We can describe this variability in terms of its features as follows: i) Number of use cases of the system and how the system interacts with the environment (final users). ii) Number of actors: number of final users of the system. iii) Use cases per actor: an actor can access different use cases.

However, the features of the application domain need to be considered too. Therefore other variability emerges: the application domain variability. The features that correspond to a specific application domain are: i) Name and type of the entities' properties by abstraction level. ii) Rules by abstraction levels (the rules describe how the entity properties are related inter-levels). iii) Level, name and type of the hypotheses used in the diagnostic process. These features will be used to decorate the base architecture skeletons in order to obtain the final products.

2.2 The Expert Systems: domain functionality

Based on the field study carried out in the Expert System domain, we have detected that a unique generic architecture captures the functionality of these systems. This generic architecture has three basic modules [2]: i) the module that contains the inference process that resolves a problem in a specific domain, ii) the module that contains the knowledge about the domain, and iii) the module that allows communication between the user(s) and the system.

But as we implement the SPL's assets using PRISMA software architectures, the modules of the Expert System generic architecture must be mapped into the following architectural elements: the Inference Process Component which establishes system control and provides the general resolution strategy for taking a decision; the Knowledge Base Component which contains the domain knowledge of the case study using application domain rules (Horn clauses) and facts (information unchanged); and the User Interface Component which establishes the man-machine interaction. Therefore there is a correspondence among the modules and their respective components. However, to be consistent with the PRISMA metamodel, it is necessary to incorporate a new architectural element (the connector), to establish the communication among the components; therefore the other architectural element is the Connector, which contains the choreography of the decision process.

However, we have detected that the base architectural model implementing the generic modular model is not unique; it varies in its structure and behavior. This implies the creation of several base architecture models for each generic model in a product line.

We have developed a modeling technique to design Expert System using PRISMA software architectures [6]. This modeling technique takes into account that an Expert System varies both in the structure and in the architectural elements of its architectural model. In our modeling technique the end user requirements are modeled using UML use case diagrams.

Furthermore, we have concluded that the variability in the structure of the Expert System is related to the features associated with the user requirements, and the variability in the behavior of the Expert System is related to the associated diagnostic process.

3. BOM: The framework

Following the results of our field study, variability management in BOM includes two types of orthogonal variabilities. The first variability (V1) has as variants the domain features and the end user requirements, which determine a specific base architecture. The second variability (V2) corresponds to the application domain features, which determine the final product of our SPL.

Since BOM follows a modeling oriented approach, we manage the variabilities as follows: The features of the first variability are represented in a Feature Model (see Figure 1). These features are represented as variability points and their variants.

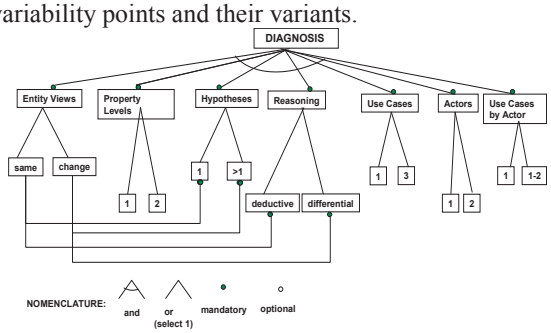


Figure 1. The Diagnostic Feature Model

The Feature Model is translated to an equivalent UML-class diagram with OCL constraints (see Figure 2): the Domain Conceptual Model. The variants are captured as instances of this Conceptual Model. The instances capturing the domain features in the medical diagnosis are: *entity view=same, property levels=2, hypotheses=14, reasoning=differential, use cases=3, actors=2, use cases by actor=2 by actor A & 1 by actor B.*

The features of the second variability, i.e. variability of the application domain, are captured as instances of the Application Domain Conceptual Model (see Figure 3). We present an example of these instances in the medical diagnosis case study: *properties of level 0=cough, fever; properties of level 1=dry_cough, constant_fever; hypotheses of level 1=warth, parotiditis; hypotheses of level 2=pneumonia, bronchitis; rules=IF (cough=true and fever=true and respiratory_dificulty=tue) THEN syndrome=warth.*

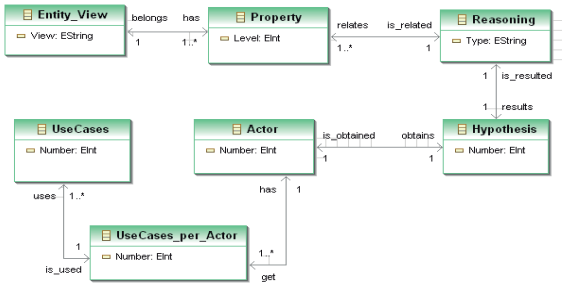


Figure 2. The domain conceptual model



Figure 3. The application domain conceptual model

For this reason, the modeling of the system is separated into two stages:

Stage 1: Constructing the Domain Conceptual Model (capturing in its instances the variants of the domain: V1) and the model of the functionality of the system (i.e. a unique generic architecture associated with the Expert System). This information, by means of Query/Views/Transformations OMG standard [7]: QVT-Relations, produces the corresponding base architectural model (as a template or skeleton of the system in the PRISMA Architectural Description Language). A unique generic architecture is shared by several base architectures (skeletons of base architectures), which represent the SPL1.

Stage 2: Constructing the Application Domain Conceptual Model capturing in its instances the application variants. The Application Domain Conceptual Model's instances together with the functional model of the system (i.e. the base architecture produced in stage 1) are transformed into a PRISMA architectural model, i.e. the final product of the SPL, i.e. the Expert System, which is compliant with the two variabilities and the functionality selected. The second type of variability involves the SPL of the application in a specific field, i.e. SPL2. This variability causes the base architectures to be decorated

with the application domain features, in order to generate the types of the PRISMA software artifacts. Therefore, a base architecture can be mapped into one or more PRISMA architectures, because in each base architecture can be inserted different properties of the application domain.

4. Expert Systems views in BOM

In BOM we have considered two kinds of views of the Expert System: the System Variability View and the System Functional View.

The System Variability View is described using the two variability conceptual models: the Domain Conceptual Model which captures the domain and user variabilities (V1) and the Application Domain Conceptual Model capturing the application domain variability (V2). The Domain Conceptual Model conforms to the V1 Metamodel (MM V1) and the Application Domain Conceptual Model conforms to the V2 Metamodel (MM V2). Both metamodels are the UML 2.0 Class Diagram Metamodel [5], but other Domain Specific Metamodels could be used giving rise to Domain Specific Models to capture the System Variability View.

The System Functional View is given, at different stages of the process, by means of two views, which are described using three architectural models.

- the modular view: for the Generic Architecture Model which conforms to the Modular Metamodel (MM MODULAR),
- the component-connector view: for the Base Architecture Models which conform to the Skeleton Metamodel (MM SKELETON), and the PRISMA Architecture Models which conform to the PRISMA Metamodel (MM PRISMA) for the final product. The Skeleton Metamodel is similar to the PRISMA Metamodel [4] but it allows feature holders.

4.1 The system´s views metamodels

We have used the Meta Object Facility (MOF) [8] graphic notation for the specification of these metamodels. The Modular and Component-Connector view types proposed in [9] have been chosen as the most appropriate views in our approach.

Figure 4 shows the Modular view metamodel. The main element considered for this view is the module. The relation class represents the styles (*decomposition, uses and layer*). These styles are treated as relations since what distinguishes one style from another is the type of relation. The labels in the links are useful for indicating how the relation is made.

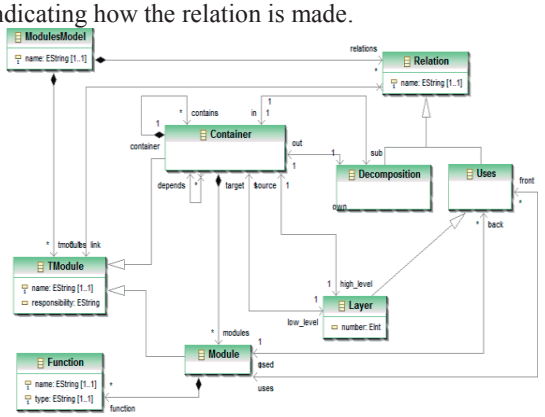


Figure 4. Modular view metamodel

Figure 5 shows the Component-Connector view metamodel. This Figure shows the component class and connector class as the main elements. Both are derived from a more general component class (TComponent). The way in which these elements are related corresponds to their relations (*Pipe-Filter, Client-Server, PeerToPeer, Publish-Suscribe, Shared-Data*). These relations are inherited from the Relation class, which links the components by means of the connector.

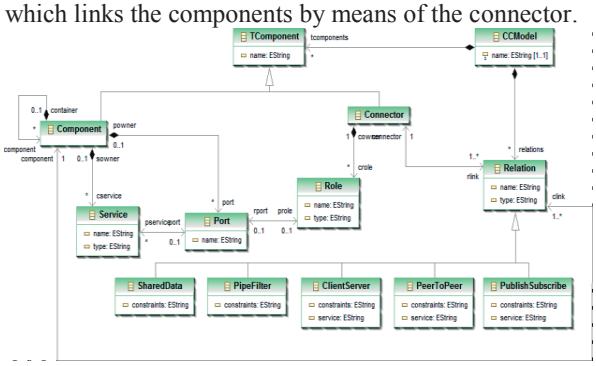


Figure 5. Component-Connector view metamodel

4.2 The relations among metamodels

We used QVT transformations to establish the relations among metamodels in MOF. The QVT-Relations language is used to describe the relations.

The source and target metamodels are identified first. The correspondence among each element of the metamodels must be defined taking into account the way in which their rules are represented through QVT. In this case, the source metamodels are the Modular metamodel and the V1 variability metamodel. The target metamodel is the Skeleton metamodel. This means that the rules considered in the relations for the elements of the modular and V1 variability metamodels, can only be of *check only* type (to verify

the elements) and of *enforce* type (to create the elements of the skeleton metamodel).

Two types of relations between the System Functional View have been identified: i) the relations between the Modular view and the Component-Connector view, ii) the relations between the Component-Connector view and the PRISMA view together with the views that define the variability of the SPL. Their respective profiles are:

R1 ⊆ MM MODULAR X MM V1 → MM SKELETON; def

R2 ⊆ MM SKELETON X MM V2 → MM PRISMA; def

4.3 The transformations among models

Figure 6 shows the transformations involved in the construction of our SPL architectures. This figure illustrates how the transformations performed at the model level are applied in the transformation processes. The modular model is made up of three modules: *InferenceMotor*, *KnowledgeBase*, and *UserInterface*. These are all linked with each other through dependence relations. Next, the transformation task for producing the skeleton model (target) is performed. To do this, the Domain Conceptual Model is used. A skeleton component is produced for each module, and each dependence relation generates a skeleton connector.

The level M1 is where the transformations T1 and T2 are carried out. A first model (skeleton model) is obtained by the transformation T1, using the modular model and the Domain Conceptual Model as sources. This skeleton model is refined by means of transformation T2 using the Application Domain Conceptual Model which allows the PRISMA model to be obtained as a refinement.

In BOM, the model transformations T1 and T2 are executed by means of our tool using QVT-Relations model transformations (these are Platform Independent Model to Platform Independent Model transformations).

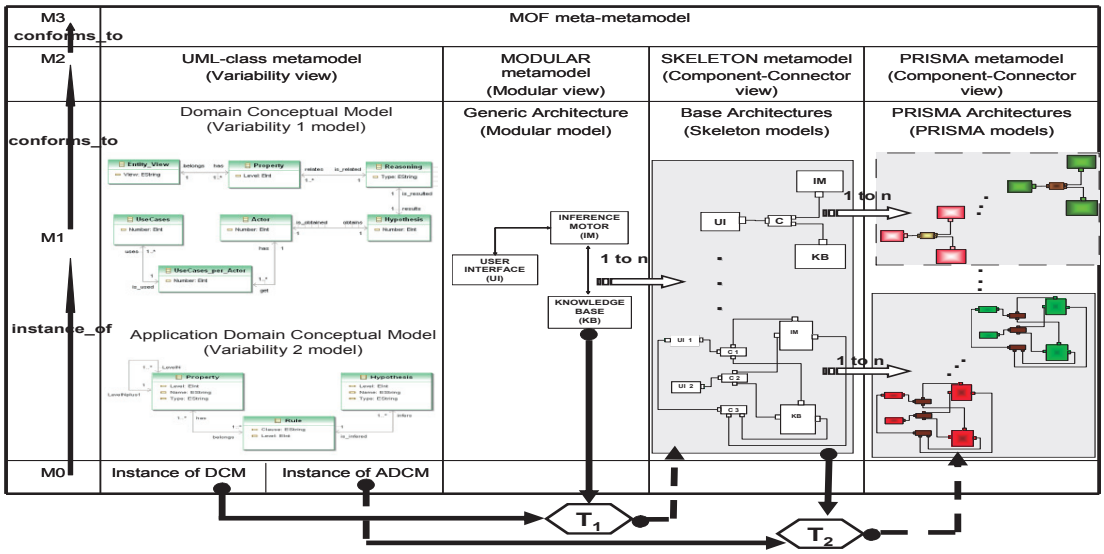


Figure 6. The model transformations T1 and T2 in BOM (expressed in the MOF levels)

The QVT-Relations code (partial) is as follows:

```
transformation modules2components(mdomain:
mview, dcmdomain: dcm, ccdomain: ccview)  {
   //… Keys declaration
  top relation ModulesModel2ComponentsModel
 {   checkonly domain mdomain modulesModel:
mview::ModulesModel { };
   checkonly domain dcmdomain varModel:
dcm::DomainConceptualModel { };
  enforce domain ccdomain componentsModel:
  ccview::CCModel {name=modulesModel.name};
 }
  where {
   UseCase2Connector(modulesModel,varModel,
componentsModel);  }     }
//… Other relations
}
```

At the end, BOM uses the PRISMA-MODEL-COMPILER tool [10] to automatically generate the executable code (in C#.NET) from the PRISMA Platform Independent Model generated model. BOM executes the object code over the PRISMA-NET Middleware [10] (our Platform Specific Model).

An example of the generated C# code which corresponds to the Knowledge Base component of the

educational program diagnosis case study is as follows:

```
namespace KBE
{ [Serializable]
  public class KBase: ComponentBase
 {public class KBase string name:base(name)
  {AddAspect (new FBaseE ( ) );
InPorts.Add("KnowPort","IDomainE","KNOW");
OutPorts.Add("KnowPort","IDomainE","KNOW");
}   } }
```

In BOM, the final user (application engineer) builds an Expert System (product of SPL) by giving as input the features of the variabilities V1 and V2 using instances of the Domain Conceptual Model and Application Domain Conceptual Model respectively.

5. Related works

SPL is nowadays a hot topic and there are a wide variety of research results offering suggestions and solutions in specific domains. Our research is related to the following: i) Batory et al. [11] capture the domain features in the Feature Model. In BOM we capture the Feature Model's features in conceptual models. ii)Trujillo [12] uses Feature Oriented Programming as a technique for inserting features. In BOM we use this technique but at the model level, i.e. we use Feature Oriented Modeling. The features are inserted, by means of QVT transformations, in the skeleton aspects of the architectural elements that make up the PRISMA models. iii) Bachman et al. [13] propose to separate the variability and the functionality description of the software artifacts. In BOM, variability and functionality are specified in separate conceptual models.

6. Conclusions and Future Work

In this paper we describe how we automatically generate Expert Systems in a specific domain, by means of our BOM framework.

In order to cope with the complexity of the Expert Systems development, we deal with the variability in two stages: in the first stage, the variability is reflected in the base architectures of the SPL sharing a generic architecture; in the second stage the variability is reflected in different application features used to produce the PRISMA architectures (i.e. our SPL) in a specific domain sharing a base architecture. BOM manages the variabilities at the model level.

BOM improves the development of Expert Systems in the following ways: i) By using the characteristics of Expert Systems to separate the inference process from the knowledge domain, and to incorporate several reasoning. ii) By applying SPL techniques. In this way cost, time, effort, and complexity are reduced. iii) By developing product line architectures as PRISMA models, reusing and integrating software components and aspects. iv) By applying MDA techniques in order to implement systems on different platforms, and to automatically obtain executable applications.

The main contributions of BOM are: application development by automatically generating code, and the management of two kinds of variability described in two independent conceptual models separated from the functionality models.

In this paper we have presented the diagnostic Expert Systems domain. In the future we will apply BOM in different domains. Additionally, we will validate BOM performances in real life situations by means of our prototype [14]. We also plan to use benchmarks in order to compare BOM results with other approaches.

References

[1] MDA: http://www.omg.org/mda.

[2] J. Giarratano, and G. Riley, *Expert Systems: Principles and Programming*. 4th Edition: (Hardcover), 2004.

[3] P. Clements, and L.M. Northrop, *Software Product Lines: Practices and Patterns,* SEI Series in Software Engineering, Addison Wesley, 2001.

[4] J. Pérez, *PRISMA: Aspect-Oriented Software Architectures*, PhD. Thesis in Computer Science, Universitat Politècnica de València, Spain, 2006.

[5] UML: http://www.omg.org/docs/formal/05-07-04.pdf.

[6] M.E. Cabello, *Análisis y diseño de un generador automático de sistemas de diagnóstico basado en LPS,* Tesis de Master, UPV, España, 2007.

[7] QVT: http://www.omg.org/docs/ptc/05-11-01.pdf

[8] MOF: http://www.omg.org/mda/specs.htm#MOF.

[9] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture, Views and Beyond*, Addison-Wesley, 2002.

[10] J. Pérez, N. Ali, C. Costa, J.A. and I. Ramos, "Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology", *3rd International Conference on .NET Technologies*, pp. 97-108. Czech Republic, 2005. ISBN 80-86943-01-1.

[11] D. Batory, D. Benavides, and A. Ruiz-Cortés, "Automated Analyses of Feature Models: Challenges Ahead", *ACM on Software Product Lines*, 2006.

[12] S. Trujillo: *Feature-Oriented Model Driven Product Lines*, PhD. Thesis, The University of the Basque Country, San Sebastian, Spain, 2007.

[13] F. Bachman, et al. "A metamodel for representing variability in product family development", *the 5th International Workshop on Product Family Engineering*, 2003, pp. 66-80.

[14] M.E. Cabello, M. Gómez, M. LLavador, and I. Ramos. *ProtoBOM: a framework that semi-automatically generates Decision Support Systems based on SPL*, Technical report: DSIC II/02/08, UPV, Spain, 2008.

Expert Systems development through Software Product Lines techniques

María Eugenia Cabello[1] **& Isidro Ramos**[2]

Polytechnic University of Valencia. Camino de Vera s/n, 46022 Valencia, Spain

[1]mcabello@dsic.upv.es, [2]iramos@dsic.upv.es

Abstract: This paper deals with Expert Systems (ES) development as the final product of a Software Product Line (SPL). We take into account software variability management in the ES domain. Two kinds of variability emerge: variability in the behavior as well as in the structure of such systems. Experts' knowledge is captured using domain conceptual models in order to manage the variability and functionality of the ES. The ES are constructed using our Baseline Oriented Modeling (BOM) approach. BOM is a framework that automatically generates software applications as PRISMA architectural models by using SPL techniques. We follow the Model-Driven Architecture (MDA) initiative for building domain models, which are automatically transformed into executable applications.

Keywords: software product lines, variability, expert systems, software architecture, model transformations.

1. Introduction

The development of Expert Systems (ES) [1] applications has acquired great importance in recent years and there is therefore a need to properly support them. But these systems are complex because their elements varient. In order to minimize this problem, we pro-

Cabello, M. E., Ramos, I. Expert Systems development through Software Product Lines techniques. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008.

pose to construct the ES using a Model-Driven Architecture (MDA) [2] approach based in Software Product Lines (SPL) [3] techniques. We use SPL techniques in order to control and minimize the time to market and high costs of the software development process. We use MDA approach to generated automatically code.

The representation and management of variability is in this case is a key factor. We describe how variability is managed in our SPL by means of our Baseline Oriented Modeling (BOM) approach. In this context, BOM automatically generates ES in a specific domain by using SPL techniques. The products of our SPL are implemented by PRISMA models [4] capturing the architecture and functionality of the rule-based ES.

To develop BOM, a field study has been done to learn about the structure and functionality of the rule-based ES that carry out tasks in a specific domain. In order to treat in depth a specific field of the ES application, we have selected the diagnosis domain. Some case studies (medical diagnosis, educational diagnosis, video diagnosis, disastre victims diagnosis) have been considered.

The structure of the paper is as follows: Section 2 presents the different technological spaces that our work integrates, and mentions some related works. Section 3 presents our field study in the ES domain. Section 4 describes how BOM manages variability. Section 5 describes ES development by BOM. Section 6 presents our conclusions and proposals for future work.

2. Foundations

Our work integrates the following technological spaces (TS) in order to cope with the complexity of the problem.

a) The Expert System (ES) TS captures the knowledge of experts and attempts to imitate their reasoning processes when the experts solve problems in a specific domain. Such systems usually have a generic architecture and its components are independent and separate units. Control is independent from the data. The input and output of the information is carried out through the user interface.

b) Model-Driven Architecture (MDA) TS is an initiative promoted by the Object Management Group (OMG) for software system development. MDA is based on the separation of the description of the functionality of the system from its implementation on specific software platforms. MDA proposes defining and using models (as first class citizens) at different abstraction levels. From these models we can automatically generate code by means of transformation rules to obtain executable models.

c) PRISMA framework TS integrates two approaches: Component-Based Software Development and Aspect-Oriented Software Development. This integration is obtained by defining the architectural elements through their aspects. The PRISMA model consists of three types of architectural elements: components, connectors, and systems.

d) Software Product Lines (SPL) TS combines systematic development and reuse of components or assets; i.e. the products are different in some features but they share a generic architecture. In the SPL approach, rather than a single application, the development process produces a family of them. More than creating products, it is a question of storing software assets in a Baseline repository, and defining a production plan that specifies the process to obtain each one of the individual products.

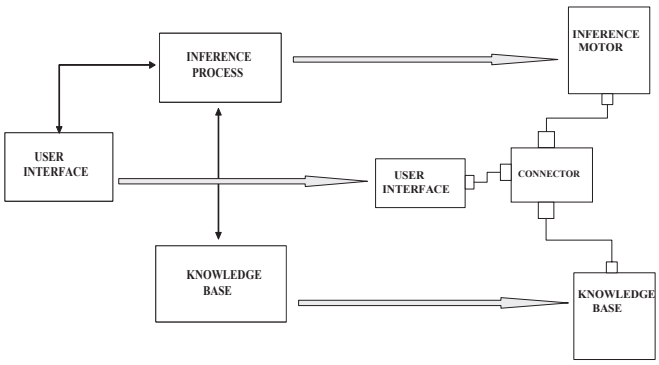
Our research is related to the following SPL works: i) Batory et al. [5] express the domain features in a Feature Model (FM). In BOM we capture domain features using FM too. However, we observed that solving the complexity of ES needs two FM, capturing the domain and application domain features respectively. ii) Trujillo [6] uses Feature Oriented Programming as a technique for inserting features into XML documents by means of XSLT templates. In BOM we uses this technique but at the model level (i.e. we uses Feature Oriented Modeling), and the features are inserted in the PRISMA architectural models, by means of QVT- transformations. iii) Bachman [7] proposes to separate the artifact's variability declaration of its funcionality. In BOM, the specification of the two variabilities and functionality of the ES are captured in separate conceptual models.

3. The architecture of the Expert Systems

The most important asset (core asset) of our SPL is the generic architecture of the ES product line (also called domain architecture). This is due to the fact that this architecture determines the scope of the SPL and the features of the products that can be developed.

The process of developing an application (member of the product line) begins with a unique generic architecture model that is specific to the domain. The generic architecture model of our SPL is represented in Figure 1 by three fundamental modules: i) Inference Process Module, transforming data into useful information for decision making and implementing the inference process. ii) Knowledge Base Module, containing the knowledge of the application domain. iii) User Interface Module, allowing user interaction with the system.

But as we implement the SPL using PRISMA software architectures models, the generic modules of an ES must be mapped into (skeleton) architectural elements. To be consistent with the PRISMA metamodel, it is necessary to incorporate a new architectural element (the connector), to establish the communication among the components and to define the choreography of the decision process. The choice of this implementation results in a mapping between the two models with a small semantic GAP. As is shown in Figure 1 (using gray arrows), there is a correspondence among the modules and their respective components.



a) The generic architecture b) A base architecture

Fig. 1. Mapping between the module generic architecture and a base architecture

3.1 The variability in the architecture of the Expert Systems

We have detected that the base architecture model implementing the generic architecture model is not unique but varies in its behavior, and since we use PRISMA models, in its structure too. In sections 3.1.1 and 3.1.2 of this paper we present the analysis of this variability.

This implies the creation of different base architecture models, every time we start to develop a product line. To obtain the base architecture model from the domain generic architecture model may require to remove or to add components or relationships between them, reconfigure components and develop specific software elements. This implies the existence of additional features of the base architecture model that are represented in the variability points and whose presence/absence defines the specific product.

3.1.1 Variability in the Expert Systems structure

In this section we present the modeling technique that we have developed to configure ES using PRISMA models. This modeling technique shows that an ES as a PRISMA architecture model varies both in the configuration and in the the component architectural elements.

Our modeling technique starts with the end user requirements modeled as UML use case diagrams. The number of use cases, the number of actors, and the number of use cases that an actor accesses are the relevant variants.

In the following, we present the step-by-step modeling of an ES using as case study the medical diagnosis.

1) *Specify the functional requirements (at high abstraction level) by using UML use case diagrams.* In our example (see Figure 2a) we have three use cases and two actors, where an actor accesses two use cases and the other one access a single use case.

2) *Create an architectural model for each use case*. In our example (see Figure 2b) we have three use cases, therefore we build three architectural models.

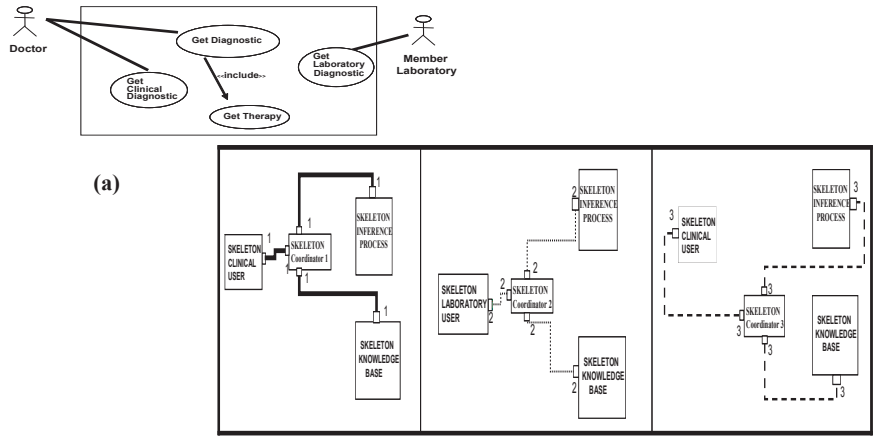


Fig. 2. Example of a use cases diagram (a) and its architectural models respectively (b)

3) *Build the final architectural model*.- the architectural elements of the final model are obtained using the following criteria:

• connectors: the number of connectors of the final architectural model is equal to the number of use cases.

• components: we have one component per module and to join each component's port we use attachments. The number of ports of a component will be equal to the number of (use cases) connectors where the component is used.

The architectural elements of the final model therefore have the following characteristics: there is one connector connecting all the architectural components for each use case; the number of ports of the Inference Process component is the same as the number of use cases, the number of ports of the Knowledge Base is the number of use cases, the number of User Interfaces is the number of actors of the use cases, the number of ports of the User Interface is the number of use cases that can be accessed by an actor.

In our example, the final architectural model of the ES is presented in Figure 3. It has seven architectural elements with the following characteristics: the Inference Process component has three ports, the Knowledge Base component has three ports, the User Interface 1 component (i.e. doctor) has two ports, the User Interface 2 component (i.e. laboratory member) has one port, the Coordinator 1 connector, the Coordinator 2 connector, the Coordinator 3 connector.

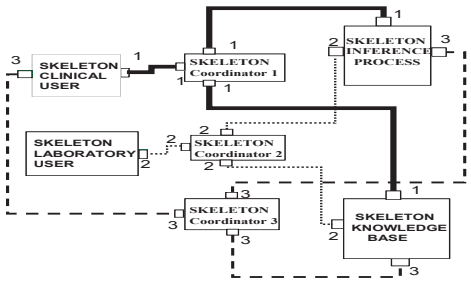


Fig. 3. Medical diagnosis base architectural model

3.1.2 Variability in the Expert Systems behavior

The behavior of the architectural elements of an ES varies according to the reasoning strategy used and is part of the Inference Process component. It contains the process of obtaining a decision or solving a problem. The reasoning strategies simulate the reasoning of experts when performing a task (e.g. diagnosis).

We have represented reasoning as processes. In the .field analysis of the diagnosis domain, we have detected two types of decision processes: static and dynamic.

In Figure 4a we present a visual metaphor of the static inference process through a graph showing the set of properties (P_i) at level i (grey color), hypothesis (H_i) (black color), as well as deductive reasoning (arrows pointing up) in this process. The inference rules are of the form:

$$p_{j,i} : P_i \leftarrow p_{k,\, i-1}{}^* : P_{i-1} \quad \text{and} \quad h_{j,i} : H_i \leftarrow p_{k,i-1}{}^* : P_i$$

where i = 0...n and n=number of levels and j,k: nat. For example, in Figure 4a we have (using a natural notation extension for the sake of simplicity):

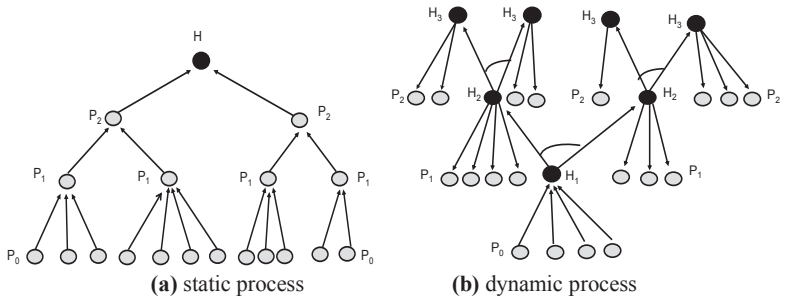
P0, P1 ← P0, P2 ← P1, H ← P2.

In Figure 4b we present a visual metaphor of the dynamic inference process through a graph, showing levels of properties (grey color), several hypotheses (black color), as well as deductive reasoning (arrows pointing up) and inductive reasoning (arrows pointing down) in this process. The inference rules are:

$$h_{j,i+1} : H_{i+1} \leftarrow p_{k,i}^{*} : P_i \ \wedge \ h_{l,i}^{*} : H_i$$

where i = 0,1,..,n and n = number of levels and j,k,l: nat. For example, in Figure 4b we have:

H1 ← P0, H2 ← P1 ^ H1, H3 ← P2 ^ H2 (extended natural notation).



(a) static process **(b)** dynamic process

Fig. 4. Inference process diagrams

The behavior of the architectural elements varies according to the reasoning strategy applied. The strategy is closely related to the view of the entity during the diagnosis process, the property levels of this entity, and the number and level of hypotheses involved in this process.

In the static process an entity has the same properties (the same view) as during the diagnostic process, and only one hypothesis (goal). In the dynamic process an entity has different properties (different views) than during the diagnostic process, and it has several

candidate diagnostic hypotheses which must all be evaluated in order to select the valid one.

Figures 5a and 5b show the static and dynamic inference processes respectively, wich are modeled using the Business Process Management Notation (BPMN).

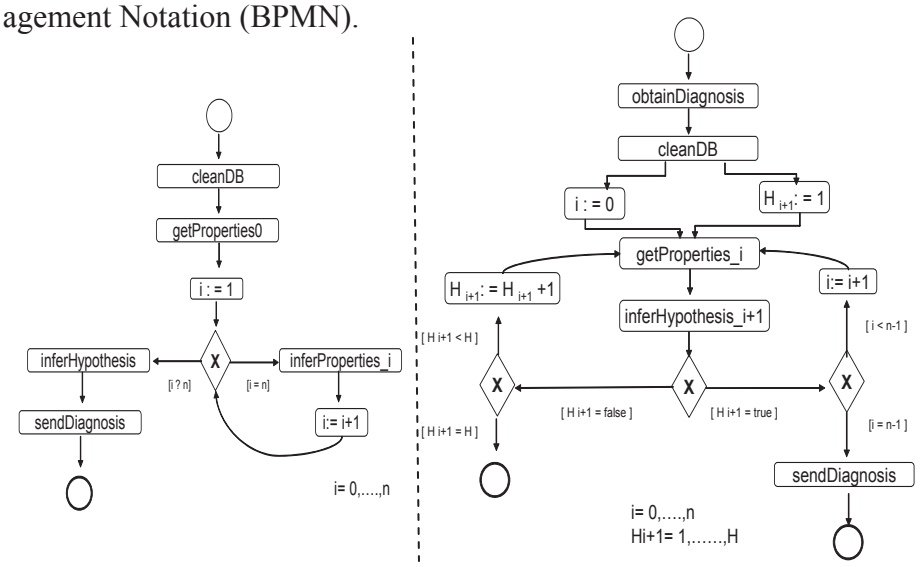


Fig. 5. Static (a) and Dynamic (b) inference processes in BPMN

4. Variability management in BOM

Since section 3.1 of this paper, we concluded that:

- the variability in structure among the architectural elements of the ES includes: number of use cases, number of actors, and uses cases by actor, i.e. this is the end user requirements variability.
- the variability in the behavior of the ES is related with: reasoning type, entity views, number of levels of the entity properties to diagnose, and number and level of hypotheses, i.e. this is the domain variability.

The union of the previous variabilities constitutes a first variability: V1. However, we also considered the scope of the application do-

main to be necessary. Therefore a second variability emerges: the application domain variability: V2.

These variabilities can be described in terms of features. For this reason and based on the field study carried out, we conclude that the features that define the ES involve two types of orthogonal variability: The first variability (V1) is given by the domain features and the end user requirements, which determine a specific base architecture. The second variability (V2) corresponds to application domain features, which determine the final product of our SPL.

5. Expert Systems development in BOM

The development in BOM of ES is based on MDA and SPL techniques, the PRISMA model, and the structure and operation of the rule-based ES. This process involves building a Baseline (a repository containing all the assets needed to build a product of the SPL) and to model the process of executing the production plan of the SPL.

In Figure 6 we show how an ES is generated by BOM in two steps. In the first step, a skeleton base architecture model is obtained from the modular model (generic architecture of the ES) and a V1 instance (captured by the Domain Conceptual Model: DCM) by means of a first QVT-transformation T1. In the second step, a PRISMA architectural model is obtained from the skeleton base architecture model (which was generated in the previous step) and an V2 instance (captured by the Application Domain Conceptual Model: ADCM) by means of a second QVT-transformation T2.

In BOM, the model transformations T1 and T2 are executed by means of the MOMENT tool [8] using OMG QVT-Relational [9] model transformations. At the end BOM uses the PRISMA-MODEL-COMPILER tool [10] to automatically generate the executable code (in C#.NET) of the generated model. BOM executes the object code over the PRISMA-NET Middleware [11].

In BOM, the user (application engineer) builds an ES (product of SPL), by giving as input the features of the variabilities V1 and V2 using instances of the two conceptual models: DCM and ADCM.

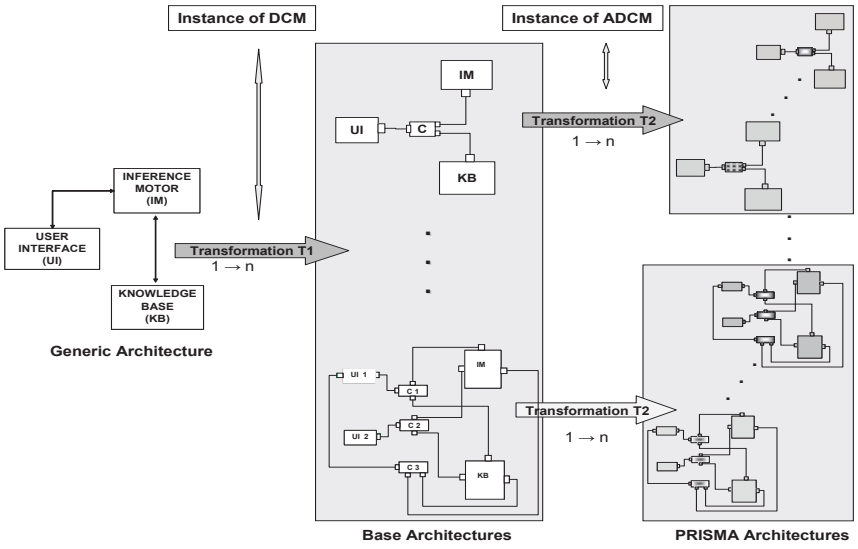


Fig. 6. Visual metaphor of ES development in BOM

6. Conclusions and Future Work

In this paper we describe the variabilities of the ES, and how are managed by means of our BOM framework for automatically generating applications in this specific domain.
In order to cope with the complexity of the ES development, we deal with variability in two stages: in the first stage, variability is reflected in the base architectures of the SPL sharing a generic architecture, in the second stage variability is reflected in the application features producing the SPL in a specific domain sharing a base architecture. BOM manages the variabilities at the model level (not at the program level). The models are automatically compiled and the generated code is an executable application.

BOM can improve the development of ES in the following ways: By using the characteristics of ES to separate the inference process of the knowledge domain, and to incorporate several reasoning strategies in order to solve a problem by applying the most efficient one. By applying SPL techniques to the construction of a design shared by all the members of a program family; in this way cost, time, ef-

fort, and complexity are reduced. By developing product line architectures as PRISMA models, and reusing and integrating software components and aspects, in order to facilitate the management of complexity. By applying MDA techniques in order to implement systems on different platforms, and to automatically obtain an executable ES.

The main contributions of BOM are: the ES development by means of automatically generating code, the management of two variability kinds described in two independent conceptual models separated from the functionality models. In the future we will apply BOM in different domains and we will validate its performances in real life cases by means of our prototype ProtoBOM [12] which we have recently completed. We also plan to use benchmarks in order to compare BOM results with other approaches.

Acknowledgments. This work has been funded under the Models, Environments, Transformations, and Applications: META project TIN20006-15175-605-01.

References

1. Giarratano, J., and Riley, G., *Expert Systems: Principles and Programming*. 4th Edition: (Hardcover), ISBN: 0534384471, 2004.
2. Model Driven Architecture: MDA. http://www.omg.org/mda
3. Clements P., and Northrop L.M., *Software Product Lines: Practices and Patterns,* SEI Series in Software Engineering, Addison Wesley, 2002.
4. Pérez J., *"PRISMA: Aspect-Oriented Software Architectures"*, *PhD. Thesis of Philosophy in Computer Science,* Polytechnic University of Valencia, Spain, 2006.
5. Batory D., Benavides D., and Ruiz-Cortés A., Automated Analyses of Feature Models: Challenges Ahead, ACM on Software Product Lines, 2006.
6. Trujillo S., "Feature-Oriented Model Driven Product Lines", *PhD. Thesis*, The University of the Basque Country, San Sebastian, Spain, 2007.
7. Bachman F., Goedicke M., Leite J., Nord R., Pohl K., Ramesh B., and Vilbig A, "A meta-model for representing variability in product family development", In Proceedings in the *5th International Workshop on Product Family Engineering*, pp. 66-80, 2003.
8. MOMENT Project, http://moment.dsic.upv.es/
9. QVT Query/View/Transformations OMG Document ad/2005-07-1.
10. Cabedo R., Pérez J., Carsí J.A. y Ramos I., "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", *IV Workshop DYNAMICA*, Murcia, España, 2005. (in spanish).
11. Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I., "PRISMANET: Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", *X Jornadas de Ingeniería de Software y Base de datos,* Granada, España, pp. 27-34, 2005, (in spanish).
12. Cabello M.E., Gómez M., LLavador M., and Ramos I. ProtoBOM: a framework that semi-automatically generates Decision Support Systems based on Software Product Lines. *Technical report*: DSIC II/02/08, Polythecnical University of Valencia, pages 68, 2008.

Model-Driven Development of Decision Support Systems: tackling the variability problem

María Eugenia Cabello[1] & Isidro Ramos[2]

Polytechnic University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain
[1]mcabello@dsic.upv.es, [2]iramos@dsic.upv.es

Abstract: In this paper we present software variability management using conceptual models for Diagnostic Decision Support Information Systems (DSS) development. We use a Software Product Line (SPL) approach. In the construction of the SPL, two orthogonal variabilities are used to capture domain (i.e. diagnosis) and application domain (i.e. medical diagnosis) particularities. In this context we describe how variability is managed by using our BOM (Baseline Oriented Modeling) approach. BOM is a framework that automatically generates applications as PRISMA software architectural models using model transformations and SPL techniques. We use Model-Driven Architecture (MDA) to build domain models (i.e. Computational Independent Models: CIMs), which are automatically transformed into Platform Independent Models: PIMs, and then compiled to a executable application (i.e. Platform Specific Model: PSM). In order to illustrate BOM, we focus on a type of Information System, the Decision Support System, specifically in the diagnostic domain.

Keywords: software product lines, conceptual models, variability management, decision support systems.

1. Introduction

The development of families of applications has become increasingly important and with it the need for automatic support. In this context Software Engineering must provide tools and methods in order to develop products with varied capabilities and adaptable to different situations. As a response to this, the concept of Software Product Lines (SPL) [1] has been developed with the aim of controlling and

Cabello, M. E., Ramos, I. Model-Driven Development of Decision Support Systems: tackling the variability problem. Proc. of the 17th International Conference of Information Systems Development (ISD´08). 2008.

minimizing the high costs of the software development process, in order to develop families of products, by means of representing and managing variability.

In this paper, we describe how variability is managed in our SPL by means of our Baseline Oriented Modeling (BOM) approach. In order to present BOM we have chosen the Decision Support Systems (DSS) [2] domain as a case study in information systems development, and we selected the diagnosis as the specific domain.

However, DSS are complex and the elements that their software architecture contains can vary significantly. We have found that in DSS, variability management cannot be dealt with by using a single variability model, but rather by using two models in two phases: the first phase using a Domain Conceptual Model (DCM) for capturing the variants that characterize the various base architectures that are derived from a unique domain generic architecture; and the second phase by means of a more classic treatment using an Application Domain Conceptual Model (ADCM) and decorating the previous base architectures with different application features, in order to obtain a specific product using Product Lines and Model Transformation techniques. In this context, BOM automatically generates DSS in a specific domain using SPL. The products of our SPL are implemented using PRISMA software architectural models [3] having the functionality of the DSS.

Our work integrates the following technological spaces to cope with the complexity of the problem: the generic architecture of the DSS to capture the knowledge of experts; Model-Driven Architecture (MDA) [4] at the abstract modeling level (Platform Independent Model: PIM); the PRISMA Architectural Framework as the target software level (Platform Specific Model: PSM), where aspect-oriented software architectures are used at a design level; Software Product Lines (SPL) as a technique for systematic reuse of products; and Feature-Oriented Programming (FOP) [5] to capture the variability of the application domain.

BOM incorporates techniques and methodologies related to a number of previous works. Batory et al. [6] express the domain features in the Feature Model. In BOM we capture features of the Feature Model in a Decision Tree and then in DCM in order to select the

base architectures. Trujillo [5] uses FOP as a technique for inserting the features into XML documents by means of XSLT templates. In BOM we use this technique in the PRISMA architectures (as XML documents). Bachman [7] proposes separation of the variability declaration of the affected artifacts. In BOM, the specification of variability and functionality constitutes separate conceptual models.

The structure of the paper is as follows: Section 2 presents the variability in the diagnostic domain; Section 3 describes how BOM manages variability; Section 4 presents the model transformations in BOM; and Section 5 presents our conclusions and future work.

2. Variability

In the DSS we have detected a unique generic architecture that captures the functionality of these systems. The generic architecture of DSS has three basic elements [2]: the module that contains the inference process that resolves a problem in a specific domain, the module that contains the knowledge about the domain, and the module that allows communication between user and system.

By capturing the user interaction requirements by means of use case diagrams we have detected the features associated with the DSS structure. This variability is reflected in the construction of the architectural elements and in the final architecture, which is shown in [8]. These features are as follows: the number of use cases: a use case indicates the division of the system based on its functionality, i.e. the various operations of the systems and how the system interacts with the environment (final users); the number of actors: represents the number of final users of the system; and use cases per actor: a user can access different use cases.

A field study has been realized in a type of DSS (the Expert Systems) in [9]. This paper explains how in BOM the three basic modules are mapped into the following PRISMA architectural elements:

- The Inference Motor Component- this establishes the system's control and provides the general resolution strategy for decision-making. It is independent of system knowledge.

- The Knowledge Base Component- this contains the domain knowledge of the case study using rules of inference (Horn clauses) and facts (information that remains unchanged).
- The User Interface Component- this establishes the man-machine interaction by allowing communication between users and system.
- The Coordinator Connector- this contains the choreography of the decision-making process.

We have chosen the diagnostic domain as our domain study. To understand this domain, a field study has been carried out considering five case studies. After finishing the field analysis we can conclude that a diagnosis consists of an interpretation of the entity states involved (viewed as a set of properties), followed by the identification of the problem by means of these properties.

The features associated with the diagnostic process are as follows:

- Entity views: an entity can be considered to have the same properties (the same view), or to have different properties (different views) during the diagnostic process.
- Property levels: the properties of the entities can have *n* different abstraction levels. The rules that are used to relate the properties of the entities have *n-1* levels, where n is the level of the entity properties.
- Number of hypotheses: the goal of the diagnosis is a single validated hypothesis. There can be one or several candidate diagnostic hypotheses which must be evaluated in order to select the appropriate one.
- Reasoning types: reasoning shows the way in which the rules are applied by the inference motor in order to infer a final diagnosis. The reasoning types can be: deductive reasoning (driven by data), inductive reasoning (driven by goals), and differential reasoning (establishing the difference between two or more diagnostic possibilities).

The features that correspond to the application domain are: the name and type of entity properties by abstraction level, the rules by abstraction levels (where the entity properties are related through rules), and the level, name and type of the hypotheses used in the diagnostic process.

3. Variability management in BOM

Based on the field analysis carried out, we conclude that the variants in DSS are included in two orthogonal types of variability:

- The first variability is given by the specific domain, for example the diagnosis, (i.e. the features of the domain associated with the behavior of the architectural elements) and the end user requirements (i.e. features related to the structure of the architectural elements and the system itself).
- The second variability corresponds to the scope of the application domain.

The features of the first variability (which are described in [8]) are represented in the Feature Model for capture of domain variability (see Figure 1a). These features in the Feature Model are represented as variability points and their variants in a Decision Tree (see Figure 1b). Both the Feature Model and the Decision Tree are Computational Independent Models (CIMs).

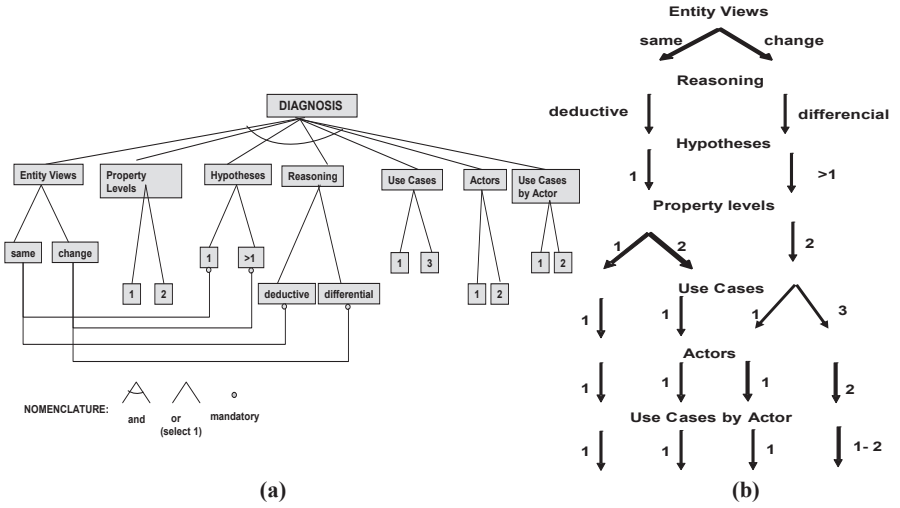


Fig. 1. The Diagnostic Feature Model (a) and the Diagnostic Decision Tree (b).

The variability shown in the Feature Model and the Decision Tree is captured in the Domain Conceptual Model (DCM) instances (see Figure 2). The variability of the application domain is represented in the Application Domain Conceptual Model (ADCM) instances (see

Figure 2). Both DCM and ADCM are Platform Independent Models (PIMs).

We specify the variability of the systems independently from their functionality using specific conceptual models defining Domain Specific Languages (DSL) suited to the domain.

The approach used in BOM models variability by using a variability model and employing an independent model for the functionality of the system (the initial generic architecture). However, in the DSS domain the functional model of the system (i.e. the basic architecture) is not unique.

For this reason it is necessary to divide the variability treatment into two stages. In the first stage two models of the system are produced: the variability model (capturing domain variability) and the functional model of the system (i.e. a unique generic architecture of the DSS). A first process produces the skeleton functional model. In the second stage two models are used: the second variability model (variability of the application domain) and the previous skeleton functional model of the system (i.e. the selected skeleton base architecture of the repository of assets or Baseline). A second process is used in order to obtain a PRISMA architecture model, i.e. a final product of the SPL.

Therefore, our approach corresponds to the development of two SPL: the base architecture SPL that shares a generic architecture, i.e. SPL1, and the application SPL in a specific domain that shares a base architecture skeleton, i.e. SPL2.

These SPLs corresponds to the two variabilities that are managed in our approach in two stages: the first variability (V1) that includes the domain variability and the user requirements, and the second variability (V2) that includes the application variability.

These variabilities are captured in instances of the two conceptual models, i.e. DCM and ADCMs.

The initial variability is dealt with through variability points embodied in a Decision Tree, which involves selecting a specific base architecture, through an instance of the DCM. The second variability is managed decorating these base architecture skeletons with the features of the application domain, using an instance of the ADCM.

This will produce the target PRISMA architecture of the final product (see Figure 2).

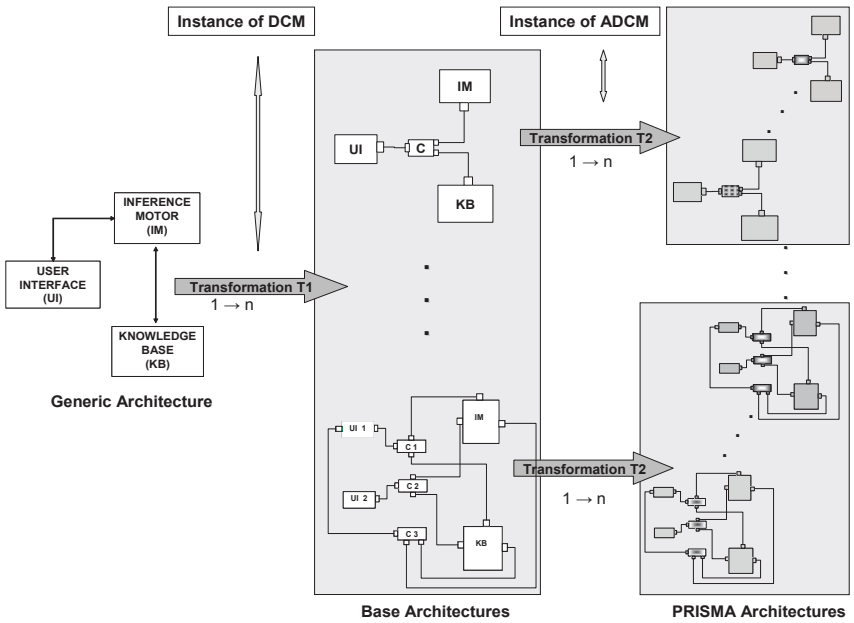


Fig. 2. Model transformations in BOM.

3.1 The first variability in BOM

The initial variability (related to the features of the domain) is represented and managed through two models:

- The first model corresponds to the functional model of the domain, represented by the generic architecture of the DSS (as a PIM) of our SPL (see Figure 2). This architecture is shared by several base architectures skeletons (as PIMs), which represent the SPL1 (see Figure 2).
- The second model is the model of the domain variability, which is represented by the DCM (as a PIM) (see Figure 2). The instances of DCM are used as the variants in the decision tree to select the skeleton base architecture that corresponds to the choice made.

The first type of variability involves a family of base architectures, i.e. SPL1. This SPL must be constructed and stored in the Baseline in the Domain Engineering phase. The Baseline then contains a SPL, formed by all the assets and the know-how necessary to construct the base architectures. Some assets are templates (or skeletons) and others are the configuration information for the base architectures. Thus, when we configure the base architectures, we obtain the first SPL.

3.2 The second variability in BOM

The second variability is also represented and managed using two models:

- The first model corresponds to the functional conceptual model of the application domain, which is captured by the skeleton base architecture (as a PIM) of the SPL1 (see Figure 2).
- The second model corresponds to the conceptual model of the variability of the application domain, which is represented by the ADCM (as a PIM) (see Figure 2).

The features used to fill the skeletons in order to obtain the PRISMA types are the features of the application domain (see Figure 2). In this way, the second type of variability involves the SPL of the application in a specific field, i.e. SPL2.

In the application variability management process, the variants of the specific requirements of the application domain should be selected. This selection is made by means of an ADCM instance given by the user. The features are inserted in the base skeletons in order to generate the types of the PRISMA software artifacts. This is shown in Table 1.

The PRISMA architectural elements will be used to configure the PRISMA architectural model of the application. This model (as a PIM) is automatically compiled into code (C#) using the PRISMA-MODEL COMPILER [10] and the PRISMA-NET Middleware [11]. The generated code contains all aspects of an executable application.

Table 1. PRISMA-ADL of the Functional Aspect of the Knowledge Base (the different section holes are depicted in bold type).

Functional aspect of the Knowledge Base of a skeleton	Functional aspect of the Knowledge Base of a PRISMA type
Functional Aspect FBaseEP using IDomainEP	Functional Aspect FBaseEP using IDomainEP
Attributes	Attributes
Variables	Variables
<FP.0>	**laboratories:string, library:string, ...**
Derived	Derived
<FP.1>	**infrastructure&services:string,faculty:string,....**
<FH>	**developmental_stage:string;**
Derivations	Derivations
<FR.1>	**{laboratories=”good” and library=“good”} infrastructure&services:=“good”;**
	.....
<FR.2>	**{infrastructure&services=“good” and faculty= “good”} developmental_stage:=“consolidated”;**
.....	.....
Services	Services
.....	.....
End_Functional Aspect FBaseEP	End_Functional Aspect FBaseEP

4. Model Transformations in BOM

Figure 2 shows the transformations involved in the construction of our SPL architectures. This figure illustrates how the transformation performed at the model level is applied in the transformation processes. The transformations T1 and T2 are executed at the model level (level M1 in OMG-MOF [12]), and they are defined at the metamodel level (level M2 in MOF). In transformation T1, the Skeleton model (level M1 in MOF) is obtained from the DSS modular

model (level M1 in MOF), and the instances of DCM (level M0 in MOF). In transformation T2, the PRISMA model (level M1 in MOF) is obtained from the Skeleton model (level M1 in MOF) generated in T1, and the instance of ADCM (level M0 in MOF). The PRISMA model is the final result (level M1 in MOF).

The generic architecture conforms to the module metamodel, the skeleton base architectures conform to the skeleton metamodel (i.e. the PRISMA metamodel with holes), the PRISMA model conforms to the PRISMA metamodel, and the DCM and ADCM variability models conform to the UML class metamodel.

The transformations can be carried out in two different ways:

a) With the orthodox method, by means of QVT-Relations [13]. In transformation T1, QVT-Relations must take into account the generic architecture DCM, and the skeleton base architecture configuration. In transformation T2, QVT-Relations must take into account the base architectures and ADCM giving as result very complex transformation rules. This implies quite a complex and non-scalable transformation as an algorithmic solution calculating variability.

b) With the BOM approach. In transformation T1, instead of calculating the transformation function using QVT relations each time, the function is computed once and the result is stored in the Baseline for each input. In this way, given all the variants, T1 is implemented as a direct access to the table that represents the Baseline. In this way, with decision tree techniques we can select the skeleton base architecture of the specific case, given the variants of the variability points (features of the domain) as instances of the DCM. In transformation T2, using FOM techniques, an XSLT decoration process of the skeleton base architectures is implemented employing the application domain features given as instances of the ADCM. The transformation profiles of T1 and T2 are:

T1 (GenericArch_model, Instance-V1_model) = BaseArch_model

T2 (BaseArch_model, Instance-V2_model) = PRISMA Arch_model

This solution has the advantage of dealing separately with the functional and the variability models, allowing conversion of a generic

architecture into several base architectures, and transforming a skeleton base architecture into a PRISMA architecture. In this approach the transformations is easy, modeling of variability is independent of functionality, and the problem is resolved in a more effective, scalable, friendly way and in addition is expressively richer and clearer.

5. Conclusions and Future Work

We have described how variability is managed in our SPL by means of our BOM framework. BOM automatically generates applications in a specific domain using SPL. However, in order to present our BOM approach, we have selected the DSS as the system domain and the diagnosis as the specific domain.

BOM manages variability at the model level (not at the program level). This approach manages variability in two stages, which correspond to the development of two SPL: the base architecture SPL1 that shares a generic architecture, and the application SPL2 in a specific domain, which shares a base architecture skeleton. These SPLs correspond to domain variability and application variability, which are captured by means of BOM by using two compiled conceptual models applications: DCM and ADCM respectively.

In this context, BOM captures data that characterize the domain variability and the application variability using conceptual models. Using Decision Tree and FOP, this data can be used in the domain engineering and product engineering phases respectively in order to obtain a specific application by means of SPL and Model Transformation techniques. In BOM, the stakeholders (application engineers) introduce the variability data during the execution of the SPL Production Plan by means of the Conceptual Model instances.

In BOM the two variability kinds are described in two independent conceptual models separated from the functionality models; and the information systems applications are automatically generated in a user friendly way. In doing so, BOM minimizes the high costs of the software development process and reduces the time to market of every new product.

In the future, BOM will be applied in different information system domains (e.g. payroll, accounting, etc.) and application domains (e.g institutions, enterprises, etc.). A prototype of the BOM framework (ProtoBOM) [14] has been implemented and will be validated in real case studies. Additionally, we will use benchmarks in order to compare BOM results with other approaches.

Acknowledgments. This work has been funded under the Models, Environments, Transformations, and Applications: META project TIN20006-15175-605-01.

References

1. Clements P., and Northrop L.M., *Software Product Lines: Practices and Patterns,* SEI Series in Software Engineering, Addison Wesley, 2002.
2. Turban, E., and Aronson, J.E., *Decision Support Systems and Intelligent Systems,* Prentice Hall, 2001, 865 pages, ISBN: 0-13-089465-6.
3. Pérez J., PRISMA: "Aspect-Oriented Software Architectures", *PhD. Thesis of Philosophy in Computer Science,* Polytechnic University of Valencia, Spain, 2006
4. Model Driven Architecture. http://www.omg.org/mda
5. Trujillo S., "Feature-Oriented Model Driven Product Lines", *PhD. Thesis*, The University of the Basque Country, San Sebastian, Spain, 2007.
6. Batory D., Benavides D., and Ruiz-Cortés A., "Automated Analyses of Feature Models: Challenges Ahead", ACM on Software Product Lines, 2006.
7. Bachman F., Goedicke M., Leite J., Nord R., Pohl K., Ramesh B., and Vilbig A, "A metamodel for representing variability in product family development", In Proceedings in the *5th International Workshop on Product Family Engineering*, pp. 66-80, 2003.
8. Cabello M.E. and Ramos I. "A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines". In Proceedings of HEALTHINF´08*, International Conference of Health Informatics-INSTICC*, Madeira, Portugal, Vol. 2, pp. 237-246, 2008, ISBN 978-989-8111-16-6.
9. Cabello M.E. and Ramos I. "Expert Systems development through Software Product Lines tecgniques". In Proceedings of ISD´08*, 17 th International Conference on Information Systems Development, Paphos, Cyprus, ago.* 2008, (paper accepted).
10. Cabedo R., Pérez J., Carsí J.A. y Ramos I., "Modelado y Generación de Arquitecturas PRISMA con DSL Tools", en Actas del *IV Workshop DYNAMICA*, Archena, Murcia, España, 2005.
11. Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I., "PRISMANET: Middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos", en Actas de las *X Jornadas de Ingeniería del Software y Bases de Datos*, Granada, España, 2005, pp. 27-34.
12. Meta-Object Facility: MOF, http://www.omg.org/mof
13. Query/View/Transformations: OMG Document ad/2005-07-1.
14. Cabello M.E., Gómez M., LLavador M, and Ramos I. ProtoBOM: a framework that semiautomatically generates Decision Support Systems based on Software Product Lines. *Technical report*: DSIC II/02/08, Polythecnical University of Valencia, pages 68, 2008.

Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together

Abel Gómez[1], Isidro Ramos[2]
Department of Information Systems and Computation
Universidad Politécnica de Valencia
Valencia, Spain
[1]agomez@dsic.upv.es
[2]iramos@dsic.upv.es

***Abstract*—Feature Modeling is a technique which uses a specific visual notation to characterize the variability of product lines by means of diagrams. In this sense, the arrival of metamodeling frameworks in the Model-Driven Engineering field has provided the necessary background to exploit these diagrams (called feature models) in complex software development processes. However, these frameworks (such as the Eclipse Modeling Framework) have some limitations when they must deal with software artifacts at several abstraction layers. This paper presents a prototype that allows the developers to define cardinality-based feature models with constraints. These models are automatically translated to Domain Variability Models (DVM) by means of model-to-model transformations. Thus, such models can be instantiated, and each different instantiation is a configuration of the feature model. This appproach allows us to take advantage of existing generative programming tools, query languages and validation formalisms; and, what is more, DVMs can play a key role in MDE processes as they can be used as inputs in complex model transformations.**

***Keywords*-Software Product Lines; Model Driven Architecture; Feature Modeling; UML; OCL**

I. Introduction

The key aspect of Software Product Lines (SPL) [6] that characterizes this approach against other software reuse techniques is how to describe and manage variability. Although several approaches have addressed this problem, the most of them are based on feature modeling, proposed in [12]. In this approach, the commonalities and varibilities among the products of a SPL are expressed by means of the so-called features (*user-visible aspect or characteristic of the domain*), which are hierarchically organized in feature models.

The use of feature models can be exploited by means of metamodeling standards. In this context, the Model-Driven Architecture [14] proposed by the Object Management Group is a widely used standard which arises as a suitable framework. In this sense, the Meta Object Facility (MOF) and the Query/Views/Transformations (QVT) standards allows us to define feature models and their instances, and use them in a Model-Driven Engineering (MDE) process. In this context, MDE and the Generative Programming approach [7] provides a suitable basis to support the development of SPLs. Moreover, Generative Programming and SPLs facilitate the development of software products for different platforms and their use under different technologies.

In this paper we discuss the main issues that arise when trying to use feature models in a MDE process, and how to easily overcome them. We also present a tool that allows the developers of SPLs to define, use and exploit feature models in a modeling and metamodeling tool. In our case, we have chosen the Eclipse Modeling Framework (EMF). Moreover, the EMF framework provides several tools which permit us to enrich these models (by means of OCL expressions and OCL interpreters) and to deal easily with them (by using model transformations). All these features of EMF allows us to use this framework to start a Software Product Line.

The remainder of this paper is structured as follows: in section II we present the starting point of our work and the main problems that arise when trying to use feature models in a MDA process; and in sections III and IV we present both the ideal approach and the practical approaches to use feature modeling in a MDA context. In section V we show our feature metamodel and how we operationalize the solution presented in section IV in our prototype tool. Related works are discussed in section VI and in section VII we present our conclusions and future works.

II. Foundations

A. Cardinality-based feature models at a glance

Cardinality-based feature modeling [8] integrates several of the different extensions that have been proposed to the original FODA notation [12]. In this sense, a cardinality based feature model is also a hierarchy of features, but the main difference with the original FODA proposal is that each feature has associated a *feature cardinality* which specifies how many clones of the feature are allowed in a specific configuration. Cloning features is useful in order to define multiple copies of a part of the system that can be differently configured.

Moreover, features can be organized in *feature groups*, which also have a *group cardinality*. This cardinality restricts the minimun and the maximun number of group members that

Gómez, A., Ramos, I. Cardinality-based feature models and Model-Driven Engineering: fitting them together. Proc. of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. 2010.

can be selected. Finally, an *attribute type* can be specified for a given feature. Thus, a primitive value for this feature can be defined during configuration.

In feature models is also quite common to describe constraints between features such as the *implies* and the *excludes* relationships, which are the most common used ones. According to the original FODA notation, these constraints can be expressed by means of propositional formulas [2], thus, it is possible to reason about the satisfiability of the feature model. As explained in [8], this interpretation for feature models is not very adequated when dealing with cardinality-based feature models due to the fact that we can have multiple copies of the same feature. Therefore, it is necessary to clearly define the semantics of the constraint relationships in the new context, where features can have multiple copies, and features can have an attribute type and a value. In this case, we need more expressive approaches to (i) define constraints between features and (ii) perform formal reasoning over the feature models and their constraints.

B. Feature models and their configuration

A configuration of a feature model is usually defined as *the set of features that are selected from a feature model without violating any of the constraints defined in it*, but it can also be defined as *a valid set of instances of a feature model*. I.e., the relationship between a feature model and a configuration is comparable to the relationship between a class and an object. The first definition, is well-suited when dealing with "traditional" feature models (those that can be defined by using the original FODA notation). In this case, every instantiation of the elements of the feature model will follow the *singleton pattern*, that is, every feature can have at most one instance. Fig. 1 shows an example of this.

In Fig. 1a an example feature model is represented. This feature model represents a system S, with two features A and B. The first one, feature A, is mandatory (it must be included in every possible product of the product line), and the second one, feature B, is optional (it can be included in a particular product or not). Thus, we have two possible configurations for this feature model, which are represented in figures 1b and 1c.

As can be seen, this process of selection of features (according to the defined constraints) is closely related with a copy mechanism, that is, a configuration of a feature model is a more restrictive copy of the original one that represents exactly one variant. This mechanism can also be used in cardinality-based feature models. In this case, when features have an upper bound greater than 1 they can be cloned at model level, thus, we can have mutiple *copies* of the same feature. Several specializations can be done at model level until only one variant is possible.

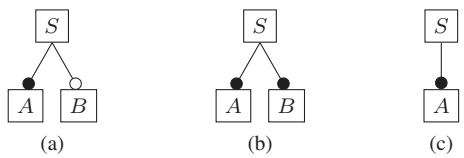


Fig. 1. Example of a feature model (1a) and the two possible configurations that it represents (1b and 1c).

C. Cardinality-based feature models and MDE main issues

In previous paragraphs we have described what cardinality based feature models are and how they are usually exploited. Nevertheless, there are some issues, that we will address in the remaining of this paper, that cause some problems when trying to use feature models in a MDE process.

The first one comes from the classical definition of configuration of a feature model (the set of features that are selected from a feature model). This definition tends to define the configuration as a *copy* mechanism instead of as an *instantiation* mechanism. Although this definition can be somewhat intuitive when dealing with traditional feature models, it can be confusing when dealing with cardinality-based ones. In this case, the instantiation concept is best suited when talking about cardinality-based feature models with attribute types, given that the configuration is more easily understandable as an *instance-of* relationship rather than as a *copy-and-refinement-of* relationship.

The second issue that comes up is how to deal with model constraints when features can be cloned in our models and such features can have an attribute type. In this case, as pointed out in section II, we need to use more expressive languages that allows us (i) to deal with sets of features (i.e. n copies of feature A) and (ii) to deal with typed variables which values can be unbounded in order to easily represent attribute types.

III. Putting Model-Driven Engineering and Feature Modeling together

The Meta Object Facility standard (MOF, [16]), which provides support for meta-modeling, defines a strict classification of software artifacts in a four-layer architecture (from M3 to M0 layer). The meta-metamodel layer (M3) is where the MOF language is found. MOF can be seen as a subset of the UML class diagram, and can be used to define metamodels at M2 layer. This way, artifacts that reside in layer x, are expressed in terms of the constructors defined in layer $x + 1$.

In turn, the Query/Views/Transformations (QVT, [15]) standard describes how to provide support to queries and model transformations in a MDE process. QVT uses the pre-existent *Object Constraint Language* (OCL, [17]) language to perform queries over software artifacts. The QVT standard provides three different languages to describe these transformations. In this sense, the QVT–Relations can be interesting because of its implicit support for traceability and its high level of abstraction, as it is a declarative language. This language uses object templates to define relationships among different domains that can be enforced, performing a model transformation when needed.

As the MOF standard provides support for modeling and metamodeling, we can use it to define cardinality-based feature models by defining its metamodel. Previousy, a configuration

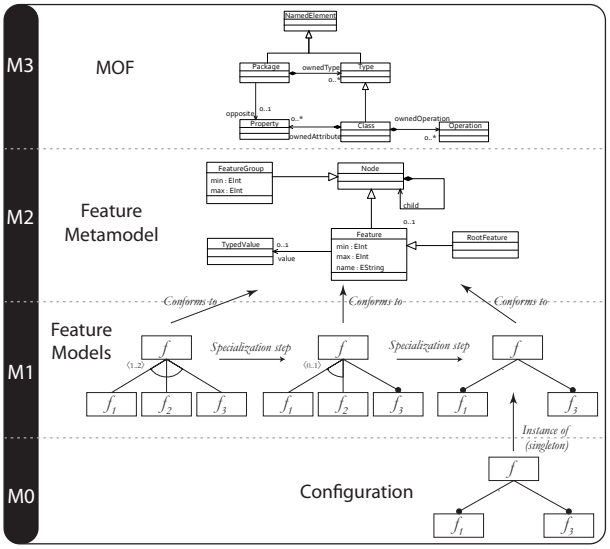


Fig. 2. Specialization and configuration of feature models in the context of MOF

process by means of specialization was shown. This conception about the configuration process involves copy of features, but it has a big implication: configurations are expressed in terms of the feature metamodel instead of in terms of the feature model. Fig. 2 shows how the specialization process fits in the four-layer architecture of MOF. In this figure the EMOF language is represented in a simplified way in the level M3. In the level M2 the metamodel for cardinality-based feature models is represented by using the MOF language (also in a simplified way), and finally, in level M1 some feature models are represented. The leftmost model is the one that represents our family of systems, and starting from it, we obtain the final model as an specialization of the original one. Thus, the configuration of the feature model is defined in terms of the features metamodel as the model it conforms to has no variability, and both feature model and configuration are practically equivalent.

In order to use a feature model in a MDE process, we need to use the one that captures the whole variability of the domain. This is necessary to define any possible configuration of the model, but it is also necessary in order to define model-based transformations that allows us to use this feature models and their configurations in other complex processes.

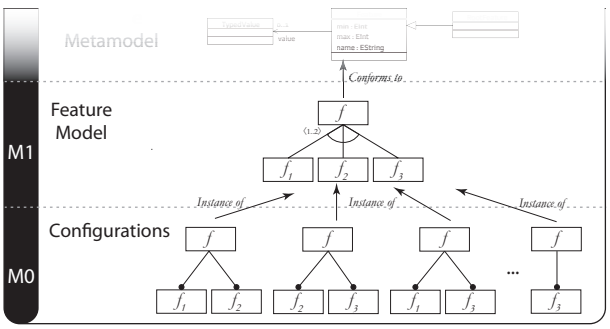


Fig. 3. Instantiation of feature models

Fig. 3 shows how a configuration without performing a specialization process looks like. This way, the feature model can be used in any modeling framework to automatically generate configuration editors, and what is more, feature models and configurations can take part of a MDE process. Developers can take advantage of related tools, feature models can be used to guide model transformations with multiple inputs, and configurations can be automatically checked againts their corresponding models using built-in query languages.

IV. Using a modeling framework for feature models and their configurations

The Eclipse Modeling Framework (EMF) [10] can be considered as an implementations of the MOF architecture. Ecore, its metamodeling language can be placed at layer M3 in the four-layer architecture of the MOF standard. By means of Ecore, developers can define their own ecore models which will be placed al the metamodel layer (M2). An example of such metamodels can be the metamodel to build cardinality-based feature models. Finally, this Ecore models can be used to automatically generate graphical editors which are capable of building *instance models*, which will be placed at M1 layer. In the case of feature modeling, these *instance models* are the feature models. The left column on Fig. 4 shows this architecture.

As can also be seen in Fig. 4, the M0 layer is empty. That is a limitation of most of the modeling frameworks which are available today. As said, EMF provides a modeling language (Ecore) that can be used to define new models and their instances. This approach only covers two layers of the MOF architecture: the metamodel and the model layers. However, in the case of feature modeling we need to work with three layers of the MOF architecture: metamodel (cardinality-based feature metamodel), model (cardinality-based feature models), and instances (configurations).

Fig. 4 shows how to overcome this drawback: it is possible to define a model-to-model transformation in order to convert a feature model (i.e. the model represented by Feature model which can not be instantiated) to an Ecore model (i.e. the Domain Varibility Model, DVM, which represents the Feature model as a new class diagram). Thus, it is possible to represent a feature model at the metamodeling layer, making

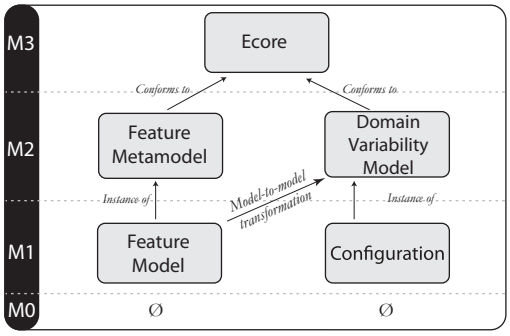
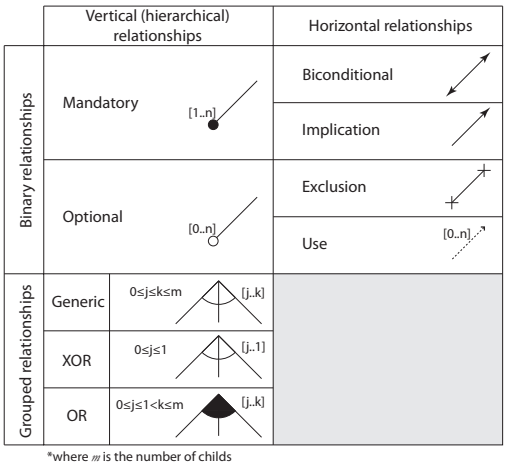


Fig. 4. EMF and the four-layer architecture of MOF

TABLE I
CARDINALITY-BASED FEATURE METAMODEL: PROPOSED TYPES OF RELATIONSHIPS BETWEEN FEATURES



*where m is the number of childs

the definition of its instances possible. This way, developers can take advantage of EMF again, and automatically generate editors to define feature model configurations, and validate them against their corresponding feature models thanks to their new representation, the DVM.

V. OUR APPROACH

Based on the concepts presented in the previous section and using EMF, we have developed a tool that allows us to automate several steps in order to prepare a feature model that can be exploited to develop a SPL in the context of MDA. In this sense, our tool provides:

- Graphical support to define (a variant of) cardinality-based feature models.
- Automatic support to generate Domain Variability Models from feature models that capture all the variability of the application domain, allowing the developers to use them in model transformations.
- Automatic support for configuration editors, which will assist the developers in the task of defining new configurations.
- Capabilities to check the consistency of a configuration against its corresponding feature model.

A. Cardinality-based feature metamodel

The basis of our work is the cardinality-based feature metamodel, which permits to define feature models. In our proposal we have decided to represent explicitly the relationships between features. Thus, our metamodel represents in an uniform way the hierarchical relationships and the restrictions between features. Table I classifies and summarizes the types of relationships that the feature metamodel is able to represent. As can be seen, relationships are classified in two orthogonal groups:

- *Vertical vs. horizontal relationships.* Vertical relationships define the hierarchical structure of a feature model and horizontal relationships define dependecies and restrictions between features.
- *Binary vs grouped relationships.* Binary relationships define relationships between two single features. In turn, grouped relationships are a set of relationships between a single feature and a group of childs.

Given this classification, the following relationships exist:

- *Binary and vertical relationships*. This relationships define structural relationships between two single features. In our approach, they represent a *has_a* relationship between a parent and a child feature. They can be mandatory and optional depending on the lower bound value. The upper bound (n) can be on both cases 1 or greater than 1, and indicates how many instances of the child feature will be allowed.
- *Grouped and vertical relationships*. Grouped and vertical relationships are a set of binary relationships where the child features share a *is_a* connotation with respect to their parent feature. A group can have an upper and a lower bound. These bounds specify the minimun and the maximun number of features that can be instantiated (regardless of the total number of instances).
- *Binary and horizontal relationships*. These relationships are specified between two features and do not express any hierarchical information. They can express constraints (biconditional, implications and exclusion) or dependencies (use). The first group applies to the whole set of instances of the involved features, however, the second one allows us to define dependencies at instance level, i.e.:
  - *Implication* ($A \longrightarrow B$): If an instance of feature A exists, at least an instance of feature B must exist too.
  - *Coimplication* ($A \longleftrightarrow B$): If an instance of feature A exists, at least an instance of feature B must exist too and vice versa.
  - *Exclusion* ($A \nleftrightarrow B$): If an instance of feature A exists, can not exist any instance of feature B and vice versa.
  - *Use* ($A \dashrightarrow B$): This relationship will be defined at configuration level, and it will specify that an specific instance of feature A will be related to one (or more) specific instances of feature B as defined by its upper bound (n).

Fig. 5 shows our feature metamodel. Such metamodel has been defined taking into account that every element will have a different graphical representation. This way, it is possible to automatically generate the graphical editor to draw feature models based on such metamodel. In that figure, a feature model is represented by means of the FeatureModel class, and a feature model can be seen as a set of Features and the set of Relationships among them. A feature model must also have a root feature, which is denoted by means of the rootFeature role.

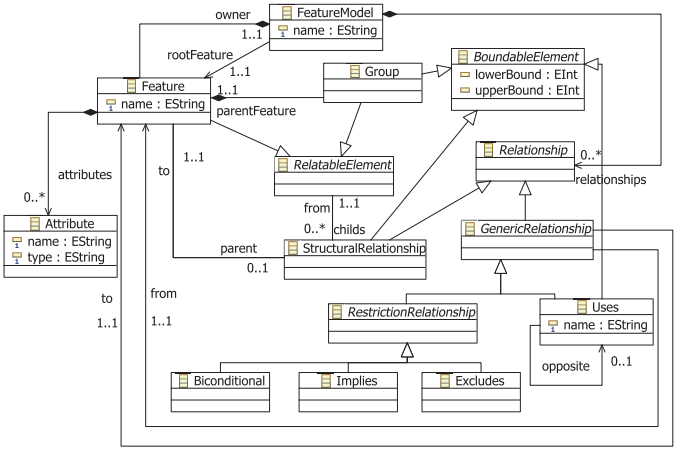


Fig. 5. Cardinality-based features metamodel

Binary relationships in table I are represented in the features metamodel as descendants of the Relationship class. Class StructuralRelationship represents the so called *Vertical* relationships and GenericRelationship represents the *Horizontal* ones. StructuralRelationships relate one parent RelatableElement (a Feature or a Group) with one child Feature. A Group specifies that a set of StructuralRelationships should be considered as a group.

It is noteworthy to point out two slight differences of our approach with respect to the classical cardinality-based feature models. First, we represent feature multiplicities at relationship level instead of at feature level (by means of the BoundableElement class). This allows us to easily define mandatory and optional relationships explicitly. Second, features can not have an attribute type. In turn, this information is expressed in terms of feature attributes. Feature attributes express information which is complementary to a feature and can be used to describe *parametric features*.

B. Cardinality-based feature modeling editor

The cardinality-based feature modeling editor allows us to easily define new feature models. Following the Model-Driven Software Development (MDSD) approach, it has been automatically generated from the metamodel presented in the previous section. To obtain this graphical editor, the Graphical Modeling Framework (GMF [9]) has been used.

Fig. 6 shows what this editor looks like. The palette is located on the right side of the figure, and shows the tools that can be used to define the feature models. In the canvas an example feature model is shown. This feature model describes a product line for mobile phones. A mobile phone must have a screen, which can be touchscreen or not. Touchscreens can use resistive or capacitive technology. Moreover, a mobile phone can also support handwriting recognition, however, this feature is incompatible with normal screens. This can be described by means of an excludes relationship (solid line with crosses at its ends between feature Normal and HandwrtingRecognition). In our product line, mobile phones can also have one or two cameras, each one with their corresponding resolutions. Those cameras can be used for videoconferencing or digital photography. If the camera is used for digital photography, it can be associated to an optional flash light. This is specified by means of the dashed line between features Camera and Flash. Finally, these mobile phones can have FM radio support, but it can be installed only if the device has a headphones connection, as the FM antenna is part of this accessory. This dependency is expressed by the implies relationship (directed line between feature Radio and feature Headphones).

C. Generating the Domain Variability Model

In section IV was explained that it is necessary to execute a model-to-model transformation in order to easily define configurations of a feature model in EMF. Following the MDA approach, this transformation has been defined by using the Relations language defined in the QVT standard. In order to integrate and execute the transformation process in our

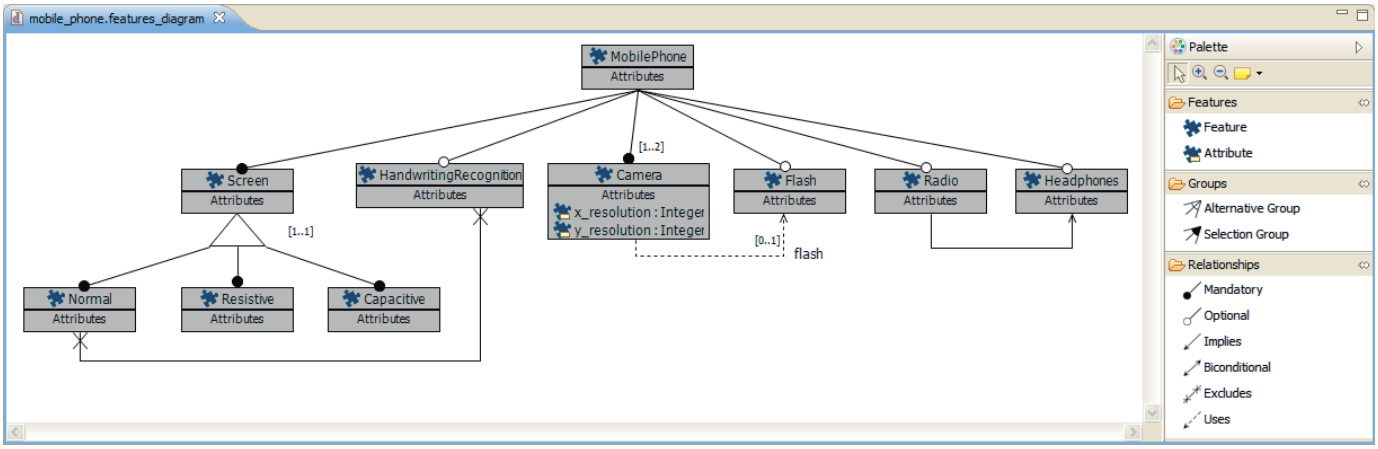


Fig. 6. Screenshot of the cardinality-based feature modeling editor

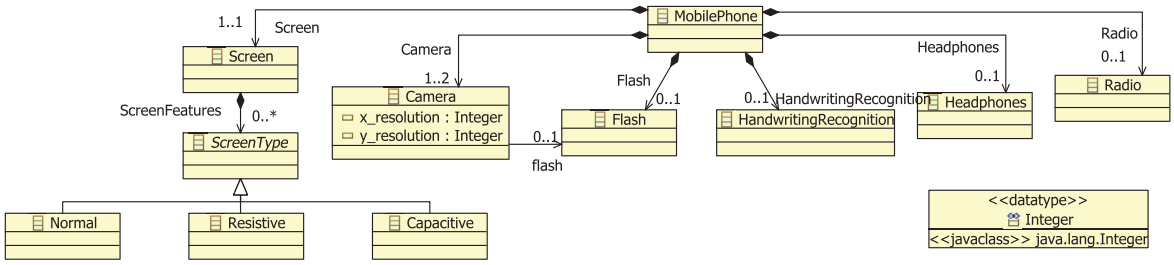


Fig. 7. Example Domain Variability Model

prototype, a custom tool based on the mediniQVT [11] transformations engine has been built.

The following paragraphs describe the transformation which transforms a feature model (expressed as an instance of the cardinality-based features metamodel in EMF), to a Domain Variability Model (DVM, expressed as an Ecore model that can be instantiated). The transformation is declared as follows and it is executed from the *feature* domain to the *classdiagram* domain.

```
transformation Feature2ClassDiagram(feature :
    features, classdiagram : ecore) { ... }
```

As feature models describe not only the structure of the features but also the relationships among them, it is necessary to define rules to generate both the structure of the DVM and the restrictions that apply to it. First, the structure of the DVM is defined by means of Ecore containment references and inheritance relationships; and second, restrictions are defined by means of OCL expressions. These OCL expressions are included on the DVM itself by means of *EAnnotations*. This *EAnnotations* are automatically used in next steps by our prototype to check that configurations are valid.

Figure 7 shows the resulting Ecore model. The rules that have been applied are:

- *Feature2Class*. For each Feature of the source model an *EClass* with the same name of the feature will be created. All these classes will be created inside the same EPackage, whose name and identifier derives frome the feature model name. All the features in Fig. 7 are example of the aplication of this rule.
- *FeatureAttribute2ClassAttribute*. For each feature Attribute, an EAttribute will be created in the target model. This EAttribute will be contained in its corresponding EClass. Any needed EDataType will be also created. Attributes in the Camera EClass are example of this.
- *StructuralRelationship2Reference*. For each StructuralRelationship from a parent Feature a *containment* EReference will be created from the corresponding EClass (i.e. same name than the Feature). The multiplicity of this EReference will be the lower and upper bounds of the StructuralRelationship. *Containment* EReferences from MobilePhone are example of the application of the rule.
- *Group2Reference*. This rule states that for each Group contained in a Feature a *containment* EReference will be created from the corresponding EClass (i.e., same name than the Feature). This EReference will point to a new abstract class, whose name will be composed by the Feature name and the suffix "Type". This rule has the following post-conditions: *GroupChild2Class*, *Group2ChildsAnnot*, *GroupChild2LowerAnnot* and *GroupChild2UpperAnnot*. Screen, ScreenType and the EReference ScreenFeatures are example of the result produced by this rule.
- *GroupChild2Class*. This rule is in charge of creating the EClasses from the Features belonging to a Group. Moreover, each one of these EClasses inherit from the abstract *EClass* that has been previously created. EClasses Normal, Resistive, Capacitive, and their inheritance relationship are example of the aplication of the rule.
- *Group2ChildsAnnot*, *GroupChild2LowerAnnot*, and *GroupChild2UpperAnnot*. These rules create EAnnotations that will contain OCL expressions. This expressions will check that the multiplicities specified for the Group and the child Features are also satisfied by the instances of the DVM.
- *UsesRelationship2Reference*. For each Uses relationship between two Features, an EReference will be created in the target model. This EReference will relate two EClasses whose names will match the Features names. The EReference flash, between Camera and Flash, shows and example of this.

The next relations generate OCL expressions in the DVM for each restriction relationship of the source model. They can be easily expressed as the following OCL invariants created in the root EPackage:

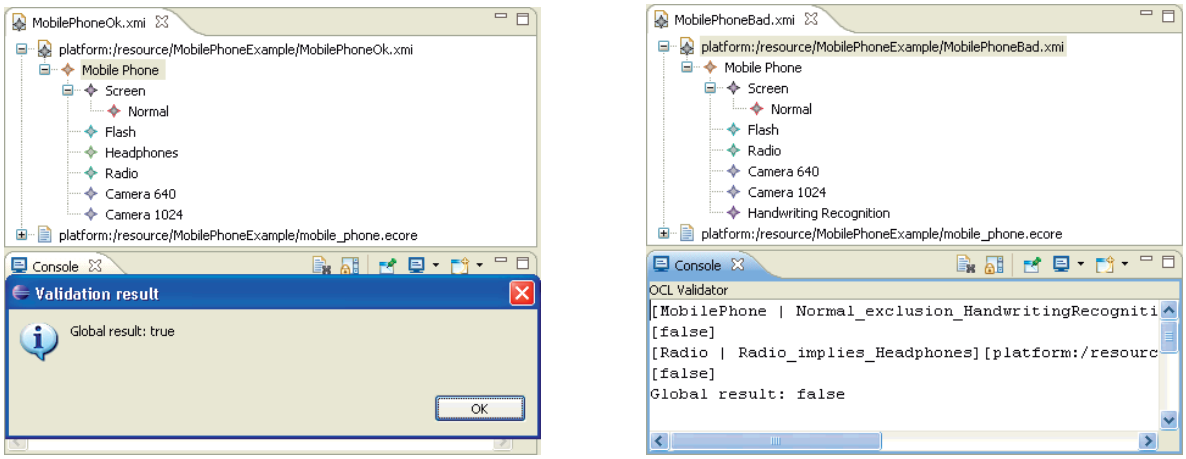
- *ExcludesRelationship2ModelConstraint*. This rule generates the following invariant for (`A excludes B`):

```
A.allInstances()->notEmpty() implies B.
    allInstances()->isEmpty()) and
B.allInstances()->notEmpty() implies A.
    allInstances()->isEmpty())
```

- *ImpliesRelationship2ModelConstraint*. If the relationship is (`A implies B`), the following OCL expresion is created by this rule:

```
A.allInstances()->notEmpty() implies B.
    allInstances()->notEmpty())
```

- *BiconditionalRelationship2ModelConstraint*. This rule creates the following OCL invariant if the source relationship is (`A if and only if B`):



(a) (b)

Fig. 8. Example of successful (8a) and unsuccessful (8b) configuration check

```
A.allInstances()->notEmpty() implies B.
    allInstances()->notEmpty()) and
B.allInstances()->notEmpty() implies A.
    allInstances()->notEmpty())
```

D. Creating and validating configurations

In order to create new configurations of feature models it is not necessary to use any custom tool. As far as we have a DVM which captures the same varibility than the original feature model, developers can use the standard Ecore tools. The most straightforward method to create a new configuration is to use the "Create Dynamic Instance…" option of the standard "Sample Ecore Model Editor". This way, the "Sample Reflective Ecore Model Editor" can be used to define new configurations of our feature model. However, although EMF provides all the libraries and technologies to exploit OCL expressions, there is not a default method to check OCL invariants which are directly stored as EAnnotations in Ecore models themselves. Thus, we have built an extension which can take advantage of the OCL invariants that have been automatically created in the previous transformation step.

Fig. 8a shows an example configuration. A mobile phone with a normal screen, radio, headphones and two cameras has been defined. This configuration is valid conforming to the example feature model, as the popup window in the figure shows. However, Fig. 8b shows a configuration with a normal screen and handwriting recongnition, which violates the excludes relationsship. Moreover, this configuration includes radio support, but it does not include the headphones connection which is also invalid. When the configuration is invalid the checking process is unsuccessful. In this situation, the prototype console shows a summary with the constraints that are not met, and which are the problematic elements.

VI. Related works

Feature modeling has been an important discussion topic in the SPL community, and a great amount of proposals for variability management have arised. Specially, most of them are based in the original FODA notation and propose several extensions to it [5]. Our work is closely related with previous research in feature modeling, however, there are several distinctive aspects:

In [8] a notation for cardinality-based feature modeling is proposed. In this sense, our tool shares most of this notation as it is widely known and used, but we have included some variants. First, in our aproach features can not have an *attribute type*, but rather, they can have typed *feature attributes* which can be used to describe *parameterized features*. Second, in [8] both feature groups and grouped features can have cardinalities. However, the possible values for grouped features cardinalities are restricted. In our proposal, these values are not restricted and have different meanings: cardinality of feature groups specify the number of features that can be instantiated, and cardinality of grouped features specify the number of instances that each feature can have.

Our work describes a prototype to define configurations of feature models. Previous work has been also done in this area, such as the *Feature Modeling Plugin* [1]. This tool allows the user to define and refine a feature model and configurations by means of specializations. The advantage of this approach is that it is possible to guide the configuration process by means of constraint propagation techniques. The main difference with our work is that configurations are defined in terms of the feature metamodel and both models and configurations coexist at the same layer. Thus, in order to be able to deal both with models and configurations it is necessary to build complex editors (as they must guarantee that the specialization process is properly done).

Some previous works have already represented feature models as class diagrams. In [8] the translation from feature models to class models is performed manually, and no set of transformation rules are described. In this work, OCL is also presented as a suitable approach to define model constraints, but as the correspondences between feature models and class diagrams are not precisely defined, there is no automatic

generation of OCL invariants. In turn, [13] do present a set of QVT rules to automatically generate class diagrams from feature models. However, in this case, neither model constraints nor configuration definitions support is presented.

In [2] a proposal for feature constraints definition and checking is done. Specifically, this work proposes to represent features as propositions and restrictions among them as propositional formulas. However, in propositional formulas only `true` and `false` values are allowed. This approach is not suitable to our work, as we can have typed attributes which can not be expresed by this kind of formulas. Thus, we state that more expressive languages are needed. In this case, we propose OCL as our constraint definition language. Nevertheless, in order to perform more advanced reasoning (for example, satisfiability of feature models) richer formalisms are needed.

VII. Conclusions

In this paper we have proposed a prototype to define and use feature models in a MDE process. This prototype addresses one of the main issues that arises when dealing with nowadays metamodeling tools: they usually are not able to deal simultaneously with artifacts located in all the MOF layers. For example, the selected modeling framework (EMF) is only able to represent three different MOF layers (from M3 to M1), so that, our prototype defines a mechanism to overcome this typical limitation by means of model transformations. This way, feature models can be transformed to Domain Variability Models that can be instantiated and reused in future steps of the MDE process.

Our tool has been also designed following the MDE principles and a metamodel for cardinality-based feature modeling has been defined. Thus, by means of generative programming techniques, a graphical editor for feature models has been built. Feature models defined with this editor are automatically transformed in DVMs that are used to define configurations of feature models. Although several tools to define feature models and configurations in the last years have arised, our approach has several advantages against previous approaches: (i) the infrastructure that we propose to build configurations is simpler and more maintainable, as it is built following the MDSD guides; (ii) configurations are actually instances of a feature model (expressed by means of the DVM), so we can take advantage of the standard EMF tools; (iii) as feature models are described by DVMs that can be instantiated, both models and configurations can be used in other MDE tasks; (iv) having a clear separation between feature models and configuration eases the validation tasks as they can be performed by means of built-in languages; and (v) as the transformation between feature models and DVMs is performed automatically by means of a declarative language we can trace errors back from DVMs to feature models.

It is important to remark that having both feature models and configurations at different layers is very useful as they can easily be used in model transformations. In [3] an example is shown. In this work, a model transformation with multiple inputs is used to generate a software architecture automatically. In this case, one of the arguments of this transformation is a configuration of a feature model, which is used to guide the architecture generation process.

Finally, to use DVMs allows us to address some satisfiability problems from new points of view. The introduction of cardinalities and unbounded attribute types makes harder to reason about feature models. Thus, richer formalisms (compared with the traditional ones) are needed. Fortunately, class diagrams are widely used and known, and several formalisms to reason about them have been proposed. In this sense, we have already done some preliminary works in model consistency checking by using formal tools [4], and future works are oriented towards this direction. A first version of our prototype can be downloaded from `http://issi.dsic.upv.es/~agomez/feature-modeling`.

Acknowledgments

This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation MULTIPLE TIN2009-13838 and the FPU fellowship program, ref. AP2006-00690.

References

[1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. *2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, 2004.

[2] D. Batory. Feature models, grammars, and propositional formulas. pages 7–20. Springer, 2005.

[3] M. E. Cabello, I. Ramos, A. Gómez, and R. Limón. Baseline-oriented modeling: An mda approach based on software product lines for the expert systems development. *Intelligent Information and Database Systems, Asian Conference on*, 0:208–213, 2009.

[4] J. Cabot, R. Clarisó, and D. Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548, NY, USA, 2007. ACM.

[5] L. Chen, M. A. Babar, and N. Ali. Variability management in software product lines: A systematic review. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*, 2009.

[6] P. Clements, L. Northrop, and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.

[7] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[8] K. Czarnecki and C. H. Kim. Cardinality-based feature modeling and constraints: A progress report, October 2005.

[9] Eclipse Organization. The Graphical Modeling Framework, 2006. http://www.eclipse.org/gmf/.

[10] EMF. http://download.eclipse.org/tools/emf/scripts/home.php.

[11] ikv++ technologies AG. ikv++ mediniQVT website. http://projects.ikv.de/qvt.

[12] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[13] M. A. Laguna, B. González-Baixauli, and J. M. Marqués Corral. Feature patterns and multi-paradigm variability models. Technical Report 2008/01, Grupo GIRO, Departamento de Informática, may 2008.

[14] Object Management Group. MDA Guide Version 1.0.1. 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[15] Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[16] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[17] Object Management Group. OCL 2.0 Specification. 2006. http://www.omg.org/cgi-bin/doc?formal/2006-05-01.

BOM–Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques

Abel Gómez
Dep. of Information Systems and Computation
Universidad Politécnica de Valencia
Valencia, Spain
agomez@dsic.upv.es

Mª Eugenia Cabello
Faculty of Telematics
Universidad de Colima
Colima, México
ecabello@ucol.mx

Isidro Ramos
Dep. of Information Systems and Computation
Universidad Politécnica de Valencia
Valencia, Spain
iramos@dsic.upv.es

***Abstract*—This paper presents Baseline Oriented Modeling–Lazy (BOM–Lazy): an approach to develop applications in a domain, Expert Systems, by means of Software Product Lines and model transformations techniques. A domain analysis has been done on the variability of Expert Systems that perform diagnostic tasks in order to determine the general and individual features, (i.e. common and variants features) of these systems. The variability of our Software Product Line is managed by means of models and model transformations; and the Production Plan is automatically generated and driven by the variability model and the core assets (the generic architecture) of the domain, in order to produce the base architecture of the Software Product Line.**

***Keywords*-Software Product Lines; Variability Management; Model Transformations; Software Architectures; Diagnostic Expert Systems**

I. Introduction

Expert Systems (ES) are a family of software products that are gaining great importance in recent years [13]. They are applications that can be used to simulate the behaviour of experts when they solve a problem in an specific domain. Such systems try to capture the knowledge and decisions of experts, improving the speed and the quality of the answers that are obtained, and increasing the productivity of the domain experts. ES have been widely used in several fields of our lives: medicine, education, military intelligence, aeronautics, archeology, agriculture, law, geology, electronics, computer science, telecommunications, etc. That is why they are becoming an important reference point in decision-making processes, especially in systems in charge of diagnostic tasks. However, the development of this kind of systems is complex because the basic elements that conform their architecture vary not only in their behaviour, but also in their architecture. That is why there is an increasing need in supporting them properly.

The changing nature of technology leads us to need multiple versions of the same or similar application in short time periods. Because of that, Software Engineering must provide the tools and methods which allow us to develop a family of products with different capabilities and adaptable to changeable situations, in place of developing only a single product. Under these circumstances, the Software Product Line (SPL) [8] concept arises with the aim of controlling and minimizing the high costs of the software development process. This approach is based on the creation of a design that can be shared among all the members of a family of programs within an application domain. This way, a design that has been done explicitly for a product can get benefit from the common assets (models, components, code…) that can be reused in different products, reducing costs and development time.

In order to develop an ES, the SPL approach is more adequate than the traditional one: this family of systems involve a wide range of application domains, with different features, that vary from a case study to another. Moreover, SPL make easier the development of software products for several target platforms and technologies.

This work integrates different technical spaces [16]: Expert Systems (ES), Software Product Lines (SPL) and the Model-Driven Architecture (MDA, [18]), as the technical framework. MDA promotes to separate the description and the functionality of a system (Platform Independent Model, PIM) from its implementation (Platform Specific Model, PSM). MDA suggests to define and use models at different abstraction levels as the main assets of the software development process. These models can be manipulated and refined (by means of model transformations) in successive steps to obtain the final implementation of the system. The standard that the OMG proposes to define these model transformations is the Query / Views / Transformations (QVT, [19]) standard. This way, in our approach we use the tools that are supported on the MDA framework to represent and manage the variability of the SPL by means of domain models and their transformation to architectural models. In the end, this infrastructure allows us to produce a SPL of ES at design level using the architectural specification language PRISMA [4] as the target domain. The PRISMA framework [21] automatically compiles the architectural configurations and executes them by using a middleware built on top of the .NET platform.

This document is structured as follows: section II de-

Gómez, A., Cabello, M. E., Ramos, I. BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. Proc. of the 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010) at SPLC 2010. 2010.

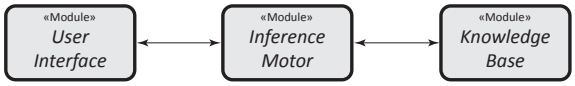


Figure 1. Generic Architecture of Expert Systems

scribes the variability found in the Expert Systems technical space; section III explains how this variability is managed in our approach. Section IV describes how our ideas have been captured in a prototype. Finally, section V describes some related works and section VI presents our conclusions.

II. Variability of the Diagnostic Expert Systems Domain

A subset of the Expert Systems domain has been chosen to describe our approach: the ES that are used in diagnostic tasks, the so-called Diagnostic Expert Systems (DES). The diagnosis of an entity lies on the evaluation of its state by interpreting its properties. A field study about diagnostic systems has been done [6] to identify the key variability points. This study allows us to know the DES behaviour and structure in several specific domains. The remainder of this paper will refer to two paradigmatic cases: systems for medical diagnosis and systems for educational diagnosis.

In the medical diagnosis example, the entity to be diagnosed is the patient and the result of the process is the disease he/she suffers. First, a clinical diagnosis is performed, which must be validated then by a laboratory-based diagnosis. In the end, both diagnosis are merged in a final diagnosis where the previous ones are taken into account. Thus, we can identify three basic functionalities: *get laboratory diagnosis*, *get clinical diagnosis*, and *get diagnosis*. The first one is used by the *laboratory assistant* and the last ones are used by the *doctor*. In this case, the properties of the entities considered in the process vary during the whole process, which implies the existence of several hypotheses that must be evaluated to determine the valid one using differential reasoning.

In the educations reasoning example the entity to be diagnosed is a post-graduate educational program where several quality criteria are evaluated, and the result of the diagnosis is the advance of the given program. The properties of the entities remain the same throughout the diagnostic process, therefore only one hypothesis is created applying deductive reasoning. In this case the DES only performs one task: *get program advance*, which is invoked by the user of the tool.

A. Diagnostic Expert Systems Generic Architecture

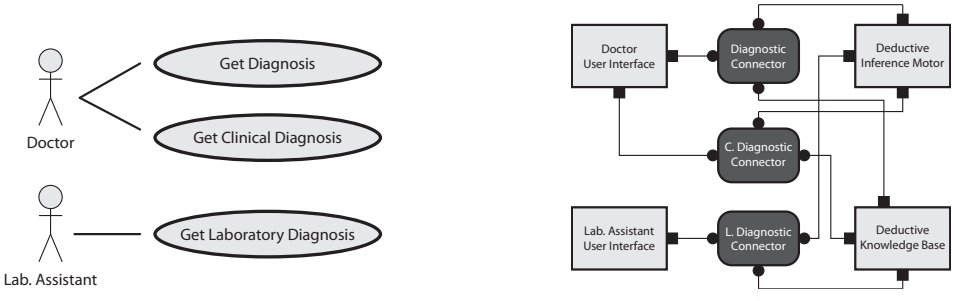
In SPL, there are parts that are shared among all the products, but some other parts vary from one product to another. The common parts are represented by the *generic architecture*, which captures the shared functionality. The variable part shows additional features that are specific for some products, and such parts are represented by the *base architecture*. The *generic architecture* of DES is expressed in our approach by a modular model made up of three basic modules (see Fig. 1):

- *Inference Motor* module. This module contains the inference process that solves a problem in a specific domain.
- *Knowledge Base* module. The *Knowledge Base* module contains the knowledge about the domain
- *User Interface* module. This module allows the communication between the user(s) and the system.

The generic architecture is used as the shared structure of an application that is member of the product line, but, there also exist additional features that are particular to an specific application. This implies the creation of an specific base architecture when a product of the SPL is obtained from the generic architecture. However, the base architecture that is generated from the generic architecture is not unique, because systems vary not only in their structure but also in their behaviour as explained in the following section.

B. Diagnostic Expert Systems Structural Variability

To illustrate how the architectural elements of a DES vary in their structure, we have modeled the functional requirements that the final product must satisfy using UML Case Diagrams. These diagrams show the different functionalities that the user expects from the system and how the system



(a) (b)

Figure 2. Medical diagnosis use case diagram (a) and its corresponding base architecture (b)

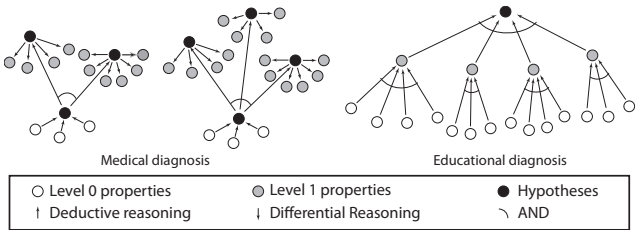


Figure 3. Graph describing the inference processes for medical diagnosis and educational diagnosis

interacts with its environment. In particular, the structure of the architectural elements vary according to the number of use cases, the number of actors and the number of use cases that are accessed by each actor.

Fig. 2 shows the use case diagram for the medical diagnosis domain together with its corresponding base architecture. As Fig. 2a shows, the ES for medical diagnosis of our case study has two actors: *doctor* and *lab. assistant*. The first one uses the system to get clinical diagnosis and the final diagnosis; and the second one uses the system to get the laboratory diagnosis. These use cases affect the final base architecture of the system. Fig. 2b shows, for example, that a user interface module is used for each one of the actors shown in the diagram.

C. Diagnostic Expert Systems Behavioral Variability

Behavioral variability of the architectural elements of an ES depends on the type of diagnosis, and therefore the reasoning to be used. As presented in the previous section, the inference process to apply is defined according to the reasoning (it can be deductive or differential). Moreover, we say that the inference process is *static* if there is only one hypothesis to evaluate and the entities involved keep the same properties throughout the whole diagnostic process. However, if the properties of the entities change during the process and there is more than one hypothesis, we say that it is a *dynamic* process. This way, medical diagnosis is a dynamic process that requires differential reasoning (Fig. 3-left); but educational diagnosis is an static process which requires to apply deductive reasoning (Fig. 3-right) [5]. Thus, the base architecture for ES in the medical diagnosis domain will have a *Differential Inference Motor* component. The behaviour of this component will differ from the behaviour of the inference motor of the ES in the educative diagnosis domain, which will have a *Deductive Inference Motor* component. Fig. 3 represents the medical and educative diagnosis processes as inference graphs.

III. Variability Management

Variability management is the key point to develop our family of DES. Variability among the products of a SPL can be expressed in terms of features [15]. In BOM–Lazy (Baseline Oriented Modeling–Lazy) [6] the observed features in the diagnostic processes and the user requirements are considered as the first group of variability points. Additionally, a second group of variability points arise: the variability that emerges from the application domain's properties. Thus, variability is managed in two stages. The first stage manages variability by using a feature model, which is used to obtain an specific base architecture. The second stage manages the variability of the properties and decorates this specific base architecture with the application domain features, conforming the final product.

A. BOM–Lazy Production Plan overview

The Production Plan of our SPL taking the BOM–Lazy approach is shown in Fig. 4 by using SPEM notation [17]. The Production Plan starts (task 1) when the features of the first variability (the domain variability) are obtained from the application engineer expressed as instances of the Domain Variability Model (DVM). Next (task 2), a skeleton architecture (the base architecture) is built by executing a QVT–Relations transformation using the DVM instances

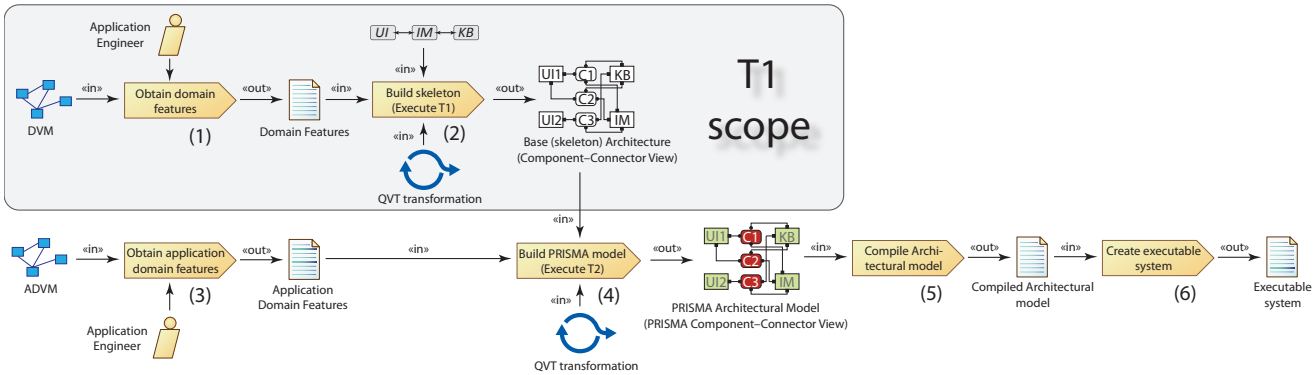


Figure 4. BOM–Lazy Production Plan

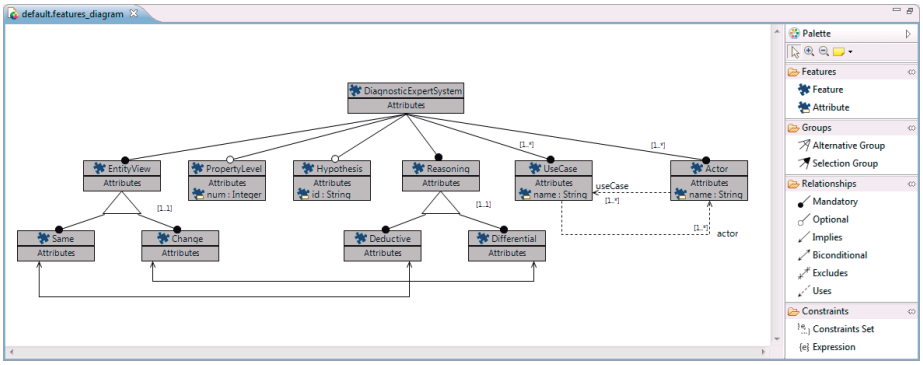


Figure 5. Feature model of the SPL (Domain Variability Model)

and the generic architecture model. This first transformation (called *T1*) creates a base architecture that corresponds to the specific product of our SPL family. Additionally, the Application Domain Variability Model (ADVM) is used to define the features of the application domain (task 3). The instances of the ADVM are used to build the final model (task 4) by executing the QVT–Relations transformation (the second transformation, *T2*). This way, the base architectural model is automatically transformed to a PRISMA architectural model [4]. Finally, the PRISMA architectural model is compiled (task 5) by using the PRISMA-MODEL-COMPILER [21]. This tool automatically generates final product of our SPL: a fully functional C# .NET program that can be directly executed by using the PRISMA-Middleware.

The remainder of this paper will focus only on the first variability (highlighted in Fig. 4 as *T1 scope*), which is related with the diagnostic process and the user requirements, as they define the behaviour and the structure of the ES as described in section II. The management of the second variability will remain unexplained in this paper as it is out of its scope and it is dealt with similar process.

B. Domain Variability Management

The *Feature Model*, our *Domain Variability Model (DVM)*, represents the features of the first variability of the DES (our SPL). The selected variant is represented as an instance of the Domain Conceptual Model (DCM). The DCM is an intermediate model used to capture the selected features of the DVM. These features are used to configure the base architecture. Fig. 5 shows and screenshot of our feature model editor. In the image the feature model of the SPL for DES is described. Our prototype uses a variant of the Czarnecki-Eisenecker notation [10]. This notation for feature modeling allows feature cloning and typed feature attributes. Feature attributes can be used to identify cloned features.

IV. BOM IN PRACTICE

This section describes how to deal with the variability in the development of DES by using SPL and MDA techniques. In BOM–Lazy the transformation in charge of building the base architectures is called “T1”. This transformation is driven by the first variability that was described in section III. The transformation generates an specific base architecture (a component–connector architectural model) from a modular model (the generic architecture, which describes the common parts of the family of DES) and a domain variability model which captures the variability of the specific conceptual domain (DVM) as shown in the highlighted part of Fig. 4.

Fig. 6 depicts a simplified version of the modular view metamodel, which is used to describe the generic architectures. It shows that a model has a set of modules (which can also have different functions) that are linked to other modules by several kinds of relationships (only the Use relationship is represented for simplicity purposes).

To define an specific variant of the SPL for DES, the generic architecture must be configured with the particular features of the conceptual domain. This is done by means of the feature model (DVM) presented in section III, which is finally represented as a UML class diagram (the DCM). This representation allows us to easily define model instances in nowadays modeling tools. The translation of the Feature Model to a class diagram is automatically performed by our prototype and is described in detail in [14]. Fig. 7 shows what this model looks like by using the class diagram representation. Thus, an instance of the DVM is a set of

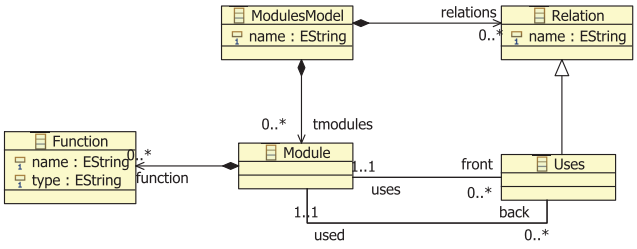


Figure 6. Simplified metamodel of the modular view.

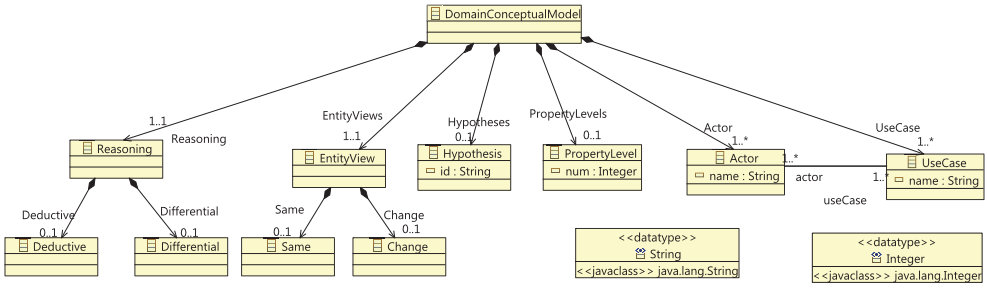


Figure 7. Domain Variability Model expressed as a class diagram (Domain Conceptual Model)

object that are instances of such model.

Finally, Fig. 8 shows the simplified metamodel to build architectural models. An architectural model has a set of components and a set of connectors. Components provide services through a set a ports. Connectors are in charge of linking the different ports of the components by means of their roles.

A. Transformation patterns

In order to define the relationships among the different models and metamodels we have used the Query/Views/-Transformations language (QVT) [19] as proposed by the OMG in their Model–Driven Architecture proposal. The transformation establishes the correspondences among the elements of the source and the target models. Table I describes in a simplified way the relationships that have been identified, together with the elements that each one involves.

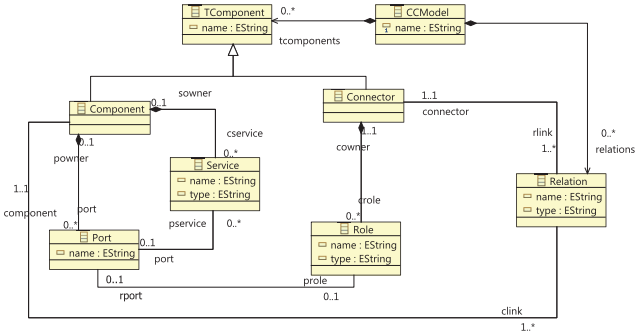


Figure 8. Simplified version of the Component–Connector metamodel

To define the correspondences among the different domain we have taken into account good practices for software development [20] in order to increase the quality of the design of the generated software architecture. Some of this practices and design decisions that the rules describe are the following:

- ***ModulesModel2ComponentsModel***. *A design must define a hierarchical organization that controls software components in a smart way*. In order to satisfy this criterion, this rule transforms the root element of the modular metamodel (ModulesModel), which contains the rest of the elements of the metamodel, to the root element of the component-connector metamodel (CCModel). The rule assigns the name of the source element to the target element.
- ***UseCase2Connector***. *A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.* Use cases constitute a partition of the system based on functionality. Thus, to achieve this requirement the rule transforms each use case to a connector in charge of coordinating the different components of the use case. This way, there is a one-to-one relationship between use cases and connectors.
  Fig. 9 shows the `UseCase2Connector` rule using the graphical notation of the QVT language. This rule creates, for each one of the use cases of the source variability model one connector of the component–connector model. The name of such connector stands for the concatenation of the use case name and the suf-

Table I
TRANSFORMATION RULES AND INVOLVED ELEMENTS

	Involved elements		
Relation name	**Variability Model**	**Modular Metamodel**	**Component-Connector Metamodel**
ModulesModel2ComponentsModel	-	ModulesModel	CCModel
UseCase2Connector	UseCase	-	Connector
Module2Component	Actor, UseCase, EntityViews / Reasoning	Module	Component
Module2RolePort	UseCase	Module	Role, Port
Function2Service	-	Function	Service
Function2Relation	-	Function	Relation

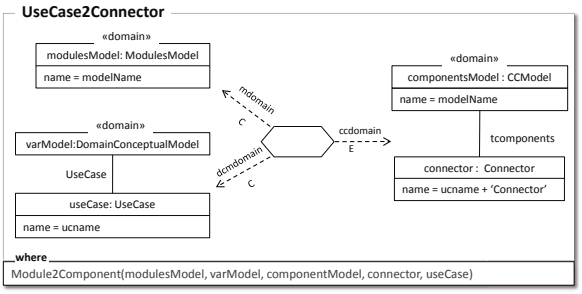


Figure 9. Graphical representation of the UseCase2Connector QVT relation

fix "Connector". The modular domain (`mdomain`) and the variability model domain (`dcmdomain`) are defined as `checkonly` (as the "`C`" attached to the arrow of the domain indicates). This keyword is used to assure that at least an object validating the pattern exists. If no object validating the pattern exists the rule will not be applied (as the transformation is executed in the direction of `ccdomain`). In turn, the component–connector domain (`ccdomain`) is defined as `enforced` (as the "`E`" next to the arrow describes). The `enforce` modifier states that the pattern must be always validated. If there are not objects that make possible to validate the pattern, the rule must create them until the application of the rule is possible. Finally, the `where` clause states that the `Module2Component` relation must be considered as the post-condition of the current rule (`UseCase2Connector`). Thus, this rule should be properly applied after the `UseCase2Connector` relation is checked.

- ***Module2Component***. The generic architecture of ES is usually conformed by three modules. This rule transforms the *Knowledge Base* and *Inference Motor* modules to the *Knowledge Base* and *Inference Motor* components respectively. It also establishes their type (deductive or differential) according to the *Reasoning* and *EntityViews* elements of the variability model. Any other additional module of the generic architecture leads to a new component that represents it using the same name in the final base architecture.
  With the aim of satisfying the following design requirement: *A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment*, the *User Interface* module is transformed into as many components as actors appear in the domain variability model. This way, a relationship one-to-one is defined between the actors of the source model and the user interfaces in the target model.
- ***Module2RolePort***. The criterion *Uniqueness of the knowledge base* indicates that knowledge must be unique for the entire system. In order to meet this requirement, there is only one *Knowledge Base* component in our base architecture. Given that for each use case exists a different view of the *Knowledge Base*, we must merge each one of these views. This is represented in the *Knowledge Base* component by adding a port that allows the communication of such views with the rest of the components. To allow this communication it is also necessary to add the corresponding roles to the adequate connectors. Therefore a one-to-one relationship is defined between use cases and the *Knowledge Base* ports, which also implies a one-to-one relationship between use cases and the roles of the different connectors. This rule also creates the corresponding ports in the *Inference Motor* and *User Interfaces* components, and their corresponding roles in the involved connectors.
- ***Function2Service***. This rule creates a new service on the corresponding component of the Component–Connector model. This service is generated from a given function of a module of the modular model. The new service is created with the same name and type of the source function.
- ***Function2Relation***. This rule states that a function of a module of the source model (ModuleModel) will generate a *Relation* element in the target model (CCModel). This *Relation* will link a connector with its corresponding component in the Component–Connector model.

B. Transformation execution

Our prototype is built on top of the Eclipse platform, and makes use of the *Eclipse Modeling Framework* [12] and the *Graphical Modeling Framework* [11] tools to represent the different models and to build the different editors. Furthermore, it integrates a QVT transformations engine to easily execute the model transformations.

Different graphical editors have been developed to visually represent the models that are involved in the *T1* transformation. Fig. 10 shows a tree editor with the DCM instance of the medical diagnosis example. Fig. 11 shows the second software artifact that takes the T1 transformation: the modular view model (generic architecture of DES). Figure shows an screenshot of the *Modular View editor* of the

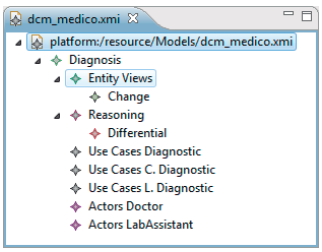


Figure 10. Domain Conceptual Model instance

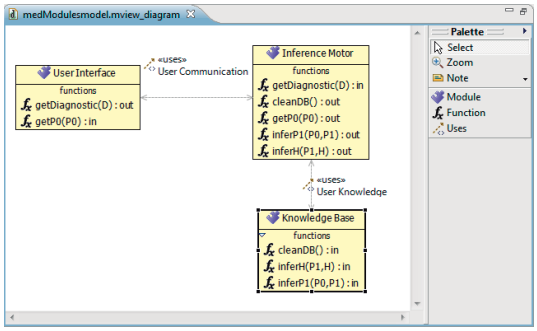


Figure 11. Expert Systems modular miew model

BOM–Lazy prototype. Finally, Fig. 12 shows an screenshot of the result model represented on the *Component–Connector View editor*.

V. Related Works

Software Product Lines have been an important discussion topic in the last decade. There are many studies on this subject. Our research is related to the following SPL ones:

- In [3] Batory et al. capture the domain features in a Feature Model. In BOM we capture features in two kinds of Feature Models. In our research, we observed that the variability problem is not solved by means of a unique feature model and the monotonic gluing of this features. We have taken a new approach, which manage the variability in two phases (one by building a base architecture using the domain features, and another one by decorating these base architectures with the application domain features) in order to obtain the final product.
- In [22] Trujillo uses Feature Oriented Programming (FOP) as a technique for inserting features into XML documents by means of XSLT templates. In BOM-LAZY we use this technique but at the model level (i.e. we use Feature Oriented Modeling) by means of QVT–Relations Transformations. The features are inserted on the skeleton model in order to obtain the PRISMA architecture model.

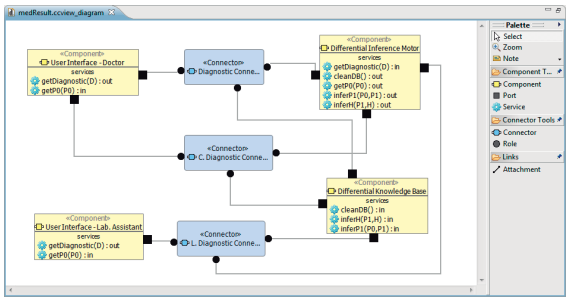


Figure 12. Resulting base architectural model

- Clements and Northrop use in [7] the SPL development approach, considering a clear division between domain engineering and application engineering phases for the reuse and the automation of the software process. In BOM, we have used this approach to develop our SPL.
- Ávila-García et al. [1] use process modeling in SPEM to pack reusable assets. In our approach we use this OMG standard too when we model the Production Plan of the SPL.
- In [2] Bachmann et. al. propose to separate the variability declaration of the affected assets in separate artifacts. In BOM, the specification of the variability and the functionality are captured in different feature models. The use of instances of such feature models allows the user to input the information of the domain features. Those features allow to build the associated assets, or to define the application domain features in order to configure the final application.
- Czarnecki et. al. propose in [9] a notation for cardinality-based feature modeling. In this sense, our work shares most of this notation as it is widely known and used, but we have included some variants. Moreover, we have provided tooling support for this notation and we have integrated it in our prototype.

VI. Conclusions

This paper describes how to develop a Software Product Line in an specific domain (Diagnostic Expert Systems in this paper) by using Model–Driven techniques. The development of such a kind of systems is a complex process because of element that compose their architecture vary not only in their behaviour but also in their structure. This situation implies that several base architectures are obtained on the same generic architecture. Our approach uses QVT–Relations as the model transformations language to manage the variability along the whole process. Our approach also enhances the development of DES by applying SPL techniques, as they are useful when the members of a family of programs share a common design. This way, an specific design can be used in different products, reducing costs, time to market, effort and complexity. By applying MDA techniques, we are able to build systems that are platform-independent, and we can think about them from the problem perspective and not the solution perspective. This makes possible to apply such solutions to different domains. Moreover, we provide a framework with several technical spaces where modern software development techniques coexist in a coordinated way. It is noteworthy to point out that, although this paper only describe the variability management for the first stage of the development process of DES (as shown in section III), the process continues til the base architecture is obtained. The second stage has been also implemented and the base architecture is then decorated with the application domain features. The result of the second stage produces a final and

specific architecture. In our SPL the final architectural model is a PRISMA [4] model. PRISMA is a framework to describe architectural models that provides the PRISMA-MODEL-COMPILER tool [21]. This tool is able to automatically generate executable C#.NET code. This way, our proposal covers the whole development process .

Furthermore, in traditional approaches, the group of base architectures is defined and implemented at design time of the SPL (domain engineering) and it remains unchanged throughout the whole life-cycle of the SPL (application domain). In the BOM–Lazy approach, the use of the T1 transformation allows us to move the creation of the base architectures to the application domain phase. This allows us to define the base architectures by using a set of rules that encode patterns of good design practices (as well as other design decisions), in a generic way. This avoids the need to define all of them explicitly. As the LPS grows in size, BOM–Lazy becomes an adequate approach to manage variability, as it supposes a great work to build *a priori* the base architectures for all the possible products of the SPL. Thus, the main effort is done in the domain engineering stage, where the acquired knowledge is formalized and encoded in a set of declarative rules (the knowledge is stored explicitly). So, it is not necessary to develop extensively all the possible combinations of base architectures (the knowledge is stored implicitly). That will increase the efficiency on the application engineering phase, where each base architecture is obtained only when it is needed by using the explicitly stored knowledge.

VII. Acknowledgments

This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation MULTIPLE TIN2009-13838 and the FPU fellowship program, ref. AP2006-00690.

References

[1] O. Avila-García, A. Estévez, and J. L. Roda. Integrando modelos de procesos y activos reutilizables en una herramienta mda. In J. C. R. Santos and P. Botella, editors, *JISBD*, pages 483–488, 2006.

[2] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Software Product-Family Engineering*, pages 66–80. 2004.

[3] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December, 2006.

[4] J. P. Benedí. *Prisma: aspect-oriented software architectures*. PhD thesis, May 2008.

[5] M. E. Cabello and I. Ramos. Expert systems development through software product lines techniques. In *Information Systems Development*, pages 299–307. Springer US, 2009.

[6] M. E. Cabello Espinosa. *Baseline-Oriented Modeling: An MDA approach based of Software Product Lines for applications development.* PhD thesis, Dec. 2008. http://hdl.handle.net/10251/3793.

[7] P. Clements and L. Northrop. *Software product lines: practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] P. Clements, L. Northrop, and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.

[9] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice*, page 2005, 2005.

[10] K. Czarnecki and C. H. Kim. Cardinality-based feature modeling and constraints: A progress report, October 2005.

[11] Eclipse Organization. The Graphical Modeling Framework, 2006. http://www.eclipse.org/gmf/.

[12] EMF. http://download.eclipse.org/tools/emf/scripts/home.php.

[13] J. C. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1989.

[14] A. Gómez and I. Ramos. Cardinality-based feature models and model-driven engineering: fitting them together. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, volume 37, pages 61–68. ICB-research report, 2010.

[15] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[16] I. Kurtev, J. Bezivin, , and M. Aksit. Technical spaces: An initial appraisal. *In Tenth International Conference on Cooperative Information Systems (CoopIS), Federated Conferences Industrial Track, California.*, 2002.

[17] Object Management Group. Software Process Engineering Metamodel (SPEM), January.

[18] Object Management Group. MDA Guide Version 1.0.1. 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[19] Object Management Group. *MOF QVT Final Adopted Specification*. Object Modeling Group, June 2005.

[20] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[21] J. Pérez, N. Ali, J. A. Carsí, I. Ramos, B. Álvarez, P. Sanchez, and J. A. Pastor. Integrating aspects in software architectures: Prisma applied to robotic tele-operated systems. *Information and Software Technology*, 50(9-10):969 – 990, 2008.

[22] S. Trujillo. *Feature Oriented Model Driven Product Lines*. PhD thesis, School of Computer Sciences, University of the Basque Country, March 2007.

Automatic Tool Support for Cardinality-based Feature Modeling with Model Constraints for Information Systems Development

Abel Gómez and Isidro Ramos

Abstract Feature Modeling is a technique that uses diagrams to characterize the variability of software product lines. The arrival of metamodeling frameworks in the Model-Driven Engineering field (MDE) has provided the necessary background to exploit these diagrams (called feature models) in information systems development processes. However, these frameworks have some limitations when they must deal with software artifacts at several abstraction layers. This paper presents a prototype that allows the developers to define cardinality-based feature models with complex model constraints. The prototype uses model transformations to build Domain Variability Models (DVM) that can be instantiated. This proposal permits us to take advantage of existing tools to validate model instances and finally to automatically generate code. Moreover, DVMs can play a key role in complex MDE processes automating the use of feature models in software product lines.

1 Introduction

The changing nature of technology and user requirements leads us to need multiple versions of the same or similar software application in short time periods. The Software Product Line (SPL) [5] concept arises with the aim of controlling and minimizing the high costs of developing a family of software products in the previous context. This approach is based on the creation of a design that can be shared among all the members of a family of programs within an application domain. The key aspect of Software Product Lines (SPL) that characterizes this approach with respect

Abel Gómez
Department of Information Systems and Computation, Universidad Politécnica de Valencia, Cno. de Vera s/n. Valencia, Spain. e-mail: agomez@dsic.upv.es

Isidro Ramos
Department of Information Systems and Computation, Universidad Politécnica de Valencia, Cno. de Vera s/n. Valencia, Spain. e-mail: iramos@dsic.upv.es

Gómez, A., Ramos, I. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. Proc. of the 19th International Conference on Information Systems Development (ISD'10) 2010.

to other software reuse techniques is how to describe and manage variability, mainly by means of feature modeling as proposed in [12].

The use of feature models can be exploited by means of metamodeling standards. In this context, the Model-Driven Architecture [14] proposed by the Object Management Group is a widely used standard which arises as a suitable framework for this purpose. In this environment, MDE and the Generative Programming approach [6] provides a suitable basis to support the development of SPLs. Moreover, Generative Programming and SPLs facilitate the development of software products for different platforms and technologies.

In this paper we present a prototype that uses nowadays metamodeling tools to define cardinality-based feature models and their configurations. The selected platform is the Eclipse Modeling Framework (EMF) [9]. Moreover, feature models can be enriched with complex model constraints that can be automatically checked by means of the internal OCL interpreter. All these features of EMF allows developers to start a Software Product Line.

The remainder of this paper is structured as follows: in section 2 we briefly introduce feature modeling and in 3 we describe how nowadays standards and tools can be used to exploit them. In section 4 our proposal is presented. Related works are discussed in section 5 and in section 6 we present our conclusions.

2 Feature models and cardinality-based feature models at a glance

Feature models are diagrams which express the commonalities and varibilities among the products of a SPL. These models organize the so-called features (*user-visible aspect or characteristic of the domain*) in a hierarchical structure. The basic relationships between a feature and its children are: *mandatory* relationships (which represent de shared design), *optional* relationships, *OR* groups and *XOR* groups.

Cardinality-based feature modeling [7] integrates several of the different extensions that have been proposed to the original FODA notation [12]. A cardinality based feature model is also a hierarchy of features, but the main difference with the original FODA proposal is that each feature has associated a *feature cardinality* which specifies how many clones of the feature are allowed in a specific configuration. Cloning features is useful in order to define multiple copies of a part of the system that can be differently configured. Moreover, features can be organized in *feature groups*, which also have a *group cardinality*. This cardinality restricts the minimun and the maximun number of group members that can be selected. Finally, an *attribute type* can be specified for a given feature. Thus, a primitive value for this feature can be defined during configuration which is useful to define *parameterized features*.

In feature models is also quite common to describe constraints between features such as the *implies* and the *excludes* relationships, which are the most common used ones. In classic feature models the semantics of these constraints can be expressed

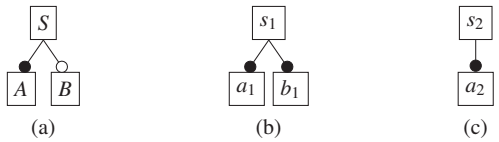


Fig. 1 Example of a feature model (1a) and the two possible configurations that it represents (1b and 1c).

by means of propositional formulas [2], thus, it is possible to reason about the satisfiability of the feature model and its configurations.

A configuration of a feature model can be defined as *a valid set of instances of a feature model*. I.e., the relationship between a feature model and a configuration is comparable to the relationship between a class and an object. In Fig. 1a an example feature model is represented. This feature model represents a system *S*, with two features *A* and *B*. The first one, feature *A*, is mandatory (it must be included in every possible product of the product line), and the second one, feature *B*, is optional (it can be included in a particular product or not). Thus, we have two possible configurations for this feature model, which are represented in figures 1b and 1c.

3 Feature Modeling and Model–Driven Engineering

Model-Driven Engineering (MDE) is a Software Engineering field that over the years has represented software artifacts as models in order to increase productivity, quality and to reduce costs in the software development process. Nowadays, there is increasing interest in this field, as demonstrated by the OMG guidelines that support this trend with the Model-Driven Architecture (MDA [14]) approach.The Meta Object Facility standard (MOF, [16]), which provides support for meta-modeling,

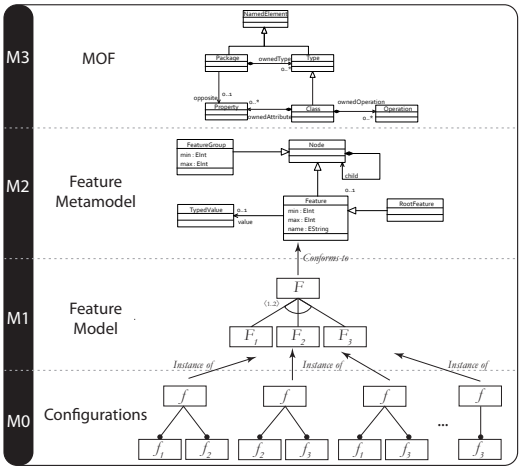


Fig. 2 Definition and configuration of feature models in the context of MOF. The EMOF language is represented in a simplified way in the level M3. In the level M2 the metamodel for cardinality-based feature models is represented by using the MOF language (also in a simplified way). In the level M1 feature models are described. Some configurations of the example feature model are shown at level M0.

defines a strict classification of software artifacts in a four-layer architecture (from M3 to M0 layer). As it provides support for modeling and metamodeling, we can use MOF to define cardinality-based feature models by defining its metamodel. Thus, we can define a model which captures the whole variability of the domain, which, in turn, allows us to define any possible configuration of the model. But, what is more, it can also be used to define model-based transformations that enable the use of feature models and their configurations in other complex processes. Fig. 2 shows where feature models and configurations fit in the four-layer MOF architecture.

3.1 MDA in practice: industrial tool support

The Eclipse Modeling Framework (EMF) [9] can be considered as an implementation of the MOF architecture. Ecore, its metamodeling language, can be placed at layer M3 in the four-layer architecture of the MOF standard. By means of Ecore, developers can define their own models which will be placed at the metamodel layer (M2). An example of such metamodels is the one to build cardinality-based feature models. Finally, these Ecore models can be used to automatically generate graphical editors which are capable of building *instance models*, which will be placed at M1 layer. In the case of feature modeling, these *instance models* are the feature models. The left column on Fig. 3 shows this architecture.

As can also be seen in Fig. 3, the M0 layer is empty. This is a limitation of most of the modeling frameworks which are available today. As said, EMF provides a modeling language to define new models and their instances, but this framework only covers two layers of the MOF architecture: the metamodel and the model layers. However, in the case of feature modeling we need to work with three layers of the MOF architecture: metamodel (cardinality-based feature metamodel), model (cardinality-based feature models), and instances (configurations).

Fig. 3 shows how to overcome this drawback: it is possible to define a model-to-model transformation in order to convert a feature model (i.e. the model represented by Feature model which can not be instantiated) to an Ecore model (i.e. the Domain Variability Model, DVM, which represents the Feature model as a new class dia-

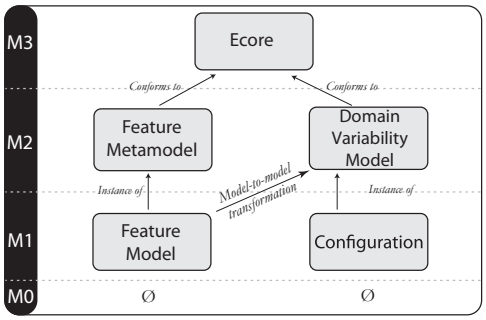


Fig. 3 EMF and the four-layer architecture of MOF. The Ecore language is placed at the M3 layer, the Ecore models (such as the metamodel for cardinality-based features models) are placed at the M2 layer, and model instances (feature models) are placed at the M1 layer. It is noteworthy that EMF can not represent more than 3 layers.

gram). Thus, it is possible to represent a feature model at the metamodeling layer, making the definition of its instances possible. This way, developers can take advantage of EMF again, and automatically generate editors to define feature model configurations, and validate them against their corresponding feature models thanks to their new representation, the DVM. Moreover, as the DVM is an Ecore model (a simplified UML class diagram) we automatically obtain support to check complex constraints (by using OCL) over the feature model configurations.

4 Our approach

Based on the concepts presented in the previous section and using EMF, we have developed a tool that allows us to automate several steps in order to prepare a feature model that can be exploited to develop a SPL in the context of MDA. In this sense, our tool provides:

- Graphical support to define (a variant of) cardinality-based feature models with model constraints expressed by using a constraint language.
- Support to automatically generate DVMs from feature models that capture all the variability of the application domain (including complex model constraints), allowing the developers to use them in model transformations.
- Support to transform model constraints to OCL expressions.
- Configuration editors, which will assist the developers.
- Capabilities to check the consistency of a configuration against its corresponding feature model by using pre-built OCL engines.

All these tasks are automatically supported by using MDE techniques (modeling, metamodeling, model transformations and code generation). The following subsections describe how this process has been implemented.

4.1 Cardinality-based feature metamodel

The basis of our work is the cardinality-based feature metamodel, which permits to define feature models. Fig. 4 shows our feature metamodel. Such metamodel has been defined taking into account that every element will have a different graphical representation. This way, it is possible to automatically generate the graphical editor to draw feature models based on such metamodel.

4.1.1 Feature models structure

In Fig. 4, a feature model is represented by means of the FeatureModel class, and a feature model can be seen as a set of Features, the set of Relationships among

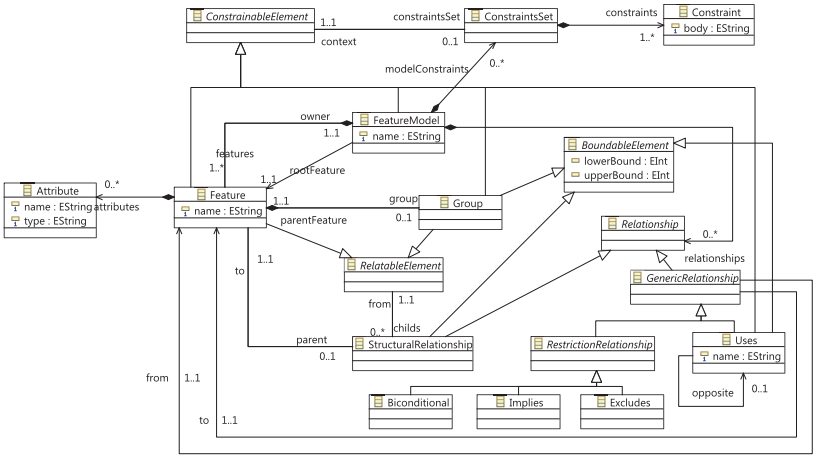


Fig. 4 Cardinality-based features metamodel

them and the set of model constraints (modelConstraints role) that are applied to it. A feature model must also have a root feature, which is denoted by means of the rootFeature role. As can be seen in Fig 4, our proposal represents explicitly the relationships between features. Thus, it represents in an uniform way the hierarchical relationships (StructuralRelationship class) and the restrictions between features (RestrictionRelationship class). The classification of these relationships is explained in detail in [10].

4.1.2 Feature model constraints

As was pointed out in section 2, it is quite common in feature modeling to have the possibility to define model constraints in order to describe more precisely which configurations should be considered as valid. Typically, these constraints are described by means of implication or exclusion relationships. This kind of relationships are the *binary and horizontal relationships* that our metamodel provides.

The *binary and horizontal relationships* are specified between two features and they can express constraints (coimplications, implications and exclusion) or dependencies (use). The first group applies to the whole set of instances of the involved features, however, the second one allows us to define dependencies at instance level, i.e.:

- *Implication ($A \longrightarrow B$)*: If an instance of feature *A* exists, at least an instance of feature *B* must exist too.
- *Coimplication ($A \longleftrightarrow B$)*: If an instance of feature *A* exists, at least an instance of feature *B* must exist too and vice versa.
- *Exclusion ($A \nrightarrow\!\!\!\!\!\!\nleftarrow B$)*: If an instance of feature *A* exists, can not exist any instance of feature *B* and vice versa.

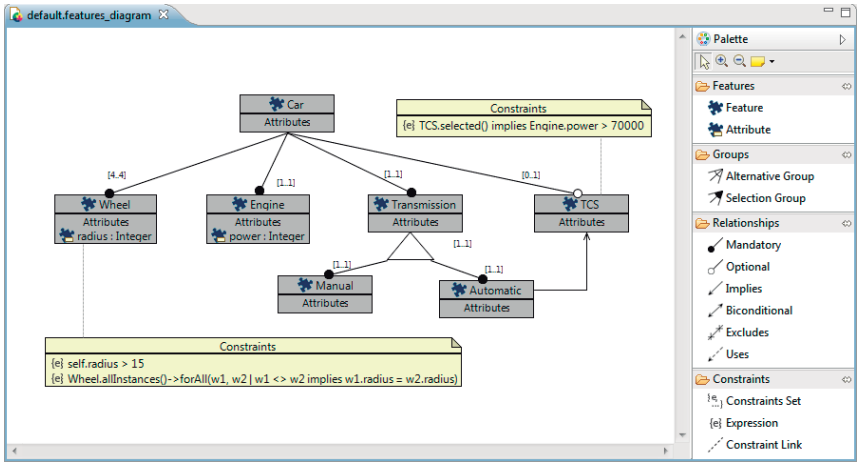


Fig. 5 Example feature model

- *Use ($A - - \rightarrow B$)*: This relationship will be defined at configuration level, and it will specify that an specific instance of feature A will be related to one (or more) specific instances of feature B as defined by its upper bound (n).

Besides these kind of relationships that describe coarse-grained restrictions, our metamodel provides capabilities to describe fine-grained restrictions. These restrictions are stored in a Constraints Set instance, and can be applied to any subclass of the abstract class ConstrainableElement (context role), i.e., FeatureModel, Feature, Group or Uses. The restrictions are expressed as a textual expression (body attribute of the Constraint class).

To describe these fine-grained restrictions we propose a constraint language, called *Feature Modeling Constraint Language* (FMCL). FMCL is a formal language without side-effects (does not modify the model instances) whose syntax is based on the widely known *Object Constraint Language* (OCL) and its semantics are defined by a set of patterns that describe the equivalences between FMCL expressions and OCL expressions.

4.1.3 Cardinality-based feature modeling editor

Following the Model-Driven Software Development (MDSD) approach, graphical editors can be automatically generated from the metamodel presented: i.e., the cardinality-based feature modeling editor. This editor allows us to easily define new feature models. To obtain this graphical editor, the Graphical Modeling Framework (GMF [8]) has been used.

Fig. 5 shows what this editor looks like. The palette is located on the right side of the figure, and shows the tools that can be used to define the feature models. In the

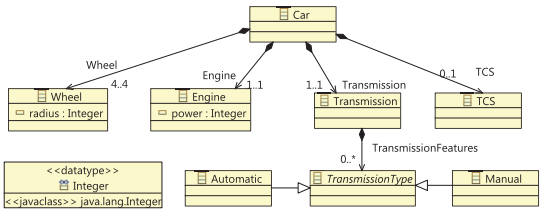


Fig. 6 Generated DVM for the example feature model. In the figure, model constraints (OCL annotations) have been omitted for clarity purpouses.

canvas an example feature model is shown. This feature model describes a simple product line for cars. A car must have four wheels (of a given radius), one engine (of an specific power in watts) and a transmission (which can be manual or automatic). As an optional equipment the car can have a Traction Control System (TCS). The feature model also describes four constraints: the arrow between the feature TCS and Automatic states that if an automatic transmission is selected, the TCS must be selected too; the annotation attached to the TCS feature states that the TCS can only be selected if the power of the engine is higher than 70.000 watts; and finally, the annotation attached to the Wheel feature specifies that the radius of the instances of the wheel must be higher than 15 inches and that all the wheels must be of the same size.

4.2 The Domain Variability Model

The Domain Varibility Model is a class diagram (an Ecore model) whose instances are equivalent to the configurations of a feature model. It is intended to ease the definition of feature models configurations in EMF as was explained in section 3.1. This model can be automatically generated by means of a model-to-model transformation. Following the MDA guidelines, this transformation is defined by using the Relations language defined in the QVT standard [15]. In order to integrate and execute this transformation process in our prototype, a custom tool based on the mediniQVT [11] transformations engine has been built.

4.2.1 The structure of the DVM

As can be observed in Figs. 5 and 6, the transformation regarding to the structure of the DVM is almost a one-to-one mapping. For each Feature of the source model an *EClass* (with the same name) is created. All the classes are created inside the same EPackage, whose name and identifier derives from the feature model name. Moreover, for each feature Attribute, an EAttribute in its corresponding EClass is created in the target model. Any needed EDataType is also created.

Regarding to the relationships, for each StructuralRelationship from a parent Feature, a *containment* EReference will be created from the corresponding EClass

and for each Group contained in a Feature a *containment* EReference will be created from the corresponding EClass. This EReference will point to a new abstract class, whose name will be composed by the Feature name and the suffix "Type". Additionally, an EClass will be generated for each Feature belonging to a Group. Moreover, each one of these EClasses inherit from the abstract *EClass* that has been previously created. Finally, for each Uses relationship between two Features, an EReference will be created in the target model. This EReference will relate two EClasses whose names will match the Features names.

4.2.2 Constraints over the DVM

The *restriction relationships* and *model constraints* (FMCL expressions) are mapped to OCL expressions in the DVM. The mappings to transform the *restriction relationships* to OCL expressions is described in [10] in detail.

The FMCL expressions are mapped to OCL expressions taking into account the mappings explained in section 4.2.1. Fig. 5 shows an example of this. As can be seen on the constraint that applies to the Wheel feature, a FMCL expression can be expressed directly using the OCL syntax. This way, an FMCL expression is directly transformed to an OCL invariant. The context of the invariant corresponds to the name of the *ConstrainableElement* that is linked to the constraint (dashed line in the figure) and the text of the expression remains the same:

```
context Wheel
inv: self.radius > 15
inv: Wheel.allInstances()->forAll(w1, w2 |
                              w1 <> w2 implies w1.radius = w2.radius)
```

However, although the FMCL expressions are almost the same than an OCL invariant, some simple conventions have been adopted to make the definition of model constraints closer to the feature modeling context. The semantics of these additions are defined by means of transformation patterns (see Table 1).

An example of the application of some of these patterns can be seen in the constraint attached to the TCS feature (Fig. 5). The example constraint is transformed to the following OCL expression:

```
context TCS inv:
TCS.allInstances()->notEmpty() implies Engine.allInstances()->forAll(power >
     70000)
```

Table 1 Summary of transformation patterns (FMCL to OCL)

FMCL expression pattern	Equivalent OCL definition
ConstrainableElement	*ConstrainableElement*`.allInstances()`
ConstrainableElement`.`*property op expression*	*ConstrainableElement*`.allInstances()->forAll(`*property op expression*`)`
ConstrainableElement`.selected()`	*ConstrainableElement*`.allInstances()->notEmpty()`
FeatureName`.childs()`	*FeatureName*`Type.allInstances()`

4.3 Creating and validating configurations

In order to create new configurations of feature models it is not necessary to use any custom tool. As far as we have a DVM which captures the same variability than the original feature model, developers can use the standard Ecore tools. However, in EMF there is not a default tool to check OCL invariants which are directly stored as EAnnotations in Ecore models themselves. Thus, we have built an extension which can take advantage of the OCL invariants that have been automatically created in the previous transformation step.

Fig. 7 shows an example configuration. It shows a car configuration with manual transmission, TCS, 4 wheels and engine. The radius of three of the wheels is 16 inches, and the radius of the fourth is 17 inches. The power of the engine is 60.000 watts. This configuration is invalid conforming to the restrictions applied to the metamodel. When the configuration is invalid the checking process is unsuccessful. In this situation, the prototype console shows a summary with the constraints that are not met, and which are the problematic elements as the figure shows.

5 Related works

Feature modeling has been an important discussion topic in the SPL community, and a great amount of proposals for variability management have arisen. Most of them are based in the original FODA notation and propose several extensions to it [4]. Our work is closely related with previous research, however, there are several distinctive aspects:

Our work describes a prototype to define and validate configurations of feature models. Previous work has been also done in this area, such as the *Feature Modeling Plugin* [1]. The main difference with our work is that configurations are defined in terms of the feature metamodel and both models and configurations coexist at the same layer. Thus, in order to be able to deal both with models and configurations it is necessary to build complex editors (as they must guarantee that the specialization

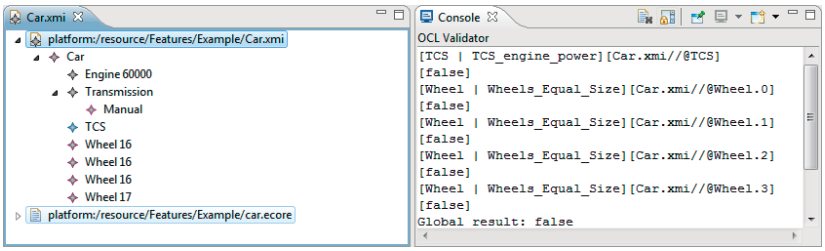


Fig. 7 Example of an unsuccessful configuration check.

process is properly done), and whats more, those artifacts can not be easily used in complex MDE processes.

Some previous works have already represented feature models as class diagrams ([7] and [13]). However, in [7] no set of transformation rules defines the mappings between features and classes. In this work, OCL is also presented as a suitable approach to define model constraints, but there is no automatic generation of OCL invariants as the transformation is not clearly defined. In turn, [13] presents a set of QVT rules to define the mappings. However, in this case, neither model constraints nor configuration definitions support is presented.

In [2] a proposal for feature constraints definition and checking is done, representing features as propositions and restrictions among them as propositional formulas. However, this approach is not suitable when feature can have typed attributes which can not be expressed by this kind of formulas. We state that more expressive languages are needed, such as FMCL/OCL.

6 Conclusions

In this paper we have presented a framework[1] to define and use feature models in a MDE process. This framework addresses two issues: first, the incapability of nowadays metamodeling tools to deal simultaneously with artifacts located in all the MOF layers; and second, the complexity to define model constraints in feature models where features can be cloned and can have attributes. These problems have been solved by transforming feature models to Domain Variability Models that can be instantiated and reused in future steps of the MDE process.

Our tool has been designed following the MDE principles and a metamodel for cardinality-based feature modeling has been defined. By means of generative programming techniques, a graphical editor for feature models has been built. Feature models defined with this editor are automatically transformed in DVMs that are used to define configurations of feature models. Although several tools to define feature models and configurations in the last years have arised, our approach has several advantages against previous approaches: (i) the infrastructure that we propose to build configurations is simpler and more maintainable, as it is built following the MDSD guides; (ii) configurations are actually instances of a feature model (expressed by means of the DVM), so we can take advantage of the standard EMF tools; (iii) as feature models are described by DVMs that can be instantiated, both models and configurations can be used in other MDE tasks; (iv) having a clear separation between feature models and configuration eases the validation tasks as they can be performed by means of built-in languages; and (v) as the transformation between feature models and DVMs is performed automatically by means of a declarative language we can trace errors back from DVMs to feature models.

[1] This framework is supported by a prototype that can be downloaded from `http://issi.dsic.upv.es/~agomez/feature-modeling`.

It is noteworthy to remark the importance of using feature models and configurations at different layers. In [3] an example where this architecture is used to integrate feature models in a MDE process is shown. This work describes how a model transformation with multiple inputs (feature models and functional models) is used to generate a software architecture automatically.

Acknowledgements This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation MULTIPLE TIN2009-13838 and the FPU fellowship program, ref. AP2006-00690.

References

1. M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. *2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, 2004.
2. D. Batory. Feature models, grammars, and propositional formulas. pages 7–20. Springer, 2005.
3. M. E. Cabello, I. Ramos, A. Gómez, and R. Limón. Baseline-oriented modeling: An mda approach based on software product lines for the expert systems development. *Intelligent Information and Database Systems, Asian Conference on*, 0:208–213, 2009.
4. L. Chen, M. A. Babar, and N. Ali. Variability management in software product lines: A systematic review. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*, 2009.
5. P. Clements, L. Northrop, and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.
6. K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
7. K. Czarnecki and C. H. Kim. Cardinality-based feature modeling and constraints: A progress report, October 2005.
8. Eclipse Organization. The Graphical Modeling Framework, 2006. `http://www.eclipse.org/gmf/`.
9. EMF. `http://download.eclipse.org/tools/emf/scripts/home.php`.
10. A. Gómez and I. Ramos. Cardinality-based feature modeling and model-driven engineering: Fitting them together. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'10)*, Linz, Austria, Jan. 2010.
11. ikv++ technologies AG. ikv++ mediniQVT website. `http://projects.ikv.de/qvt`.
12. K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.
13. M. A. Laguna, B. González-Baixauli, and J. M. Marqués Corral. Feature patterns and multiparadigm variability models. Technical Report 2008/01, Grupo GIRO, Departamento de Informática, may 2008.
14. Object Management Group. MDA Guide Version 1.0.1. 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.
15. Object Management Group. *MOF QVT Final Adopted Specification*, June 2005.
16. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

cation is expected to have an impact on several Product Architectures (PAs), namely each PA that contains the modified component. For instance, when B evolves into B', the assumption and guarantee pairs of both B and B' must be checked.

Our solution thus envisages a combination of regression testing and assume-guarantee testing applied to evolving PLAs. Conceptually, this requires us to better understand two issues.

First, how to extend assume-guarantee testing to evolving architectures. The assume-guarantee pair associated with each component in an architecture is normally used to generate component-specific testing traces which, once evaluated against the appropriate assume-guarantee premise, can show whether the composition of components can produce failures. Assuming a PLA/PA has been tested, the challenge becomes how to apply assume-guarantee reasoning for regression testing the modified PLA/PA (see (1) and (2) in Figure 2, respectively). Second, how to use the relationships between a PLA and its PAs (see (3) in Figure 2) to apply regression testing to the evolved PAs.

In summary, we currently envisage what we call a double regression testing approach, in which an evolved product (eg PA3' in Figure 2) can be tested based on how it regressed from PA3, and based on its relationship with the architecture of its family, PLA' (cf. Figure 2). Our expectation is that this considerably reduces the effort needed to re-analyze products of an SPL upon its evolution.

We are working out the details of our approach, after which we intend to implement it and investigate its effectiveness by applying it to case studies.

Please contact:
Maurice ter Beek
ISTI-CNR, Italy
E-mail: maurice.terbeek@isti.cnr.it

Software Evolution in Model-Driven Product Line Engineering

by Silvia Abrahão, Javier González-Huerta, Emilio Insfran and Isidro Ramos

New requirements and technology changes lead to continuous changes of the assets comprising a software product line. Since the product line represents a large number of potential products (or already deployed products) in a given domain, managing these changes becomes a key issue when dealing with evolution. We present a framework to support the development and evolution of high-quality software product lines. The framework is based on several interrelated models or system views (eg, functionality, variability, quality) and a production plan defined by model transformations that generate a software system that meets both functional and quality requirements. We used our framework to develop a software system for the automotive domain.

Software Product Lines (SPLs) are families of products that share common functionality but also have variations tailored for different customer needs. Many of the benefits expected from SPLs are based on the assumption that the additional investment in setting up a product line pays off later when products are created. However, product line assets have to evolve continuously in order to keep the economic benefits of a product line at an optimal level. Managing this evolution therefore becomes a key issue when maintaining a product line.

In the MULTIPLE project, we defined a framework to support the development and evolution of high-quality SPLs. This framework is based on the definition of a multimodel that represents the different system views and the relationships among them. The variability management involves the manipulation of features, represented as cardinality-based feature models, and the support of such variability in the so-called product line core assets. Specifications of the system variability, functionality, and quality can be dealt with models that are independent from each other but where their inter-consistency is assured by means of the relationships defined in this multimodel.

The SPL production plan is parameterized by means of the multimodel which specifies the corresponding model transformations that are needed to obtain a specific product with the desired features, functions, and quality.

Figure 1 shows the multimodel playing a pivotal role in the SPL production plan. In the domain engineering phase it is used to express the impact and constraints among features, functional components and quality attributes, describing in this way, the extension of the product line. In the application engineering phase, it guides the product configuration, allowing the selection of features and functional components that meets the quality attributes selected by the application engineer. This approach leads us to re-consider the problems related to the intra (eg internal consistency of the feature model) and inter-model consistency (eg correspondences among elements of the feature and quality models) in a broader and realistic context.

Model-based evolution of software systems implies the evolution by using models, eg applying model-driven techniques to support product evolution, or the evolution of models, ie the evolution of the models/metamodels that describe the product. In our framework these evolutions are done by means of reification of software artifacts at the metalevel. When the evolution affects the types then the metamodels need to be reified and evolved. A reflection process translates to the source level the evolved artifact. At a model level, the process is similar. The models are reified at the metalevel, evolved, and then

Abrahao, S., Javier González, Insfrán, E.,Ramos, I. Software Evolution in Model-Driven Product Line Engineering. ERCIM News 88, Special Theme: Evolving Software. 2012.

reflected. In the MOF hierarchy, the evolution behaves in a similar way in each level and the consequences over the instances of the evolved artifact can be computed automatically.

Figure 2 shows the main steps of a production plan to develop a software system by applying model transformations for the automotive domain. The quality attributes that the product must fulfill are used as the decision factor when alternative transformations appear. In this way, the application engineer can experiment with different quality-driven model transformations (which represent design alternatives) and choose the one that better satisfies the functional and quality requirements.

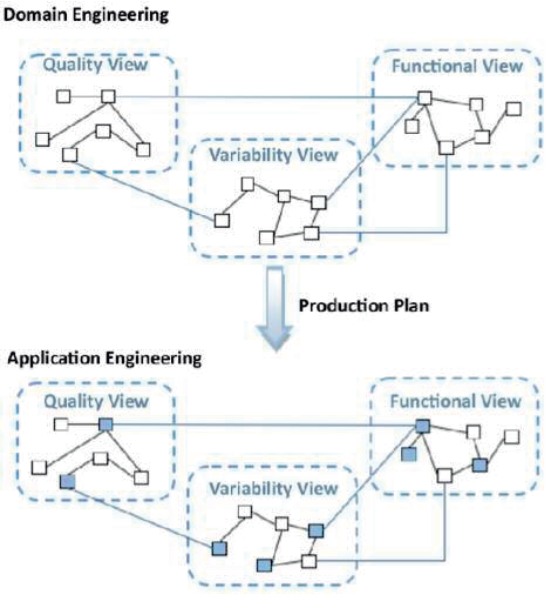


Figure 1: The Multimodel in the Software Product Line development process

The output of the process is the product represented by means of its architectural models. These architectural models can be completed in successive model transformation processes to generate the fully functional version of the product. In addition, in each step of the process, the quality model can be used to determine whether the artifacts obtained (eg product architecture) meet the expected quality levels by automatically applying metrics defined on the quality model. This assessment provides evidence about the correctness of the individual models as well as the multimodel relationships.

Finally, the multimodel helps to establish the traceability and to ensure the consistency among the system views. Furthermore, the model transformations are guided by the structural constraints established by the multimodel and the quality view contributes to the quality assurance and improvement of the product line artifacts. Currently, we are building a tool to give support to this approach.

Links:
ISSI Research Group (Universitat Politècnica de València)
http://issi.dsic.upv.es/projects

FMCL: Feature Modeling Constraint Language
http://issi.dsic.upv.es/~agomez/feature-modeling

Please contact:
Silvia Abrahão
Universitat Politècnica de València, Spain
Tel: + 34 963877000
E-mail: sabrahao@dsic.upv.es

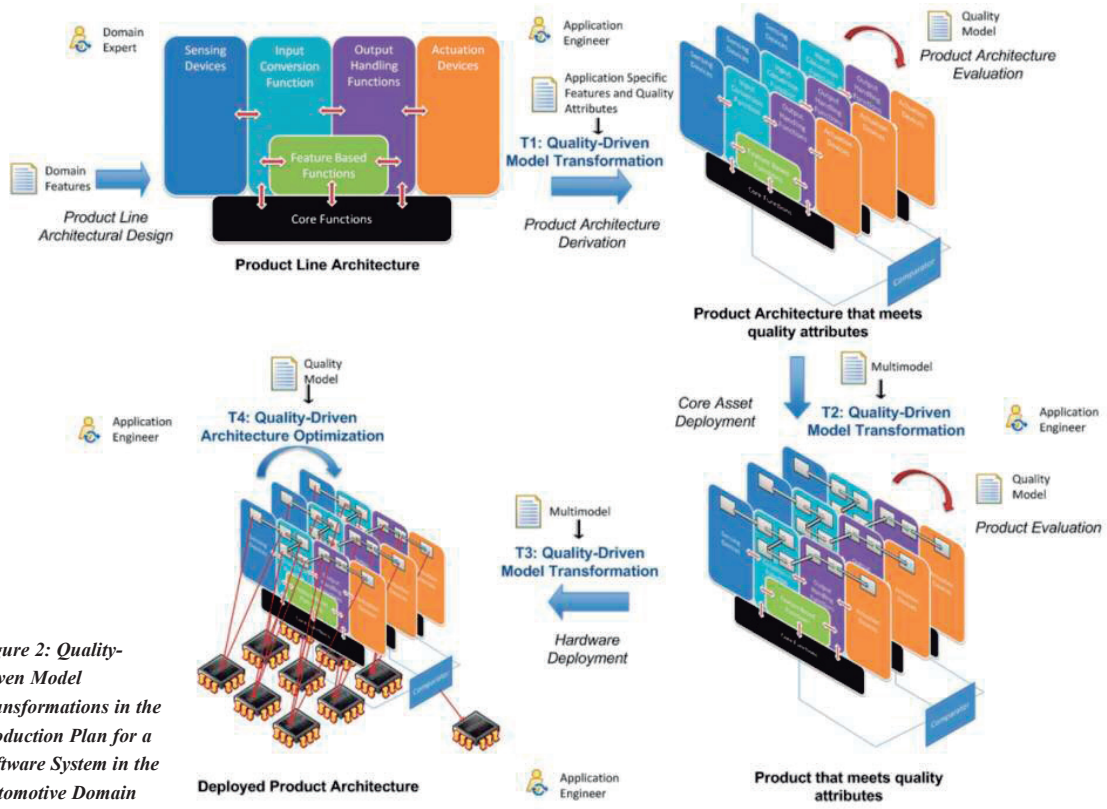


Figure 2: Quality-driven Model Transformations in the Production Plan for a Software System in the Automotive Domain

Int. J. Intelligent Information and Database Systems, Vol. 6, No. 2, 2012 129

SPL variability management, cardinality and types: an MDA approach

María Eugenia Cabello Espinosa*

Faculty of Telematics, Universidad de Colima,
Av. Universidad 333, C.P. 28040 Colima, Colima, México
E-mail: ecabello@ucol.mx
*Corresponding author

Isidro Ramos Salavert

Department of Systems Engineering and Computation,
Universitat Politècnica de València,
Camino de Vera s/n, 46020 Valencia, Spain
E-mail: iramos@dsic.upv.es

Jorge Rafael Gutiérrez Pulido

Faculty of Telematics, Universidad de Colima,
Av. Universidad 333, C.P. 28040 Colima, Colima, México
E-mail: jrgp@ucol.mx

Abel Gómez Llana

Department of Systems Engineering and Computation,
Universitat Politècnica de València,
Camino de Vera s/n, 46020 Valencia, Spain
E-mail: agomez@dsic.upv.es

Rogelio Limón Cordero

Department of Systems Engineering and Computation,
Universitat Politècnica de València,
Camino de Vera s/n, 46020 Valencia, Spain
E-mail: limonroge@hotmail.com

Abstract: This paper presents a baseline-oriented modelling (BOM) approach to develop families of software products. BOM is a generic solution implemented as a framework that automatically generates software applications using executable architectural models by means of software product line (SPL) techniques. In order to cope with the variability problem, BOM considers its cardinality and type and implements two solutions: the BOM-EAGER and the BOM-LAZY approaches. BOM has been designed following the model-driven architecture (MDA) standard: all the SPL software artefacts are models, and model transformations enact the SPL production plan.

Copyright © 2012 Inderscience Enterprises Ltd.

Cabello, M. E., Ramos, I., Gutiérrez, J. R., Gómez, A., Limón, R. SPL Variability Management, Cardinality and Types: an MDA Approach. International Journal of Intelligent Information and Database Systems (IJIIDS), Vol. 6, No. 2. 2012.

Keywords: intelligent information; metamodels; models; model-driven architectures; model transformations; software product lines; software product line production plan; variability management.

Reference to this paper should be made as follows: Cabello Espinosa, M.E., Ramos Salavert, I., Gutiérrez Pulido, J.R., Gómez Llana, A. and Limón Cordero, R. (2012) 'SPL variability management, cardinality and types: an MDA approach', *Int. J. Intelligent Information and Database Systems*, Vol. 6, No. 2, pp.129–153.

Biographical notes: María Eugenia Cabello Espinosa received her Bachelor in Physics from the Universidad Nacional Autónoma de México (México), Master in Computer Science from the Fundación Arturo Rosenblueth (México), and both Master in Software Engineering, Formal Methods and Information Systems and PhD in Declarative Programming and Software Engineering from the Universitat Politècnica de València (Spain). Her research interests include: expert systems, diagnosis domain, software architectures, software product lines, MDE, model transformations, and ontologies.

Isidro Ramos Salavert received his Bachelor and PhD in Physics from the Universidad Complutense de Madrid (Spain) and Bachelor in Computer Science from the Universidad Politécnica de Madrid (Spain). He is currently working in the fields of MDE, software architectures, and software product lines.

Jorge Rafael Gutiérrez Pulido received his Bachelor in Informatics Engineering and Master in Telematics from the Universidad de Colima (México), and PhD in Computer Science from Nottingham University (UK). His main research interests include: ontologies, knowledge-based systems, Semantic Web, self-organising maps, intelligent data mining, artificial neuronal network.

Abel Gómez Llana received his Bachelor in Computer Science and Master in Software Engineering, Formal Methods, and Information Systems from Universitat Politècnica de València (Spain). He is currently working on his PhD in Declarative Programming and Software Engineering at the Universitat Politècnica de València (Spain). His research interests are MDE, model transformations, software product lines, variability management and their applications in different contexts such as bioinformatics and feature models.

Rogelio Limón Cordero received his Bachelor in Industrial-Electric Engineering from the Instituto Tecnológico de Oaxaca (México), Master in Computer Science from the Centro Nacional de Investigación y de Desarrollo Tecnológico (México), and PhD in Declarative Programming and Software Engineering from the Universitat Politècnica de València (Spain). His research interests include software architecture views, MDE, and model transformations.

1 Introduction

Nowadays, software engineering promotes automation in software development and the use of open and standard mechanisms to deal with the complexity of software systems. However, the automatic generation of such systems is only possible when there is a framework that supports this process. The Object Management Group (OMG) (http://www.omg.org) proposes the model-driven architecture (MDA)

(http://www.omg.org/mda) approach for model-driven engineering (MDE) (Kent, 2003). MDA advocates the use of standards and platform independence in the software development process as a new way of producing applications.

Furthermore, the development of complex systems is becoming more elaborate due to a series of factors. These factors are the emergence of new interaction technologies and the interconnection of different systems and platforms. Also, the need to develop custom software for each type of user complicates the specific aspects of systems in different implementation platforms. This state of affairs requires having multiple versions of the same application in order to deal with this variability.

To cope with this variability problem, software product lines (SPLs) (http://www.softwareproductlines.com) have emerged in an effort to control and minimise the high costs of the software development process and to reduce the time to market of these new products. This approach is based on having a base design that is shared by all the product family members. Thus, a specific product can benefit from the common parts of the software. The base design can be re-used in different products by adding different features that characterise them.

The development of expert systems (ES) (Giarratano and Riley, 2004) has also become increasingly important in recent years creating a need to properly support such applications. Since systems are more and more relevant, the need for techniques for their development has also become more important. Additionally, ES introduce a difference regarding the decision making process: they store expert knowledge in a knowledge base. However, these systems are complex because their architectural elements vary.

In this context, we have developed the baseline-oriented modelling (BOM) framework (Cabello, 2008) which is an MDA approach based on SPL for applications development.

Our work follows the MDA approach in order to automatically generate code from models, by means of transformation rules and SPL techniques, to minimise the variability impact on the cost of the software production. The PRISMA framework (Pérez, 2006) is our selected target platform, and the diagnostic expert system domain is our case study. Therefore, BOM automatically generates diagnostic ES (as PRISMA architectural models) based on SPL by using the generative programming approach of the MDA trend.

However, not everything is automated: user interaction is required to provide information on the domain variability using domain conceptual models (DCMs). In the role of application engineer BOM users, develop a specific ES when they introduce the variability information of a particular domain into the system.

The products of our SPL have been designed as PRISMA models that capture the architecture and functionality of the rule-based ES. The process of creating our SPL uses a set of reusable resources or assets (core assets) to create a family of software products, using two OMG standards: the reusable asset specification (RAS) (http://omg.org/technology/documents/formal/ras.htm) and the software process engineering metamodel (SPEM) (http://omg.org/technology/docs/ad/06-06-02.pdf).

One of the key elements of a SPL is how to represent and manage variability. We use two approaches for variability management: BOM-EAGER and BOM-LAZY for small and large cardinality of the variants set, respectively. To illustrate BOM, we have chosen the diagnostic ES example. The main goal of this kind of system is to capture the state of an entity from a series of data (observation variables) and produce a diagnosis. The domain of ES for diagnosis includes systems for medical diagnosis, educational

diagnosis, and emergency diagnosis, among others. In this paper we present the medical diagnosis as the application domain, using a case study of infantile infectious diseases.

We have built BOM to achieve the following goals:

1 create new (diagnostic) systems in different domains

2 decrease production costs by reusing software packages or assets

3 generate code automatically to increase the productivity and quality of software and to decrease the time to market

4 construct systems in a simpler way by using diagnosis and application domain models closer that are to the problem domain and facilitating user interaction

5 develop platform-independent systems from the problem perspective and not from the solution perspective, which will provide generality in the development approach and applicability in different domains and platforms.

The structure of the paper is as follows: Section 2 presents the different technological spaces that we integrate. Section 3 outlines our field study. Section 4 presents our BOM framework. Section 5 describes the BOM-LAZY and BOM-EAGER approaches. Section 6 presents our prototype. Section 7 describes some related work. Finally, Section 8 presents some conclusions and our future work.

2 Foundations

In order to deal with the complexity of the problem, our work integrates various technological spaces. These are described in turn:

1 the MDA (http://www.omg.org/mda) as an approach to the new MDE trend which is promoted by the OMG for software system development

2 the SPL approach (http://www.softwareproductlines.com), which from a practical point of view is one of the most successful ones since it combines systematic development and reuse of assets; i.e., the products are different in some features but share a basic architecture

3 the PRISMA framework (Pérez, 2006), which defines the architectural elements (components, connectors, and systems) through their aspects

4 the ES (Giarratano and Riley, 2004), which capture the knowledge of experts and try to imitate their reasoning processes when solving problems in a specific domain.

3 Our field study: diagnostic ES

We have carried out a field study to learn about variability in diagnostic ES.

3.1 The diagnosis: domain variability

To understand the diagnostic domain, a field study has been carried out considering the following examples: medical diagnosis, diagnosis of victims in disasters, television diagnosis, educational programme diagnosis, and scholarship candidate diagnosis.

Based on the analysis of the diagnostic process carried out in Cabello (2008), we can conclude that:

- diagnosis consists of an interpretation of the states of the involved entities (viewed as a set of properties), followed by the identification and specification of domain properties using rules.
- there is variability in the diagnostic process (system behaviour) and this variability can be described in terms of its features as follows:
  1 entity views: an entity can be considered to participate with the same properties (the same view) or have different properties (different views) during the diagnostic process
  2 property levels: the properties of the entities can have n different abstraction levels
  3 number of hypotheses: the goal of the diagnosis is a single validated hypothesis, but there can be one or several candidate diagnostic hypotheses that must be evaluated in order to select the valid one
  4 reasoning types: these show the ways in which the rules are applied by the inference engine in order to infer a final diagnosis.

There is also variability in the user requirements. We have elicited and modelled as UML-use case diagrams (http://omg.org/docs/formal/05-07-04.pdf) the user interaction requirements since they impact on the software architecture structure. We can describe this variability in terms of its features as follows:

1 number of use cases of the system and how the system interacts with the environment (final users)

2 number of actors: number of final users of the system

3 use cases per actor: an actor can access different use cases.

However, the features of the application domain must also be considered. Therefore, another variability emerges: the application domain variability. The features that correspond to a specific application domain are:

1 name and type of the entities' properties by abstraction level

2 rules by abstraction levels (the rules describe how the entity properties are related inter-levels)

3 level, name and type of the hypotheses used in the diagnostic process.

3.2 The diagnostic ES: domain functionality

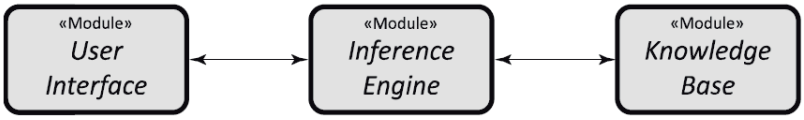
Based on the field study carried out in the diagnostic ES domain, we have detected that a unique *generic architecture* captures the functionality of these systems. This generic architecture has three basic modules (Giarratano and Riley, 2004):

1 a module that executes the inference process that resolves a problem in a specific domain

2 a module that contains the knowledge about the domain

3 a module that allows communication between the user(s) and the system.

Figure 1 Expert system generic architecture



Since we implement the SPL's assets using PRISMA software architectures (Pérez, 2006), i.e., platform-specific models (PSM), the modules of the expert system generic architecture must be mapped into the following architectural elements: the inference engine component(s), which establishes system control and provides the general resolution strategy for taking a decision; the knowledge base component(s), which contains the domain knowledge of the case study using application domain rules (Horn clauses) and facts (constant information); and the user interface component(s), which establish(es) the human-computer interaction. Therefore, there is a correspondence among the modules and their respective components. However, to be consistent with the PRISMA metamodel, it is necessary to incorporate a new architectural element (connector) to establish the communication among components.

Nevertheless, we have detected that the *base architectural model* that implements the generic modular model is not unique; it varies in its structure and behaviour depending on design decisions. This implies the creation of several base architecture models for each generic model in a product line.

We have developed a modelling technique based on architectural patterns to design ES by using PRISMA software architectures. This modelling technique takes into account that an expert system varies both in the structure and in the architectural element types of its architectural model. Furthermore, we have concluded (Cabello, 2008) that the variability in the structure of the expert system is related to the features associated with the user requirements and that variability in the behaviour of the expert system is related to the associated diagnostic process.

4 The BOM framework

BOM is a framework for variability elicitation, specification, and management in the domain engineering phase of a software products line. It is used for the enactment of the software production plan during the product engineering phase of the SPL. In order to illustrate our conceptual framework, we use the diagnostic ES domain.

4.1 Variability management in BOM

Currently, there is no a standard way to represent software variability. Many approaches have proposed adding variability annotations directly to the models that capture the functionality of the system (UML, ADLs), while other more powerful approaches capture variability specification in models. In BOM, the specification of the variabilities and system functionality are modelled in separate conceptual models. The user introduces the instances of these conceptual models in order to define the domain features used to produce the corresponding assets, or introduces to BOM the application domain features used to configure the final application, respectively.

The process of developing a specific application (member of the product line) begins with a domain-dependent generic architecture, which is unique. This implies that domain variability is captured in additional features that are represented as variants of the variability points. This variability is initially reflected in several base architectures (which are derived from the generic architecture). We also considered application domain variability to be necessary, and as a result, we found that there are two types of orthogonal variability: the first one comes from the particular domain (diagnostic ES in our case study), and the second one comes from the application domain (for example, diagnostic medical ES).

For this reason, and based on the field analysis carried out, we conclude that the variants in ES are captured in two types of orthogonal variability:

1 Variants of the domain, for example, the diagnosis (i.e., the features of the domain associated with the behaviour of the architectural elements), and the end-user requirements (i.e., features related to the structure of the architectural elements and the system itself)

2 Variants of the application domain. These variants are the features of the final products.

Figure 2 Features of the first variability (V1) and the second variability (V2) of our SPL

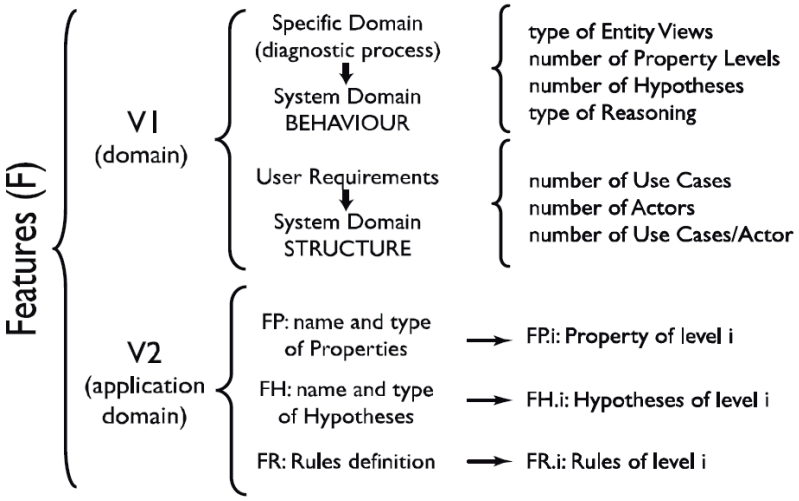


Figure 2 presents a schema with features (variants) of these two variabilities. These two variabilities (domain variability and application variability) are managed in our approach in two stages. These stages correspond to the development of the SPL in two artefact steps: the base architecture SPL1, which shares a generic architecture and the application SPL2 in a specific domain, which shares a base architecture.

BOM captures both variabilities in two feature models (FMs) (Czarneki and Eisenecker, 2000) which are automatically translated to two UML conceptual models: the DCM and the application domain conceptual model (ADCM), respectively.

In this way, we have specified the variability of the systems independently from their functionality using specific FMs which we consider a Domain Specific Language (DSL) suited to the domain.

4.1.1 First variability in the BOM framework

The V1 variability (related to the features of the domain) is represented and managed through two models: The first model corresponds to the functional model of the domain, represented by the generic architecture of our SPL. This architecture is shared by several base architectures (or templates of base architectures), which represent the first SPL: SPL1. The second model is the model of the domain variability, which is represented by the FM and is shown in Figure 3.

Figure 3 The FM of the first variability (see online version for colours)

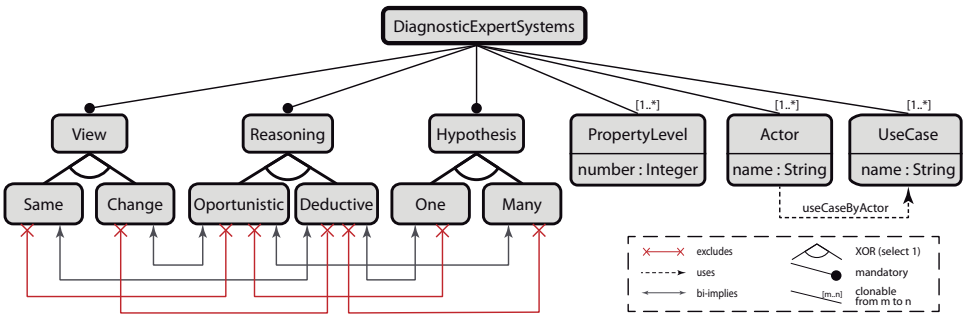
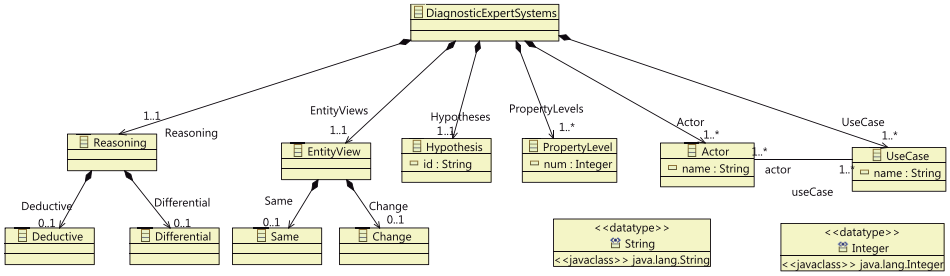


Figure 4 The DCM (see online version for colours)



This FM is translated to the equivalent DCM and it is shown in Figure 4. In this way, the features selected in the FM are instances of the DCM. For example, the domain features in the medical diagnosis are: *entity views = same, property levels = 2, hypotheses = 14, reasonings = differential, use cases = 3, actors = 2, use cases by actor = 2 by actor A & 1 by actor B*. The instances of this model are created by the domain engineer capturing the first variability during the application engineering phase. These instances are used to produce the skeleton base architecture that corresponds to the choice made (see Figure 8).

4.1.2 Second variability in the BOM framework

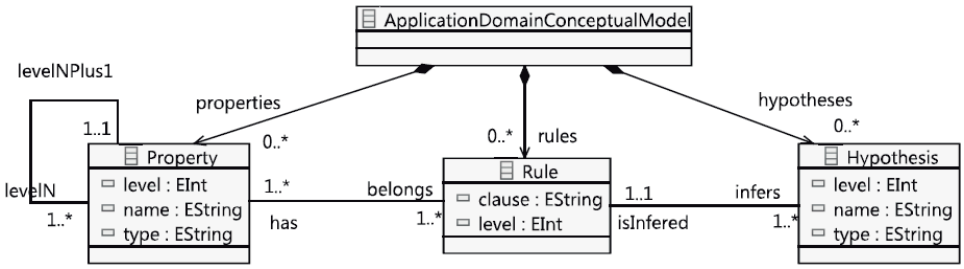
The second variability: V2 variability is also represented and managed using two models: The first model corresponds to the functional conceptual model of the application domain, which is captured by the skeleton base architecture of the SPL1. The second model corresponds to the FM of the variability of the application domain, which is translated to the ADCM. The features of the application domain are used to instantiate the skeletons in order to obtain the PRISMA types (see Figure 8).

It is important to mention that a skeleton base architecture can be instantiated into one or more PRISMA architectures. An example of this situation appears in the two case studies: educative diagnosis and television diagnosis. These two cases share the same skeleton base architectures because they have the same variants of the first variability however they have different PRISMA architectures, because different properties of the two application domains are inserted in each skeleton base architecture.

The second type of variability involves the features of the application domain in a specific field, i.e., the second product line: SPL2. This variability allows the base architectures to be enriched or decorated with the application domain features.

In the application variability management process, the variations of the specific requirements of the application domain should be selected. This selection is reified as an ADCM instance given by the user. The features are inserted in the base skeletons in order to generate the types of the PRISMA software artefacts. These PRISMA architectural elements will be used to configure the PRISMA architectural model of the application.

Figure 5 The application DCM



The ADCM which is shown in Figure 5, captures the application domain variability. The instances of this model are created by the application engineer, and they capture the specific application domain variants. Some instances of this model (i.e., application domain feature) for the diagnosis of infantile infectious diseases are:

```
properties of level 0: cough, fever

properties of level 1: dry_cough, constant_fever

hypotheses of level 1: warmth, parotiditis

hypotheses of level 2: pneumonia, bronchitis

rules: IF (cough = true and fever = true and
respiretory_dificulty = tue) THEN syndrome = warth
```

4.2 Software system views in BOM

In BOM, we have considered two kinds of views for the expert system: the system variability view and the system functional view.

The system variability view is described using the two variability conceptual models: the DCM, which captures the domain and user variabilities (V1), and the ADCM capturing the application domain variability (V2). The DCM conforms to the V1 metamodel (MM V1), and the ADCM conforms to the V2 metamodel (MM V2). Both metamodels are the UML 2.0 class diagram metamodel (http://www.omg.org/docs/formal/05-07-04.pdf), but other domain specific metamodels can be used, producing other domain-specific models to capture the system variability view.

The system functional view of the expert system is given during the production process by means of two views, which are described using three architectural models.

1 The modular view: for the generic architecture model which conforms to the modular metamodel (MM MODULAR).

2 The component-connector view: for the base architecture models which conform to the skeleton metamodel (MM SKELETON) and the PRISMA. Architecture models, which conform to the PRISMA metamodel (MM PRISMA) for the final product. The skeleton metamodel is similar to the PRISMA metamodel (Pérez, 2006), but it allows feature holders (holes).

4.3 The metamodels of the system functional views

We have used the meta object facility (MOF) (http://www.omg.org/mof) graphic notation for the specification of these metamodels. For the specification of the modular and component-connector view metamodels, several proposals have been analysed. The modular and component-connector view types proposed in Shaw and Clements (2006) have also been chosen as the most appropriate views for our approach.

Figure 6 shows the simplified modular view metamodel (MM modular VIEW). The main element considered for this view is the module itself. This figure shows that a model contains a set of modules (which can contain different functions), which are linked to other modules by means of relations (*decomposition*, *uses* and *layer*). This figure only presents the *use* relation. The labels in the links are useful for indicating how the relation is made.

Figure 6 Simplified modular view metamodel

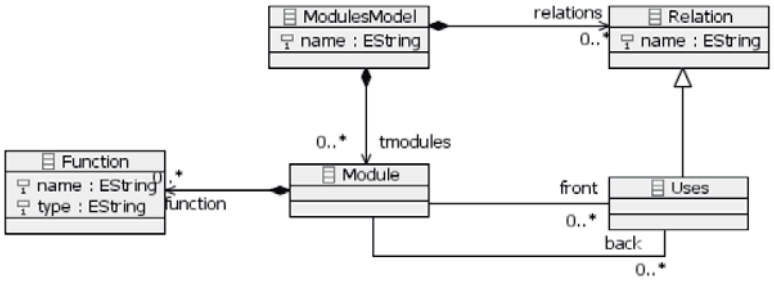
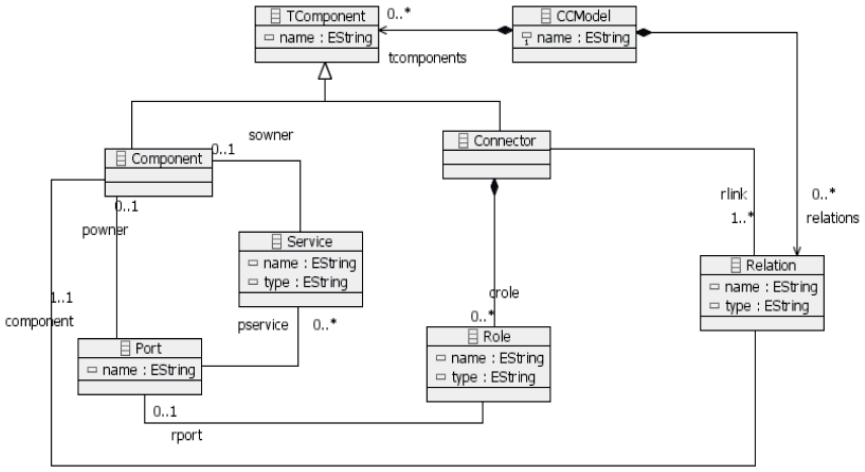


Figure 7 shows the simplified component-connector view metamodel (MM component-connector VIEW). The component class and connector class are the main elements. Both are derived from a more general component class (*TComponent*). The components provide a set of services through a set of ports. The connectors link the ports of components by means of their roles.

Figure 7 Simplified component-connector view metamodel



4.4 Relationships among metamodels

We used OMG query/views/transformations (QVT) (http://www.omg.org/docs/ptc/05-11-01.pdf) to define relationships among metamodels in MOF. Specifically, the QVT-relations language is used to describe them.

The source and target metamodels are identified first. The correspondence among each element of the metamodels must be defined taking into account the way in which their rules are represented through QVT. In this case, the source metamodels are the modular metamodel and the V1 variability metamodel. The target metamodel is the skeleton metamodel. The rules considered in the relationships for the elements of the modular and V1 variability metamodels are check only type (to verify the elements) in the left part and enforce type in the right part to create the elements of the skeleton metamodel.

Two types of relations between the system functional view have been identified:

1 the relations between the modular view and the component-connector view

2 the relations between the component-connector view and the PRISMA view together with the views that define the SPL variability.

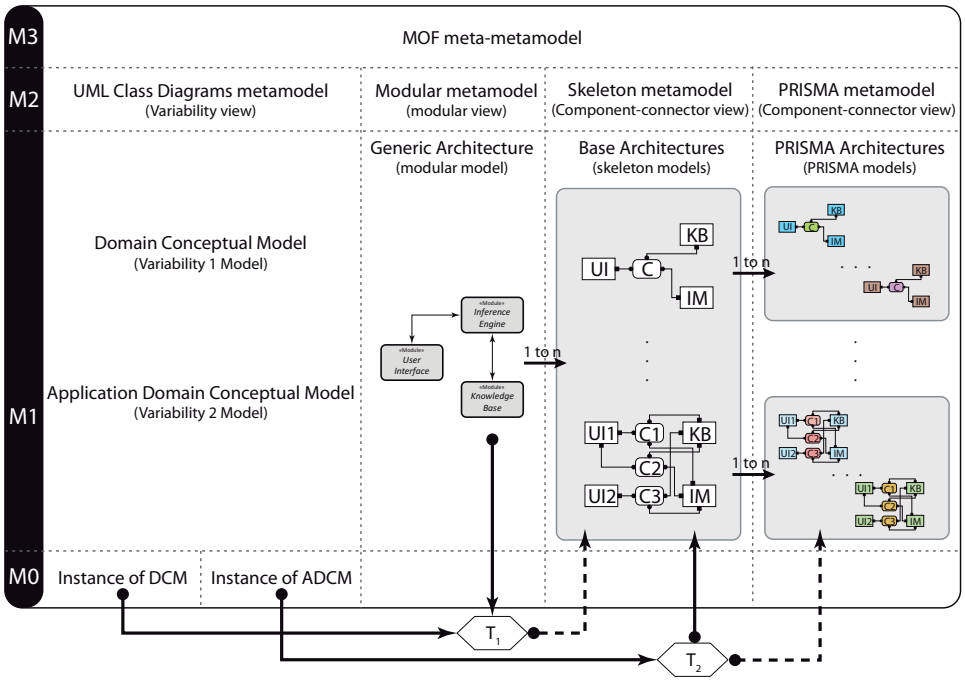
Their respective profiles are:

```
R1: MM_MODULAR X MM_V1 X MM_SKELETON;
R2: MM_SKELETON X MM_V2 X MM_PRISMA;
```

4.5 Transformations among models

Figure 8 shows the transformations involved in the construction of our SPL architecture. This figure illustrates how the transformations performed at the model level (M1) are applied in the transformation processes.

Figure 8 The T1 and T2 model transformations in BOM (located in the MOF levels) (see online version for colours)



The modular model is made up of three modules: *InferenceEngine*, *KnowledgeBase*, and *UserInterface*. These are all linked with each other through dependence relationships. Next, the transformation task for producing the skeleton model (target) is performed. To do this, an instance of the DCM is used. A skeleton component is produced for each module, and each dependence relationship generates a skeleton connector.

The MOF level M1 is where the T1 and T2 transformations are carried out. A first model (skeleton model) is obtained by the T1 transformation, using the modular model

and a DCM instance as sources. This skeleton model is completed by means of the T2 transformation by using an ADCM instance which allows the PRISMA model to be obtained as a refinement. The T1 and T2 model transformations are PIM to PIM transformations.

5 The BOM approaches

In BOM, the variability management is the base of the SPL production plan. If the cardinality of the SPL baseline is:

1 Small: the BOM-EAGER approach is taken because the baseline size is small enough to have all the asset variants explicitly developed and stored in the repository. The baseline is explicit and its assets are built and stored during the domain engineering phase and decision tree techniques driven by V1 instances are used for retrieving the right ones during the application engineering phase. Feature-oriented modelling (FOM) techniques (Cabello, 2008) are used to instantiate the final product (V2 variability).

2 Great: the BOM-LAZY approach is taken. In this case, the baseline size is too big to have all the asset variants explicitly developed and stored in the repository Thus, model transformations are used to enact the production plan (using QVT-relations). In BOM-LAZY, the transformations dynamically compute the needed assets, i.e., the baseline is implicit and the assets are built during the production plan enactment in the application engineering phase.

5.1 The BOM-EAGER approach

In the BOM-EAGER approach, given that the number of base architectures assets is limited, instead of having the T1 transformation (see Figure 8) calculating the assets using QVT on each run of the system as the BOM-LAZY approach does, the assets are developed at once and stored in the baseline for each input in the domain engineering phase.

In this way, given all the variants, T1 is substituted by an access to the repository that implements the baseline. By means of decision tree techniques, we can select the base architecture of the specific case, given the variants of the variability points (features of the domain) as instances of the DCM. The variability points of the first variability are represented in the nodes of this decision tree, and its leaves represent the asset families of the SPL.

In the T2 transformation, using FOM techniques, a decoration process of the base architectures is implemented employing the application domain features given as instances of the ADCM.

5.2 The BOM-LAZY approach

The BOM-LAZY approach uses model transformation techniques in order to define T1 and T2 as PIM to PIM model transformations, which are implemented trough the QVT Relations language.

In the T1 transformation, QVT-relations takes into account the generic architecture, the instance of conceptual model of the domain variability, and the skeleton base architecture configuration. The T1 transformation profile is:

```
T1 (GenericArch_model, instance-V1_model) =
BaseArch_model
```

In T2 transformation, QVT–Relations consider the base architectures, the instance of the application domain variability, and the PRISMA architecture configuration. The T2 transformation profile is:

```
T2 (BaseArch_model, instance-V2_model) = PRISMA-Arch_model
```

5.3 Modelling the BOM approaches

Our work is based on two OMG standards: the RAS which identifies, describes, and packs assets in a standard way; and the SPEM, which defines the standard language for modelling the software process.

In BOM, a clear separation between the domain engineering and application engineering phases is made. This partition is the basis for reuse and automation of the software process. In the domain engineering phase, a set of assets/transformation rules and processes are created. In the application engineering phase, by executing the production plan, these assets are used/created to produce software products of high quality with a minimal cost and time.

In BOM, the domain engineer creates the production plan and all the software artefacts that are necessary to carry out the various tasks. The application engineer provides information of the application domain to the production plan process during its enactment, thus obtaining the final product.

5.3.1 Domain engineering: building assets

In the domain engineering phase as computational independent models (CIM) and platform independent models (PIM) all the software artefacts are built.

The domain engineer creates the baseline as the repository of all the assets necessary to obtain a SPL product. The baseline is structured as a set of Kit-Boxes and the production plan (asset) of the SPL, where we store the assets and the 'know-how' of how to use them in order to produce the SPL. The baseline and its assets are modelled at a high abstraction level.

Figure 9 The baseline (see online version for colours)



The Kit-Boxes are packaged as a new composite asset, and they contain all assets and know-how for building a specific application and are the recovery units. A Kit-Box contains XML documents, processes, models, information, and configurations.

The production plan describes the process by means of tasks or activities to obtain a final product of the SPL. This process shows the production life cycle of our SPL. In the domain engineering phase, this process is specified in SPEM, and it is inserted in the baseline as an asset. In the application engineering phase, the production plan is recovered and enacted to generate an application as one SPL product.

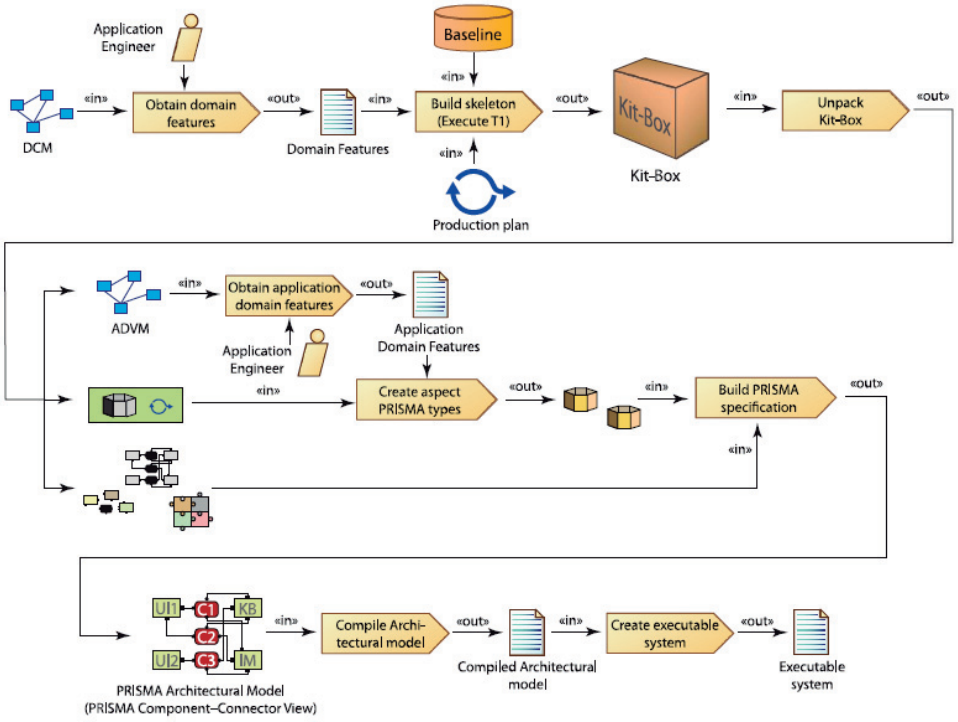
5.3.2 Application engineering: executing the production plan

In the BOM-EAGER and BOM-LAZY approaches, the user (application engineer) builds an application (a product of the SPL), by just giving as input the features of the variabilities V1 and V2 by means of instances of the DCM and ADCM conceptual models, respectively.

5.3.2.1 BOM-EAGER production plan

The production plan of our SPL in the BOM-EAGER approach is shown in Figure 10 by using SPEM notation.

Figure 10 Production plan through BOM-eager approach (see online version for colours)



The production plan starts when BOM obtains (from the application engineer) the features expressed as DCM's instances of the variability points of the first variability. Next, BOM selects the assets from the baseline, i.e., one Kit-Box asset that corresponds to the specific product. The Kit-Box that is selected by the engineer using BOM must be unpackaged in order for each asset to be used.

One of the assets recovered from the Kit-Box is the ADCM, which is used by BOM in order to obtain (from the application engineer) the features of the application domain considered as variants of the second variability.

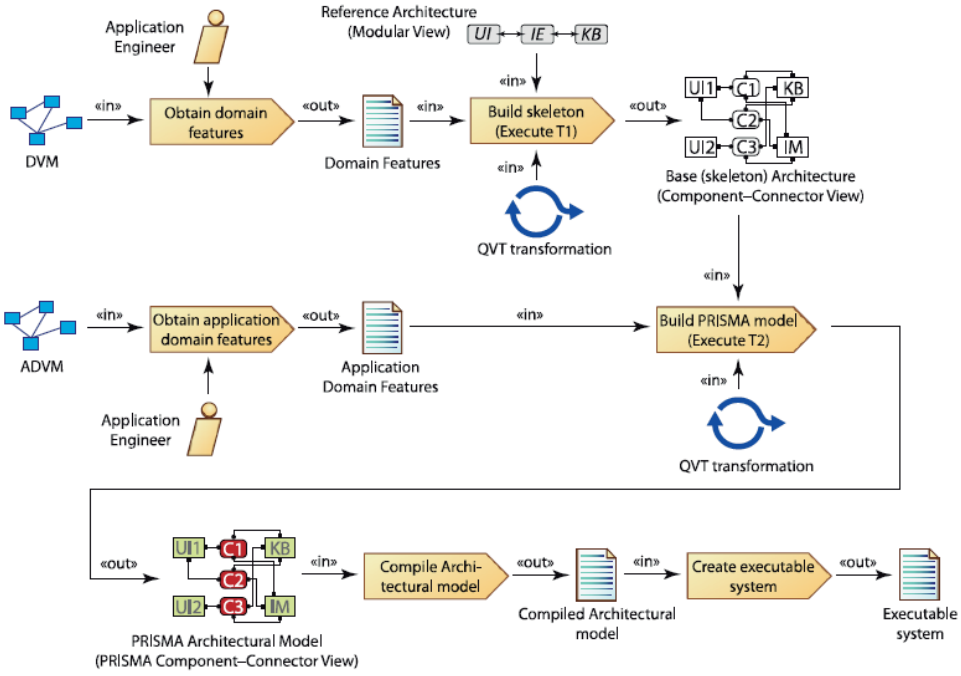
Other assets recovered from the Kit-Box are the packaged hybrids. These assets are in turn unpacked to produce the PRISMA type aspects. BOM applies the main feature insertion process (which invokes the individual features insertion processes) to fill the selected skeletons (aspects) with the specific features of the case study defined by the engineer, thereby creating the PRISMA type aspects.

BOM uses the aspect PRISMA types and the other PRISMA type artefacts (interfaces architectural elements, and architectural model) to create the PRISMA configuration. These artefacts are input to the PRISMA-MODEL-COMPILER tool (Pérez et al., 2005) to automatically generate the code (in C# .NET). At the end, BOM creates the final system as an executable application, i.e., a final product of the SPL. This application is executed on top of the PRISMA-NET MIDDLEWARE (Pérez et al., 2005).

5.3.2.2 *BOM-LAZY production plan*

The production plan of our SPL taking the BOM-LAZY approach is shown in Figure 11 by using SPEM notation.

Figure 11 Production plan through BOM-lazy approach (see online version for colours)



The production plan starts when BOM obtains (from the application engineer) the features expressed as DCM's instances of the first variability. Next, BOM executes the QVT-relations transformation T1 and obtains the base architecture that corresponds to the specific product. Then, BOM executes the QVT-relations transformation T2 and produces the PRISMA architectural model. The ADCM is used by BOM in order to obtain (from the application engineer) the features of the application domain considered as variants of the second variability. This PRISMA type artefact is sent to the PRISMA-MODEL-COMPILER tool, and the process continues as the BOM-EAGER approach does.

6 The BOM framework implementation

This section presents briefly the development of our prototype for the BOM framework.

6.1 BOM-EAGER approach implementation

The prototype for the BOM-EAGER approach is called ProtoBOM (Cabello et al., 2008). The production plan process diagram is used as a graphical metaphor for the user interface (GUI) in SPEM notation.

ProtoBOM can be used by the application engineer to generate a product of the SPL. ProtoBOM integrates the use of several tools, offering an approach to build applications, in a simple way. Some of the tools have been developed on purpose in ProtoBOM and others had already been created for other domains. The application engineer only introduces the variability features to the system in order to build an application (product of SPL). The rest of the activities will be carried out automatically by ProtoBOM.

In ProtoBOM, the V1 variability is solved by means of our tool using decision tree techniques in order to select a base architecture. The T2 transformation is executed by means of our tool using FOM techniques, in order to decorate this architecture. The baseline has been implemented as a repository accessed by a web service. In this way, the SPL that are developed can be easily shared and distributed, which promotes the reuse of the assets contained in the baseline.

The asset selection process computes paths in the decision tree which are used to select the assets in the baseline. This process is created by the domain engineer and uses the decision tree and the assets of the baseline. This process is executed by ProtoBOM in the application engineering phase, when the application domain engineer instantiates the DCM (i.e., inserting the specific domain information into the system as variants of the domain variability points). This process uses the decision tree techniques in order to carry out the V1 resolution.

The feature insertion process is used to insert the application domain features in their specific skeleton aspect. This process uses the FOM technique in order to carry out the V2 resolution.

We explain how to insert the application domain features in a skeleton aspect in order to obtain the corresponding PRISMA type aspect. The features are modelled as functions, representing refinements of the input model. In the educational programme diagnosis, we have: Fx.i • S-KB-EDj, which means 'add feature Fx.i to the S-KB-EDj model', where S-KB-ED0 is the skeleton-knowledge base educational domain, $0 \leq j \leq n$, and '•' denotes the application of the function. The gluing process is iterated step by step:

S-KB-EDj + 1 = Fx.i • S-KB-EDj

In Table 1, we present an example of the functional aspect skeleton and its respective aspect type, for the knowledge base component of the educational programme diagnosis.

Table 1 Example of a skeleton and PRISMA type aspects

Functional aspect of the knowledge base of a skeleton	*Functional aspect of the knowledge base of a PRISMA type*
Functional Aspect FBase using IDomain	Functional Aspect FBase using IDomain
Attributes	Attributes
Variables	Variables
< FP.0 >,	pictures:string;
	TVstructure:string;
	music:string;
	...
Derivеds	Derivеds
< FP.1 >,	production:string;
	content:string;
	technicalQuality:string;
< FH >	stateVideo:string;
Derivations	Derivations
< FR.1 >	{pictures = "proper" and TVstructure = "authorized" and music = "good"} production: = "good";
	...
< FR.2 >	{production = "good" and content = "good" and technicalQuality = "good"} stateVideo: = "VideoOK";
.....	.....
Services	Services
.....	.....
End_Functional Aspect FBase	End_Functional Aspect FBase

These software artefacts are XML documents that are specified using the PRISMA-ADL. In order to simplify this example, we have omitted the XML syntax. In this table, <Fx.i> are the features place holders. These features are the following: FP.i = features of the properties of level i, FP.i = features of the properties of level i, FH = features of the hypothesis, FR.i = features of the rules of level i.

We have used the PRISMA-MODEL-COMPILER to automatically generate C # code from the system architectural models and to create the application (an instance of the SPL), which is executable over the PRISMA-NET-MIDDLEWARE.

An example of (part of) the generated C# code (by the PRISMA-MODEL-COMPILER) that corresponds to the knowledge base component of the educational programme diagnosis case study is the following:

```
namespace KBMD {
   [Serializable]
   public class KnowledgeBaseEducational : ComponentBase {
     public class KnowledgeBaseEducationalDiag string name
     : base(name) {
       AddAspect(new FBaseED());
          InPorts.Add("KnowPort", "IDomainED", "KNOW");
          OutPorts.Add("KnowPort", "IDomainED", "KNOW");
}    }    }
```

For the same case study, Figure 12 shows two captures of the GUI executable system over the PRISMA-NET-MIDDLEWARE.

Figure 12 Some dialogs of the executable system (see online version for colours)



6.2 BOM-LAZY approach implementation

The BOM-LAZY architecture is shown in Figure 13. In this approach, the model transformations T1 and T2 are executed by means of our tool using the QVT–Relations language.

Figure 13 The BOM-lazy architecture

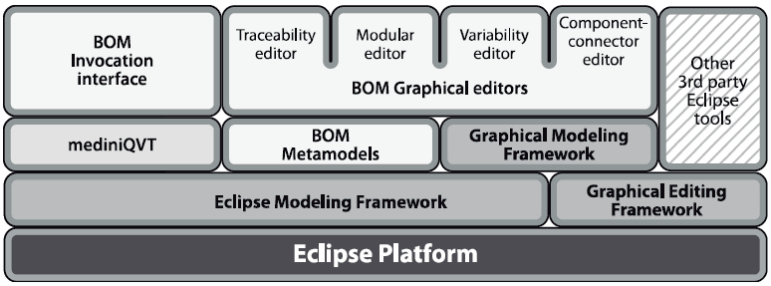


Figure 14 shows a rule (*ModulesModelToComponentsModel*) defining the T1 transformation, expressed in the graphical notation of the QVT-relations language. For this transformation, the rule relates the *ModulesModel* element of the modular metamodel with the *CCModel* element of the skeleton metamodel, which assigns the same name. For

T2 transformation, the rule relations the *CCModel* element of the skeleton metamodel into the *PRISMA_Architecture* element of the PRISMA metamodel, which assigns the same name. In Gómez (2009), we present all rules.

Figure 14 A rule in graphical notation of the QVT-relations language

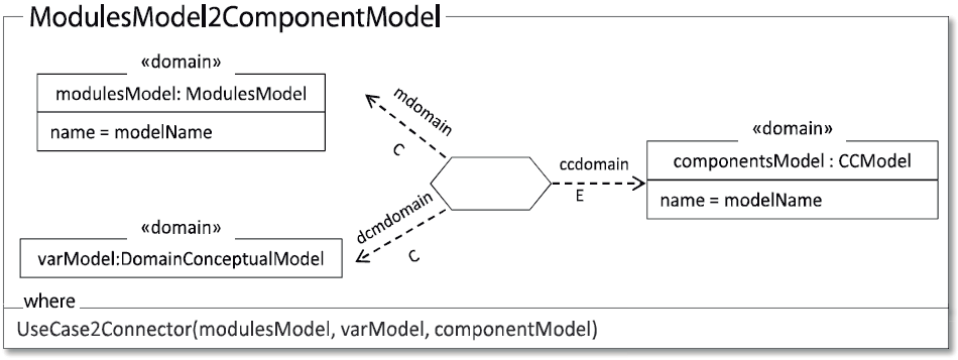
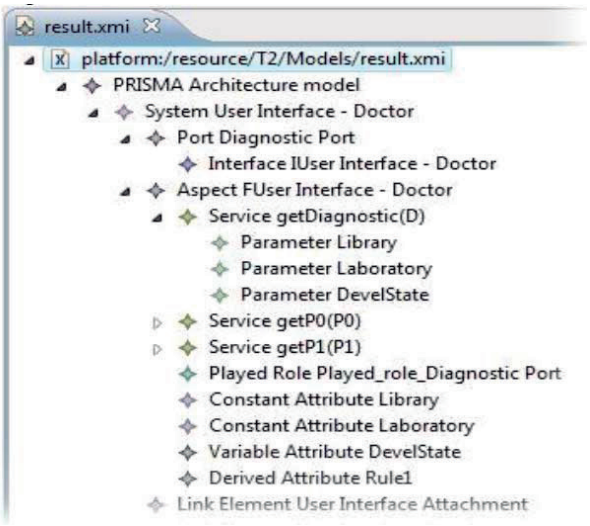


Figure 15 shows the PRISMA architecture in the medical diagnosis case study.

Figure 15 Example of a generated PRISMA model (see online version for colours)



7 Related works

There are a great number of works that are related to our approach. The methodologies and applications on this subject have produced a wide variety of research products, offering suggestions and solutions in specific domains. Our research is related to the following works:

1 A study made by Liao (2005) mentions that the development of ES has been characterised by the separation of knowledge and processes as independent units. In our architectural model, the elements in the type level are defined taking into account this concept, specifically when there is a component that contains the domain knowledge and another component that executes the inference process of the diagnosis.

2 ES have also been implemented in the development of different programming paradigms, which are oriented toward fourth-generation languages and visual programming methods to provide user-friendly communication. In BOM, we use the PRISMA Architecture Description Language (ADL) to define an architectural model that follows the MDA approach, which allows the automatic generation of code.

3 Our work also applies the detection of the components based on the functional decomposition of the problem, which is compatible with the architecture-based design method (ABD) methodology (Bachman et al., 2000). This methodology has been applied in the building of our architectural models.

4 The work by Garlan et al. (2001) is a very important reference in establishing the elements of a complex software system. In his work, he introduces the component, connector, system, input port, and output port concepts. These concepts have been included in our model.

5 MDA proposes the definition of models at high abstraction level, which are independent of the technology (PIM). In our work, we have considered this line of research focusing on diagnostic experts systems that are based on product lines.

Other works of the SPL technical space related to our approach are the following:

1 Batory et al. (2006) capture the domain features in a FM. In BOM, we capture features in two kinds of FMs. In our research, we observed that the variability problem is not solved by means of a unique FM and the monotonic gluing of these features. We have taken a new approach, which manages the variability in two phases (one by selecting a base architecture using the domain features, and another one by decorating these base architectures with the application domain features) in order to obtain the final product.

2 Trujillo (2007) uses feature-oriented programming (FOP) (http://aosd.net/2005/tutorials/fop.php) as a technique for inserting features into XML documents by means of XSLT templates. In BOM, we use this technique but at the model level, i.e., we use FOM. The features are inserted on the model skeleton aspects of the artefacts that make up the PRISMA architecture model (represented as XML documents) in two ways: In the BOM-EAGER approach by means of FOM techniques, and in the BOM-LAZY approach by means of model transformation techniques.

3 Clements and Northrop (2001) use the SPL development approach, considering a clear division between the domain engineering and application engineering phases for the reuse and the automation of the software process. In BOM, we have used this approach to develop our SPL.

4 Ávila-García et al. (2006) use process modelling in SPEM and RAS to pack reusable assets. In our approach, we also use both OMG standards: RAS when we model our baseline as a repository of all the assets, and SPEM when we model the production plan of the SPL.

5 Bachmann et al. (2003) propose separating the variability declaration of the affected assets into separate artefacts. In BOM, the specification of the variability and the functionality are captured in different models. The use of instances of FMs allows the user to input the information of the domain features. Those features allow to select/build the associated assets, or to define the application domain features in order to configure the final application.

6 Bachmann et al. also propose a variability metamodel formed by variability points and variants as the main elements of this metamodel. Our feature metamodel provides features as variability points and their sub-features as variants.

7 Gomma and Shin (2007) present a semi-orthogonal variability model, where the variability is represented in a separate view, but the variability points and their variants are shown in a stereotypical way in a UML diagram. In our work, we use a specific variability model (FM) in order to specify variability points (features) and their variants (sub-features).

8 Czarnecki and Kim (2005) propose a notation for cardinality-based feature modelling. Our work shares most of this notation as it is widely known and used, but we have included some variants. Moreover, we have provided tooling support for this notation, and we have integrated it in our prototype.

9 Chen et al. (2009) mention in a systematic review of SPL that there are a great number of proposals for variability management. Specially, most of them are based on the original FODA notation and propose several extensions to it. Our work is closely related to previous research in feature modelling however, there are several distinctive aspects, which are shown in Gómez and Ramos (2010).

10 González-Baixauli and Laguna (2005) apply the MDA trend and requirements engineering of product lines. We take into account the requirements of the final user by means of a use cases UML diagram, which they are used in an explicit way in the DCM.

8 Conclusions and future work

In this paper, we have presented a generic solution to the development of applications by means of our BOM approach. BOM implements two approaches depending on the SPL cardinality: the BOM-EAGER approach and the BOM-LAZY approach.

We follow the MDA proposal for building domain models (as CIM), which are automatically transformed (into PIM) and then compiled to executable applications (as PSM). We also describe how the variability is managed in our SPL by means of our BOM framework.

Our approach also enhances the development of software systems by applying SPL techniques as they are useful when the members of a family of systems share a common design (software reuse). Thus, a specific design can be used in different products,

reducing costs, time to market, effort and complexity. By applying MDA techniques, we are able to build systems that are platform-independent, and we can think about them from the problem perspective and not the solution perspective. This makes it possible to apply these solutions to different domains. Moreover, we provide a framework that integrates several technical spaces where modern software development techniques coexist in a coordinated way.

The modelling process is based on the MDA proposal which has two main advantages of metamodelling:

1 to create the metamodels of the views to generate models

2 to maintain consistency between models.

The strategy of transformation of models maintains consistency between the models considered, when one of them changes for maintenance and evolution. Metamodels serve as templates to create the models, so when a model is created, it contains only what is specified in your specific template, in other words, no surprise elements or relationships appear.

In order to illustrate BOM, we have selected the diagnostic ES domain. In this context, BOM is a framework that automatically generates software applications as PRISMA architectural models by using SPL techniques.

We can conclude that the main characteristics of BOM are the following:

1 Variability is managed at a high abstraction level (i.e., at the model level rather than at the program level).

2 The system variability is modelled using models that are separate from their functional models. The DSL for expressing the variability are suited for the domain, instead of adding tangled variability annotations directly to the functional models (UML or ADL) as other approaches have proposed.

3 Variability is operated by two orthogonal ways: one provided by the features of the domain, and another one provided by the features of the application domain.

4 The variability is given by instances of the conceptual models (DCM and ADCM).

5 Model generation and transformation are implemented using decision tree techniques and FOM techniques, or QVT-relations, in the production plan. In BOM, the model transformations are resolved in an effective, scalable and user friendly way and its expressivity is also richer and clearer.

6 Various technological spaces are integrated to deal with the complexity of the problem. They are current trends in software engineering.

7 BOM uses OMG standards and implements a generic approach to SPL development that can be applied to different domains, application domains, systems, and platforms.

8 BOM offers an approach to build software applications in a simple way: the user only inputs the features of the domain and the application domain.

In the future, we want to extend the experience gained in the diagnostic domain to other application domains in order to offer more generality. In addition, we want to improve the DSL for the diagnostic domain. Finally, we plan to validate the BOM framework in

real-life cases by means of our final prototype, which will integrate both the BOM-EAGER and the BOM-LAZY approaches. We also plan to use benchmarks in order to compare our prototype results with other approaches. This is part of the target of a new Project: MULTIPLE, which will be developed over the next three years by the authors' research team.

Acknowledgements

This work has been funded under the Multimodeling Approach for Quality-Aware Software Product Lines: MULTIPLE Project TIN2009-13838, and FPU fellowship programme from the Spanish government, Ref. AP2006-00690.

References

Ávila-García, O., García, A.E., Rebull, V.S. and García, J.L.R. (2006) 'Integrating process models and assets an MDA tool', *XI Conference on Software Engineering and Databases*, Barcelona, España, in Spanish.

Bachman, F., Bass, L., Chastek, G., Donohoe, P. and Peruzzi, F. (2000) 'The architecture based design method', Technical Report CMU/SEI-2000-TR-001, Carnegie Mellon University, USA.

Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B. and Vilbig, A. (2003) 'A meta-model for representing variability in product family development', *5th International Workshop on Product Family Engineering*, pp.66–80.

Batory, D., Benavides, D. and Ruiz-Cortés, A. (2006) 'Automated analyses of feature models: challenges ahead', *ACM on Software Product Lines*.

Cabello, M.E. (2008) 'Baseline-oriented modeling: an MDA approach based on software product lines for applications development', PhD thesis, Universitat Politècnica de València, España, in Spanish.

Cabello, M.E., Gómez, M., LLavador, M. and Ramos, I. (2008) 'ProtoBOM: a framework that semi-automatically generates decision support systems based on software product lines', Technical Report DSIC II/02/08, Universitat Politècnica de València, España, 68 pages.

Chen, L., Babar, M.A. and Ali, N. (2009) 'Variability management in software product lines: a systematic review', *13th International Software Product Lines Conference*, CA, USA.

Clements, P. and Northrop, L.M. (2001) 'Software product lines: practices and patterns', *SEI Series in Software Engineering*, Addison Wesley, USA.

Czarnecki, K. and Eisenecker, U. (2000) *Generative Programming: Methods, Tools and Applications*, pp.267–304, Addison-Wesley, ISBN 0-201-30977-7.

Czarnecki, K. and Kim, C.H.P. (2005) 'Cardinality-based feature modeling and constraints: a progress report', *OOPSLA International Workshop on Software Factories*, San Diego, California, USA.

Garlan, D., Cheng, S. and Kompanek, A.J. (2001) 'Reconciling the needs of architectural description with object modeling', *Notations. Science of Computer Programming Journal*, Special UML Edition, Elsevier Science.

Giarratano, J. and Riley, G. (2004) *Expert Systems: Principles and Programming*, 4th ed., ISBN: 0534384471, Hardcover.

Gómez, A. and Ramos, I. (2010) 'Cardinality-based feature models and model-driven engineering: fitting them together', *Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, Vol. 37, pp.61–68.

Gómez, M. (2009) 'Gestión de la variabilidad en Líneas de Productos Software', Bachelor thesis. Universitat Politècnica de València, Spain, in Spanish.

Gomma, H. and Shin, M.E. (2007) 'Automated software product line engineering and product derivation', *40th Hawaii International Conference on System Sciences*, Waikoloa, Hawaii, USA.

González-Baixauli, B. and Laguna, M.A. (2005) 'MDA e Ingeniería de Requisitos para Líneas de Producto', *Taller sobre Desarrollo Dirigido por Modelos*, Granada, Spain, in Spanish.

Kent, S. (2003) 'Model driven engineering', *Integrated Formal Methods, Third International Conference*, Turku, Finland, Springer-Verlag-LNCS, Vol. 2335, pp.286–298.

Liao, S.H. (2005) 'Expert systems methodologies and applications – a decade review from 1995–2004', *Expert Systems with Applications*, Vol. 28, No. 1, pp.93–103.

Pérez, J. (2006) 'PRISMA: aspect-oriented software architectures', PhD thesis, Universitat Politècnica de València, Spain.

Pérez, J., Ali, N., Costa, C., Carsí, J.A. and Ramos, I. (2005) 'Executing aspect-oriented component-based software architectures on .NET technology', *3rd International Conference on .NET Technologies*, pp.97–108, Czech Republic.

Shaw, M. and Clements, P. (2006) 'The golden age of software architecture', *IEEE Software*, Vol. 23, No. 1, pp.31–39.

Trujillo, S. (2007) 'Feature-oriented model driven product lines', PhD thesis, Universidad del País Vasco, Spain.

A Multimodeling Approach for Quality-Driven Architecture Derivation

Emilio Insfran, Silvia Abrahão, Javier González-Huerta, John D. McGregor, Isidro Ramos

Abstract. Product architecture derivation is a crucial activity in Software Product Line (SPL) development since an inadequate decision during the architecture design directly impacts the quality of the product under development. Although some methods for architecture derivation have been proposed in the last few years, there is still a need for approaches that model the impact among architectural design decisions and quality attributes and use this information to drive the derivation of high-quality product architectures. In this paper, we present an approach for integrating quality attributes in early stages of the SPL lifecycle. The approach is based on a multimodel that explicitly represents the product line from multiple viewpoints (e.g., variability, functional and quality) and the relationships among them, as well as on a derivation process that makes use of this multimodel to derive a product architecture with the required quality attributes from the product line architecture. The feasibility of the approach is illustrated using a case study on the automotive domain.

1 Introduction

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features developed from a common set of core assets in a prescribed way [7]. Software product line engineering (SPLE) aims to develop these systems by taking advantage of the massive reuse of core assets in order to improve time to market and product quality.

Software product line development has characteristics, which distinguish it from the development of individual products. In SPL development there is a product line architecture, which includes the variations needed to cover the different products, and the product architectures, which are instances of the product line architecture achieved by exercising variation mechanisms. However, in those cases

Emilio Insfran, Silvia Abrahão, Javier González-Huerta, Isidro Ramos
ISSI Research Group, Universitat Politècnica de València, Camino de Vera s/n, Valencia, Spain.
e-mail: {einsfran, sabrahao, jagonzalez, iramos}@dsic.upv.es

John D. McGregor
Department of Computer Science, Clemson University. 312 McAdams Hall, Clemson, USA
e-mail: JohnMc@cs.clemson.edu

Insfrán, E., Abrahao, S., Javier González, McGregor, J. D., Ramos, I. A Multimodeling Approach for Quality-Driven Architecture Derivation. Proc. of the 21st International Conference on Information Systems Development (ISD2012). 2012.

where levels of quality attributes that fall outside the original specification of the product line are needed (and cannot be achieved by the product line variation mechanisms), some architectural transformations may be applied when deriving a product architecture to ensure that these levels of quality attributes are met [3].

Although several methods for architecture derivation using multiples viewpoints have been proposed in the last few years (e.g., [2], [4], [17], [9]), there is still a need for approaches that model the impact among architectural design decisions and quality attributes and use this information to drive the derivation of high-quality product architectures. The objective of this paper is to present a multimodel (i.e., a set of interrelated viewpoints describing the products that can be derived from the product line) as well as its use for deriving product architectures with the required quality attributes from the product line architecture. This multimodel contains the functional and variability viewpoints, and also integrates a quality viewpoint allowing the explicit representation of the relationships and constraints among the elements of these viewpoints.

The main contribution of the paper is the use of model-driven engineering principles as a way to provide a richer semantic representation of a software product line (the multimodel). It provides a sufficiently formal interrelated model that can be supported by tools capable of automating portions of the product line production planning. In this paper, we illustrate how the multimodel can be used as input in a quality-driven derivation process that will apply architectural transformations (i.e., architectural patterns) to the product line architecture in order to derive product architectures with the required levels of quality attributes, when this cannot be achieved by exercising the product line built-in variation mechanisms.

The remainder of the paper is structured as follows. Section 2 discusses existing approaches for the automatic derivation of product architectures. Section 3 provides details of the multimodel concept and the different viewpoints that can be used for representing a software product line. Section 4 describes a quality-driven model transformation process that allows obtaining product architectures with the desired quality attributes. Section 5 illustrates the feasibility of the approach through a case study in the automotive domain. Finally, conclusions and future work are presented in Section 6.

2 Related Work

Several approaches dealing with the use of multiple viewpoints for the automatic derivation of product architectures have been proposed in the last few years (e.g., [2], [4], [17], [9]).

In the approach by Botterweck et al. [2] the product architectures are produced by means of an ATL model transformation process, which takes as input a domain architecture model and an application feature model and generates an application architecture model, simply by copying the elements. This approach describes the product line architecture as the union of components that can be present in any

configuration, but does not explicitly address quality attributes. Similarly, Cabello et al. [4] produce product architectural models by means of a QVT transformation. The transformation takes as input the variability view expressed in a feature model and the modular view of the architecture and generates the PRISMA component and connector architectural view. Furthermore, Perovich et al. [17], automate the derivation of product architectures by taking as input a feature configuration model. The transformation encapsulates the knowledge of how to build the product architecture when the corresponding feature is present in the feature configuration model. Finally, Duran-Limon et al. [9] present an approach to modeling the architectural variation points and the variability tree and then using ontology queries to obtain the commonalities, variants and dependencies of a set of selected features. The output generated by the ontology queries is the input of an ATL transformation that generates the product architecture.

These approaches automate, to varying degrees, the derivation of product architectures by means of transformations using multiple viewpoints. These approaches do not integrate quality attribute concerns in the transformation process.

3 A Multimodel for Describing Software Product Lines

A *multimodel* is a set of interrelated models that represents different viewpoints of a particular system. A *viewpoint* is an abstraction that yields a specification of the whole system restricted to a particular set of concerns. In any given viewpoint it is possible to define a model of the system that contains only the objects that are visible from that viewpoint. Such a model is known as a *viewpoint model*, or a *view* of the system from that viewpoint [1].

In SPL development, the multimodel can be used to represent the different viewpoints of a set of products that can be derived from the product line. We distinguish (at least) three different viewpoints: variability, functional and quality. The relationships with the quality viewpoint express the relative impact that the elements of the variability or the functional view have on the quality attribute levels of the final product. A quality attribute level indicates a threshold which a system must cross to be acceptable to the stakeholders.

The multimodel plays a key role in the SPL development process at two different levels: i) in the *Domain Engineering* phase to express the impacts and constraints among variations, functional components and quality attributes, providing in this way, a more comprehensible and integrated model of the product line, and ii) in the *Application Engineering* phase to provide information to the transformation processes that will integrate the production plan. The production plan describes how the products are produced from the core assets including the derivation of product architectures from the product line architecture.

3.1 The Multimodel Viewpoints

The multimodel is composed of three viewpoints: variability, functional and quali-

ty. The **variability viewpoint** has been defined using a variant [12] of the cardinality-based feature model presented in [5]. The main element of this model is the *feature*, which is a user-visible aspect or characteristic of a system. The cardinality-based feature model is a hierarchy of features, but also allows the definition of how many clones of a feature can participate in a specific product configuration. Cloning features is useful in order to define multiple copies of a part of the system that can be differently configured [12]. Features can be organized in *feature groups*, which can have *group cardinalities*. This cardinality restricts the minimum and the maximum number of group members that can be selected. In feature models it is common to define relationships and constraints among features. According [12], these relationships can be hierarchical: i) *has_a*: structural relationship between a parent and *n* copies of a child feature as defined by its cardinality ii) *is_a*: structural relationship between a parent feature and a group of child features; non-hierarchical: i) *implication*: the presence of the feature A implies the presence of the feature B, ii) *co-implication* the presence of the feature A implies the presence of the feature B and vice versa, iii) *exclusion*: the presence of the feature A implies that the feature B should not be present and vice versa, and iv) *use*: specifies that a specific instance of feature A will be related to one (or more) specific instances of feature B as defined by its cardinality. This specific variation of the cardinality view feature model also allows adding OCL-like constraints to the model and, based on these constraints, to check whether the instances are valid or not. The feature model variant has been specifically defined for being applied in a Model-Driven Product Line development context.

The **functional viewpoint** on the multimodel represents the structure of the system containing the different software components that satisfy the requirements of the different features and the SPL architecture. The functional view can be defined using different styles (e.g., component-and-connector, modular, allocation) [6]. In the last years, several Architectural Description Languages (ADL's) and notations has been proposed for documenting architectures such as AADL, SysML, UML, or EAST-ADL. We have selected the Architectural Analysis and Design Language (AADL) [10] for expressing the functional view of the multimodel because i) it is an extensible standard in the avionics, aerospace, automotive and medical device industries; ii) it provides mechanisms for dealing with variability at the architectural level, as demonstrated in [19]; iii) AADL models are a good basis for evaluating quality attributes that focus on the design of an architecture [6]; and iv) the structure of the toolsets supporting AADL allows the implementation of analysis models operationalizing different software measures. The AADL standard defines a textual and graphical representation of the runtime architecture of software systems as a component-based model in terms of tasks, their interactions and the hardware platform where the system is executed.

The **quality viewpoint** on the multimodel has been defined by a quality model which is an extension of the ISO/IEC 25010 (SQuaRE) standard [14] to evaluate the quality of SPLs. Fig. 1 shows the main classes of the metamodel that gives support to the quality view, with the hierarchical decomposition of quality in characteristics, sub-characteristics and quality attributes. The model shows that a

quality attribute can be associated with different quality perspectives, which are characterized by a SPL development phase and/or type of stakeholder. A quality attribute can positively or negatively impact other quality attributes (e.g., testability impacts positively on maintainability, maturity of core assets impacts positively on reusability). Those impacts can be expressed using a nominal or ordinal scale measure (when it is described by a label expressing the relative importance or severity) or using a ratio scale measure (when it is described by a numeric value that expresses the impact). The model also expresses the software measures associated with each quality attribute. Each measure can be operationalized depending on the software artifact and/or the quality perspective that is selected.

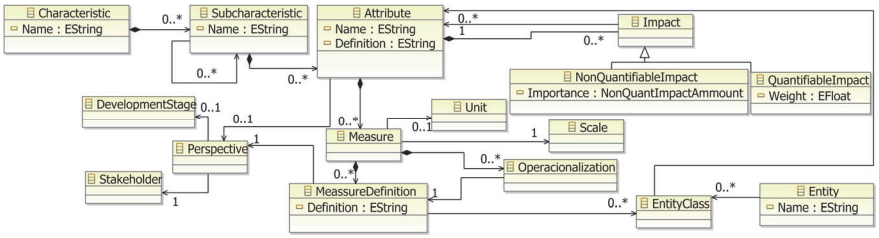


Fig. 1. Quality View Metamodel

3.2 Relationships among Viewpoints

In order to define production plans to guide the development of software products using the multimodel we should establish relationships among the elements of the *viewpoint models* or *views* that represents the product line viewpoints. This will allow us to formally analyze properties over the product line as a whole. The multimodel has been defined with the following relationships among views:

- *Functional and variability views:* a set of elements in the functional view (e.g., AADL systems, system implementations, devices, and processes) can be combined in order to fulfill the requirements of one or more features (e.g., a multimedia GPS navigator feature in a car is fulfilled by a combination of software and hardware components) and can be seen as a subset of the relationships that can be defined applying the VML4Arch approach [21]. Furthermore, the constraints defined in the feature model allow this model to become a filter to partially determine what elements can be selected in the functional model.
- *Functional and quality views: components* of the functional view may impact one or more *quality attributes* (e.g., integrating *components* in the functional view with low resource consumption will impact on the *resource consumption* quality attribute of the quality view). During the product derivation the application specific quality attributes, selected by the application engineer, and the impacts established among the functional and quality views drives the selection of the most appropriate components that satisfy the quality attribute requirements.

- *Variability and quality views: features* or *feature groups* in the variability view may impact one or more *quality attributes* of the product (e.g., a feature group related to safety options may improve the reliability of a system).

The multimodel helps to have a richer semantic view of the product line as well as to ensure the consistency of the relationships among the different elements of the product line viewpoints. Furthermore, the product line production planning can be guided by the structural constraints established by the multimodel.

4 Quality-Driven Product Architecture Derivation

Product definition begins with the derivation of the product architecture. The multimodel can be used to create product-specific production plans that describe how a specific product can be build from the core assets, including how the product architecture can be derived from the product line architecture. The product line architecture defines the allowable variations within the product line's scope and the variation mechanisms for achieving them. In some cases the product architecture may include variation points that are not permitted by the original product line architecture or to assure some product-specific quality attributes. We describe how the multimodel is used for deriving product architectures with the required quality attribute levels from the product line architecture. In this work, we focus on *architectural patterns* [3] [8], which are represented as *architectural transformations*, as a means to ensure the quality of the product architectures.

In the domain engineering phase, our approach requires the domain expert to establish the impacts that the different alternative architectural transformations have on the quality attributes of the product architecture. The impact that each alternative has on the quality attributes can be determined by using empirical evidence or based on the domain experts' experience. In order to perform a trade-off analysis among quality attributes, the Analytic Hierarchy Process (AHP) can be applied. AHP [18] is a decision-making technique used to resolve conflicts in which it is necessary to address multi-criteria comparisons. The result of the AHP is a comparison matrix that shows the relative importance of each alternative architectural transformation with regard to each quality attribute and is used as input for the quality-driven product architecture derivation process. The trade-off process do not consider all the possible quality attributes in a given domain but only those quality attributes that could be directly impacted by the architectural patterns and which are of relevance in a specific context (e.g., safety-critical systems for the automotive domain).

Fig. 2 shows an overview of this derivation process, which is performed in the application engineering phase. The starting point is a product line architecture that has been designed by the product line architect for supporting the whole set of possible products within the product line scope. The process contains two different activities, which can be performed by model transformations: product architecture derivation and product architecture evaluation.

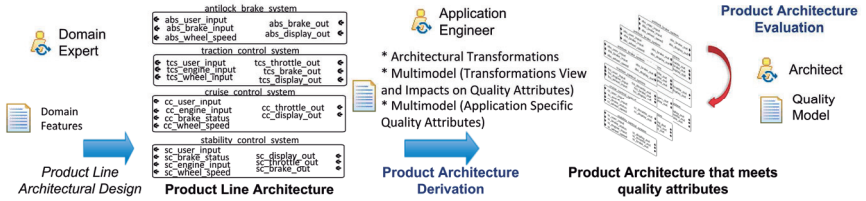


Fig. 2. Overview of the Quality-Driven Product Architecture Derivation Process

The first model transformation (product architecture derivation) creates the product architecture as an instance of the product line architecture by applying an architectural transformation to meet the required quality attribute levels. This transformation takes as input i) the product line architecture ii) the comparison matrix with the impacts among the alternative architectural transformations and the quality attributes, and iii) the desired product-specific quality attributes selected by the application engineer. This transformation generates as output a product architecture. The second model transformation (product architecture evaluation) applies the software measures from the quality view of the multimodel to the derived product architecture in order to evaluate if it satisfies the desired quality attribute levels. This transformation takes as input the derived product architecture, the required quality level, and the quality view from the multimodel and generates as output an evaluation report.

5 Case Study

This section presents a case study in the context of the safety-critical embedded systems from the automotive industry. The Vehicle Control System (VCS), which is based on the example introduced in [13], was modified and extended in order to apply the variation points described in [19]. This system is composed of several subsystems (features) which are: i) Antilock Braking System (ABS), which ensures that the maximum braking force is transmitted to all four wheels of the vehicle; ii) Traction Control System (TCS), which prevents the wheels from slipping; iii) Stability Control System (SCS), which keeps the vehicle going in the direction in which the driver is steering the car; and iv) Cruise Control System (CC), which attempts to maintain a constant driver determined speed.

These features comprise a set of embedded systems present in modern automobiles. Fig. 3 shows the inner structure of the product line architecture for the subsystems, which are composed of *sensors* that collect input signals, *processors* that process and transforms the inputs, and *actuators* that control the mechanical parts of the system (e.g., engine, throttle position, brakes, security belts).

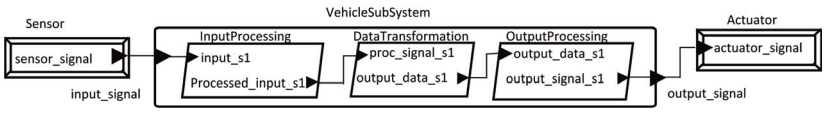


Fig. 3. Inner structure of the Product Line Architecture for the VCS subsystems

5.1 Domain Engineering

In the domain engineering phase, the domain expert must identify the quality attributes and the architectural transformation patterns that will allow to achieve the required quality attribute levels when this cannot be achieved by the product line variation mechanisms. The product architecture derivation process is driven by the key quality attributes. Safety, reliability, and performance are considered key quality attributes in real-time embedded systems for the automotive domain [20]. In this case study, we focus on the reliability sub-characteristic *fault tolerance* and the performance sub-characteristic *time behavior*. In particular, the time behavior sub-characteristic is broken down into *latency time*, which is the time elapsed between firing an input event, and obtaining a response from the system.

The product architecture derivation process will use the quality view of the multimodel to determine which architectural pattern must be applied to the product line architecture in order to achieve the desired quality attribute level. In particular, the architectural patterns that we analyze in this case study are: the homogeneous redundancy pattern and the triple modular redundancy pattern.

The *homogenous redundancy* pattern [8] (HR) improves the *reliability of the product* by offering multiple (in our case two identical) units of subsystem for monitoring and performing the same operations on the input signals. The disadvantages of this pattern are the recurring costs and a lack of coverage for systematic faults (if a fault systematically occurs when running the primary channel it can also occur when running the backup channel). The primary channel runs as long as there are no problems detected. In the case of a failure in the primary channel the system is able to detect the fault and switch to the backup channel and vice versa. There is no concurrency at run-time, only replication.

The *triple modular redundancy* pattern [8] (TMR) improves both the *reliability* and *safety* of a system by offering an odd number of channels operating in parallel. TMR negatively affects *performance*, since there are three subsystems running in parallel. The computational results or resulting actuation signals are compared, and if there is a disagreement, then the results with a two out of three majority win and are sent to the actuator. If the system suffers a failure affecting only one channel then the other two channels will continue to produce the correct result.

In order to define the corresponding relationships among architectural patterns and quality attributes, the domain expert must rank each pattern with regard to the Q quality attributes in a trade-off analysis using the AHP technique. For each quality attribute Q_a, s/he compares the N potential architectural patterns in a pairwise comparison. To determine how an architectural pattern Ax supports the quality attribute Q_a, in comparison to the pattern Ay, a weight is assigned (1 for equally important, 3 for moderately more important, 5 for strongly more important, 7 for very strongly more important, and 9 for extremely more important). For example, the domain expert defines that TMR is strongly more important (a weight of 5) than HR with regard to fault tolerance, and that HR is moderately more important (a weight of 3) than TMR with regard to latency.

The result of this comparison is an N x Q matrix that shows the relative support of the different architectural patterns to the quality attributes as shown in Table 1(a). Then, these values are normalized by applying the formula (1) to Table 1(a) to produce Table 1(b), and finally, the *impact* that an architectural pattern has on a quality attribute Q_a is calculated by applying the formula (2) to produce Table 1(c). This information is valid for all the products in the product line.

$$NormQ_y[i,j] = \frac{Q_a[i,j]}{\sum_{k=1}^{n} Q[k,j]} \qquad (1) \qquad I[i] = \frac{\sum_{k=1}^{n} NormQ_a[i,k]}{n} \qquad (2)$$

Table 1. Architectural Patterns and quality attributes trade-off analysis

(a)	Fault Tolerance		Latency	
	TMR	HR	TMR	HR
TMR	1	5	1	1/3
HR	1/5	1	3	1
Sum	1.2	6	4	1.3

(b)	Fault Tolerance		Latency	
	TMR	HR	TMR	HR
TMR	1 / 1.2	5 / 6	1 / 4	1/3 / 1.3
HR	1/5 / 1.2	1 / 6	3 / 4	1 / 1.3

(c)	Impacts	
	Fault Tolerance	Latency
TMR	0.83	0.24
HR	0.17	0.76

5.2 Application Engineering

In the application engineering phase, the application engineer introduces the quality attribute levels Q that the specific product must fulfill as normalized weights ranging from 0 to 1. For k quality attributes, the transformation process calculates the ranking R for each pattern j by applying the formula (3). For example, introducing a weight of 1 for *fault tolerance* and 0 for *latency* will make the transformation process to select the TMR pattern using the impact values in Table 1(c) (TMR: 1*0.83+0*0.24 > HR: 1*0.17 + 0* 0.76).

$$R_j = \sum_{i=0}^{k-1} Qi * Iij \qquad (3)$$

We have defined the model transformations to perform the product architecture derivation in QVT-Relations [16]. These transformations are not included in this paper due to space constraints. The transformations were applied to the product line architecture shown in Fig. 3 resulting in the product architecture shown in Fig. 4(a) or Fig. 4(b) depending whether the quality attribute *fault tolerance* or *latency* is selected.

After applying the architectural transformations we evaluate the derived product architecture in order to assess if the application of the architectural transformation pattern resulted in an improvement of the quality of the product architecture. This is done by comparing the results of the metrics measured over the product architecture derived directly from the product line architecture with the results of the metrics measured over the product architecture derived after applying the architectural pattern. As an example, we use the fault tolerance quality attribute to illustrate the product architecture evaluation. The fault tolerance quality attribute can be measured by applying the *key node safety* metric [15] over a fault

tree. The fault tree analysis is a top down approach to failure analysis, starting with a potential undesirable event called a *top* event, and thereafter determining all the ways in which this event can happen. The causes of a top event are connected by logic gates OR, AND or XOR. Fig. 5 (b) shows the fault tree associated with the *SystemFailure* event (the system gives no response due to sensor error, input processing error, or failure on the actuator).

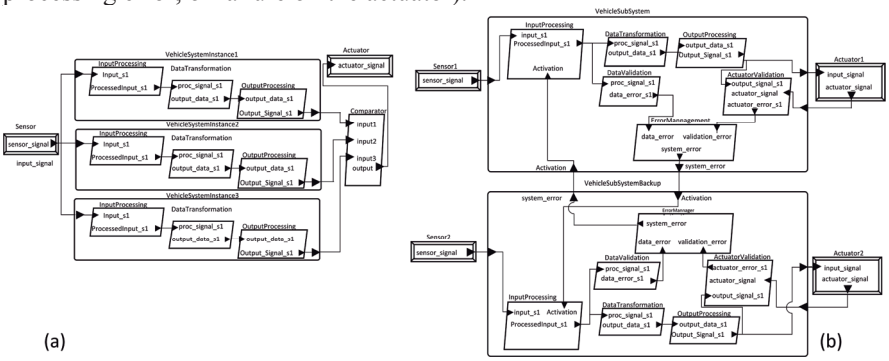


Fig. 4. Alternative Product Architectures prioritizing Fault Tolerance or Latency

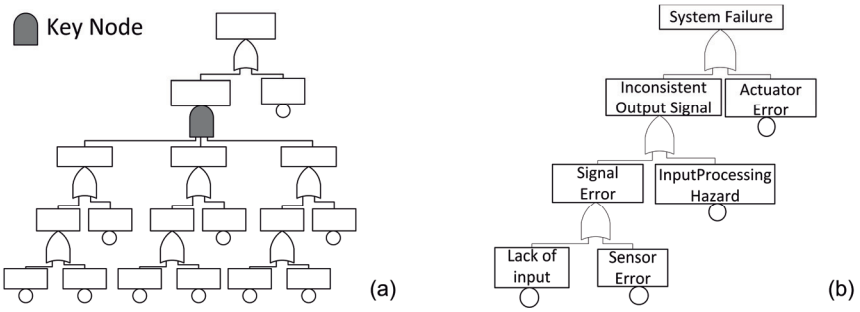


Fig. 5. Fault Tree for the Product Architecture *with* and *without* Applying the TMR Pattern

The *key node safety* metric provides a design tool for comparing a fault trees' fault tolerance without requiring *a priori* knowledge of the component reliability. This metric is based on identifying *key nodes* within a fault tree and considers the impact of these nodes on the safety of the system. A *key node* is a node in a fault tree that allows a failure to propagate towards the tree root if and only if multiple failure conditions exist in the node. Therefore, a *key node* can be identified in a fault tree as a node that connects two or more events by means of an AND or XOR gate. The lower bound of this metric is 0, and occurs when any single component failure in the fault tree causes the system to fail. The upper bound is open and occurs when the system fails if and only if every component fails and its value depends on the number of nodes in the fault tree. The value of the metric expresses how a change in the architecture improves its fault tolerance; the higher the upper bound is the better the fault tolerance the system has [15]. Fig. 5 (a) and Fig. 5 (b) show the fault tree including the *key node* information for the product architecture generated with and without the application of the TMR pattern, respectively.

The formula (4) operationalizes the *key node safety* metric. Table 2 shows the metric calculation for the product architecture with and without the application of the TMR pattern (TMR and Original). This metric shows that the application of the TMR pattern to the product architecture improves the fault tolerance of the product when compared with the values for the product architecture without the application of the TMR pattern (TMR: 0.069 > Original: 0).

$$S = \frac{kh'}{n^2} \sum_{i=0}^{k-1} \frac{c_i}{d'_i} \tag{4}$$

Table 2. Key Node Safety Metric Calculation

	Original	**TMR**
k: Number of key nodes in the fault tree	0	1
h': Total height of the fault tree +1	5	6
n: Total number of nodes in the fault tree	7	18
c_i: Number of nodes in the sub-tree rooted at key node k_i	0	15
d'_i: Depth of the sub-tree rooted at key node k_i +1	0	4
***S*: Key Node Safety Metric**	**0**	**0.069**

This case study has allowed us to illustrate the use of a multimodel to establish relationships among quality attributes and architectural transformation patterns. These relationships were used in a transformation process to determine which architectural pattern should be applied to the product architecture in order to meet specific quality requirements.

6 Conclusions and Future Work

This paper presented an approach for quality-driven product architecture derivation. The approach relies on the definition of a multimodel for representing the different software product line views (quality, functional and variability) and it is used to drive the derivation of product architectures from the product line architecture taking into account quality requirements. The feasibility of the approach has been illustrated through a case study on the automotive domain. This approach has three main benefits: i) the multimodel helps to have a richer semantic view of the product line by defining relationships among elements of its constituents views; ii) the architectural model transformations are driven by the relationships and the structural constraints established in the multimodel; iii) the quality view contributes to the quality improvement of the product architectures in the product line.

As future work, we plan to introduce the multimodel into the product configuration process and to select and deploy core assets over the product architecture taking into account functional and the quality requirements that the product must fulfill. We also plan to apply this approach in other domains where architectural patterns and quality attributes have been identified, to empirically validate the approach in practice, and to study trade-off and optimization techniques for improving the linear multi-objective AHP technique.

Acknowledgments This research is supported by the MULTIPLE project (MICINN TIN2009-13838), the ValI+D fellowship program (ACIF/2011/235) and the Universitat Politècnica de València PAID program (PAID-00-12).

References

1. Barkmeyer, E.J., Feeney, A.B., Denno, P., Flater, D.W., Libes, D.E., Steves, M.P., and Wallace, E.K. (2003) Concepts for Automating Systems Integration. NISTIR 6928, National Institute of Standards and Technology, U.S. Department of Commerce, USA
2. Botterweck, G., O'Brien, L., and Thiel, S. (2007) Model-driven derivation of product architectures. Proc. of the ASE 2007 Conference. ACM, New York, USA, pp. 469-472.
3. Bosch, J. (2000) Design and Use of Software Architectures. Adopting and Evolving Product-Line Approach. Addison-Wesley, Harlow.
4. Cabello, M. E., Ramos, I., Gómez, A., and Limón, R. (2009) Baseline- Oriented Modeling: An MDA Approach Based on Software Product Lines for the Expert Systems Development. Conf. on Intelligent Information and Database Systems, Dong hoi, Vietnam, pp. 208-213.
5. Czarnecki, K., and Kim, C.H. (2005) Cardinality-based feature modeling and constraints: A progress report. Proc. Int. Workshop on Software Factories, San Diego, USA.
6. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2010) Documenting Software Architectures: Views and Beyond, Addison-Wesley, Boston, USA.
7. Clements, P., and Northrop, L. (2007) Software Product Lines: Practices and Patterns, Addison-Wesley, Boston, USA.
8. Douglass, B. P. (2002) Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley, Boston, USA.
9. Duran-Limon, H. A., Castillo-Barrera, F. E., and Lopez-Herrejon, R.E. (2011) Towards an ontology-based approach for deriving product architectures. Proc. of SPLC 2011 Conference, Vol. 2, Article 19, 5 pages, Munich, Germany.
10. Feiler, P. H., Gluch, D. P., and Hudak, J. (2006) The Architecture Analysis & Design Language (AADL): An Introduction. Tech. Report CMU/SEI-2006-TN-011. SEI, CMU, USA.
11. Feiler, P.H., and Hansson, J. (2008) Flow Latency Analysis with the Architecture and Design Language. Tech. Report CMU/SEI-2007-TN-010. SEI, CMU, USA.
12. Gómez, A., and Ramos, I. (2010) Cardinality-based feature modeling and model-driven engineering: Fitting them together. Proc. of the VAMOS 2010 workshop, Linz, Austria.
13. Hudak, J., and Feiler, P. (2007) Developing AADL Models for Control Systems: A Practitioner's Guide. Tech. Report CMU/SEI-2007-TR-014, SEI, CMU, USA.
14. ISO/IEC 25010:2011 (2011) Systems and software engineering, Systems and software Quality Requirements and Evaluation (SQuaRE), System and software quality models.
15. Needham, D., and Jones, S. (2006) A Software Fault Tree Metric. In Proc. of. 22nd IEEE Int. Conference on Software Maintenance, Philadelphia, USA, pp. 401-410.
16. OMG (2008) Meta Object Facility 2.0 Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.0/PDF.
17. Perovich, D., Rossel, P.O., and Bastarrica, M.C. (2010) Feature model to product architectures: Applying MDE to Software Product Lines, Proc. of WICSA/ECSA 2010 Conference Helsinki, Finland, pp. 201-210.
18. Saaty, T.L. (1980) The Analytical Hierarchical Process. McGraw- Hill, New York.
19. Shiraishi, S. (2010) An AADL-Based Approach to Variability Modeling of Automotive Control Systems. Proc. of the MODELS conference Oslo, Norway, LNCS 6394, 346-360.
20. Thiel, S., and Hein, A. (2002) Modeling and using product line variability in automotive systems. IEEE Software, vol.19, no.4, pp. 66-72.
21. Zschaler, S., Sanchez, P., Santos, J., Alferez, M., Rashid, A., Fuentes, L., Moreira, A., Araujo, J., and Kulesza, U. (2009) VML* a family of languages for variability management in software product lines. In Proc. SLE 2009 Conference, Denver, USA.



PARTE IX

LOS AMIGOS

PARTE IX

Los Amigos

Lo que a continuación encontrará el lector es una serie de mensajes de afecto escritos por una parte muy pequeña de los muchos amigos que Isidro ha hecho a lo largo de su carrera. Los remitentes son, mayoritariamente, investigadores que han tenido una relación especial con Isidro, en particular dentro de la comunidad de Ingeniería del Software. También se incluyen mensajes de varios de sus estudiantes de doctorado, así como de amigos y colegas de otros ámbitos.

Estamos seguros de que los mensajes hubiesen sido muchos más si hubiésemos enviado la petición a más gente, por lo que pedimos disculpas a todos aquellos a los que, por una razón u otra, no se les ha solicitado.



Querido Isidro:

El capricho de los acontecimientos quiso que un buen día nuestros caminos se cruzaran y permitiera que, a tu lado como director y catedrático de nuestro querido DSIC, coincidiéramos, y que de esta forma, pudiéramos compartir un sinfín de momentos contigo.

Juntos llegamos a conformar un estupendo equipo tanto en el aspecto profesional como en el aspecto personal. Hemos salido de fiesta contigo; en multitud de ocasiones nos hemos quedados fascinados por tu brillante oratoria emanada de una vasta formación cultural y hemos llegado a asimilar en muchos aspectos tu apasionada defensa de la «excelencia» como brújula de la vida universitaria. A través de las magníficas descripciones de tus relatos (por supuesto, y como buen científico, acompañadas siempre de estupendos y valiosísimos documentos gráficos) hemos podido viajar en velero por las paradisíacas islas griegas, nos hemos permitido viajar y conocer culturas de países tan lejanos como China y Vietnam, nos hemos paseado por Central Park y Times Square en la cosmopolita Nueva York y tal vez, como punto culminante, hemos cenado ostras a la luz de las velas en la sin par «Coupole» parisina.

Y todas estas experiencias no podrían haber sido posibles si no nos hubieras permitido compartirlas contigo. Por todo esto, recibe un calurosísimo abrazo de tus amigos

Ana, Amparoles («tus chatis besonas») y de Tonín («el que te las birlaba»)

Hola Isidro,

Ha merecido la pena toda la dedicación y contribución a tu exitosa carrera profesional. Seguro que habrá muchos logros y reconocimientos en el camino. Tu energía positiva me ha inspirado a seguir y siempre vas a ser un ejemplo a seguir. Estoy orgullosa de haber trabajado contigo y de todos tus logros. Recuerdo el día en que me aceptaste en el grupo y confiaste en mí. Has sido mi mentor y guía, y espero aprender todavía más de tu gran experiencia.

¡¡¡A ver si pronto hacemos una carrera y me ganas!!!

Muchos besos

Nour Ali

Enhorabuena, Isidro, por toda una carrera de excelencia en la investigación y dedicación a la Universidad. Has sido y continúas siendo un ejemplo para todos. Un fuerte abrazo.

Gustavo Alonso

Querido Isidro, Me uno a este merecido homenaje, para manifestarte mi reconocimiento por tu apoyo en mis estudios de doctorado. Como parte de mi equipaje de recuerdos, llevo mis estancias en Valencia donde tu voz de aliento siempre me dio las fuerzas para continuar adelante. Gracias por permitirme ser parte de tu grupo, gracias por creer en mí. Además de tutor, fuiste un anfitrión excelente para conocer la cultura, la buena comida y la vida valenciana.

Que la Providencia siga guiando tu camino y que al finalizar tu jornada laboral tengas la paz interior y la satisfacción que viviste para servir.

Con todo cariño,



Raquel Anaya



De Isidro recibí lo más importante:

La seguridad de que era posible…

… ¡y lo fue!

Fernando Arango Isaza



He tingut el plaer de gaudir de la direcció de doctorat i de l'experiència d'Isidre Ramos i Salavert en el camp de l'enginyeria de programari. Em senc afortunat d'haver col·laborat amb ell en els projectes PRISMA I MOMENT.

Una aproximació formal a l'enginyeria de models de programari:

Una llesca vertical representativa de la carrera professional d'Isidre ha girat evolutivament al voltat de l'enginyeria de models de programari. En aquest context, l'invariant en el seu missatge didàctic, partint de reiterades manifestacions de la seua habilitat coordinadora durant aquests projectes, es pot generalitzar a l'aplicació de tècniques de verificació formal en l'enginyeria de programari. En concret, al modelat de sistemes d'informació, arquitectures de programari i desenvolupament de programari dirigit per models. El seu discurs integrador fou plasmat, juntament amb José Meseguer i José Á. Carsí, en la realització en Maude d'un marc formal de metamodelat emprant lògica equacional de pertinença, considerant estàndards de modelat i suport tecnològic en la plataforma Eclipse. En aquesta aproximació de metamodelat investigàrem un fonament conceptual, alineat amb els principis de la iniciativa Model-Driven Architecture, per a especificar llenguatges de modelat institucionals i per intercanviar metainformació en ecosistemes heterogenis de programari. Certament, aquesta línea d'investigació només ha constituït una xicoteta branca en la mencionada llesca sobre enginyeria de models de programari, que Isidre culmina com a *chair* de MODELS 2014.

Moltes gràcies a Isidre per tota la seua ajuda i suport durant la meuaetapa inicial com a jove investigador. Sempre romandrà una afectuosa amistat.

Artur Boronat



A l'amic que als anys 70, i quan jo treballava al Centre de Càlcul de la (llavors) UPB, va fer un curs a Barcelona junt amb en Claude Pair, a qui ell va convidar, i que em va obrir els ulls i la ment a noves perspectives.

A l'amic que a la dècada dels 80 va estar present al tribunals que em van fer progressar en la meva carrera universitària, quan em van fer doctor i quan em van fer catedràtic.

A l'amic amb qui he compartit algunes conspiracions positives, com un dinar, junt amb José Maria Troya, i quan en Miguel Toro preparava una «Jornada de Ingeniería de Software» a Sevilla, on vam acordar donar continuïtat anual a aquestes jornades, una de les llavors que van convergir en les JISBD.

A l'amic amb qui he compartit tants tribunals, i amb qui, junts, hem ajudat al progrés acadèmic de tants investigadors en l'àmbit de l'enginyeria del software, i amb qui he passat tantes bones estones a les ja feliçment consolidades JISBD.

A l'Isidre, a l'estimat amic, a l'excel·lent professor de la UPV i de tants altres, amb tota la meva admiració i respecte.

Pere Botella

Isidro:



Éste es un reconocimiento por tu gran aportación a la informática y a tu magnífica labor de tantos años que han dado su fruto. Aplaudiendo tu jubilación, tus hijos académicos nos sentimos orgullosos de ti y felices porque no hay alguien mejor que la merezca. Te reiteramos nuestro cariño, agradecimiento y admiración. En particular, siempre tendrás un lugar especial en mi corazón, que agradece tu constante apoyo, tus consejos, tus majestuosas ideas, tus valiosas enseñanzas y hasta tus exigencias que ayudaron a enriquecerme profesionalmente y a fomentar mi crecimiento personal.

¡¡¡¡¡Gracias maestro, gracias amigo!!!!!

María Eugenia Cabello

Recuerdo mis primeros contactos con Isidro cuando, siendo yo alumno de doctorado del programa del DSIC e Isidro Rector de la Universidad de Castilla la Mancha, empezó a impartir, el curso 86/87, la asignatura «Programación Lógica». El resultado del trabajo de la asignatura fue uno de los primeros artículos que publiqué junto con otros compañeros de departamento. También recuerdo una noche en Praga, durante una asistencia de ambos a un DEXA, buscando, ya casi desesperadamente, el «u fleku» y su «intensa» conversación en alemán con el taxista.

Estas anécdotas vienen bien para resaltar la que creo que ha sido, desde mi experiencia, una de las principales virtudes de Isidro en su actividad en el DSIC y por lo tanto en la Universitat Politècnica de Valènica, ya que ambas están vinculadas con su actividad investigadora y de formación de investigadores. Isidro es una persona con una trayectoria profesional influenciada por la investigación y esta motivación ha calado en su paso por el DSIC con su contribución al carácter investigador que caracteriza a este departamento, ya que favoreció sinergias que han influido en muchos de los que hemos desarrollado nuestra carrera investigadora en el seno del DSIC animándonos a sumergirnos en el inmenso océano de la investigación. Por ello mi más sincero reconocimiento.

Vicent Botti

Voy a utilizar este texto para presentar una perspectiva de Isidro Ramos Salavert distinta a la que otros posiblemente hayan realizado. Lo digo porque, a pesar de que Isidro fue mi profesor en la universidad, mi tutor en el programa de doctorado y mi cotutor de tesis doctoral, voy a centrarme en su actividad de gestor que ahora comprendo mejor, ya que yo ahora, como él entonces, soy el director del Departamento de Sistemas Informáticos y Computación de la Universtat Politècnica de València.

Isidro empezó a colaborar en actividades en nuestro departamento en el curso 1986-1987, cuando aún era Rector en funciones de la Universidad de Castilla—La Mancha. En ese periodo, nuestro departamento está naciendo y formándose, siendo director del mismo José Llorens Sánchez. Cuando se puso en marcha la nueva LRU, hubo elecciones a director, e Isidro, a pesar de tener intenciones de presentarse, no pudo hacerlo por no haber terminado su actividad como rector, impidiéndole esta circunstancia realizar una comisión de servicios a nuestra universidad. De manera provisional accedió al cargo de director Vicente Hernández García. Fue el 6 de octubre de 1988 cuando ya pudo ser votado y elegido director de nuestro departamento. Como anécdota, apuntar que el resultado de la votación fue de 34 votos a favor, 10 abstenciones y 1 en contra. El equipo directivo que le arropaba estaba formado por Pedro Blesa Pons como subdirector docente, Vicente Hernández García como subdirector de investigación, Francisco Casacuberta Nolla como secretario, Matilde Celma Giménez como encargada de la biblioteca y José Miguel Benedí Ruíz como subdirector de infraestructuras. Voy a transgredir el secreto de voto… yo no fui el que votó en contra.

Su periodo de gobierno en el departamento duró hasta 1994, ya que el 29 de octubre de 1992 se presentó a una reelección. Fueron años de crecimiento, de orientar la actividad académica de un equipo de profesores muy jóvenes, de luchar por un reconocimiento de la Informática en una universidad donde era una novedad, desde el punto de vista docente e investigador. Años donde pasamos de estar ubicados en distintos barracones, a tener una sede, el D3, y en años siguientes, a tener una planta del edificio de la Escuela Universitaria de Informática. Años convulsos, en los que el impulso de la investigación como parte sustancial de la actividad académica llevó a situaciones problemáticas (como muestra, solo incluyo las fechas de los consejos de departamento en los que se trataron estos temas en el año 1993: 29 de abril, 6, 27 y 31 de mayo, 3 y 9 de junio. Supongo que Isidro los tendrá en la memoria). El 23 de septiembre de 1994, presenta su dimisión como director de departamento.

Visto con la perspectiva de los años y de la experiencia desde los cargos de gestión, he de destacar de esta época de Isidro su dedicación, esfuerzo y empeño en impulsar una línea de trabajo. Aquellas semillas dieron sus frutos y hoy nuestro departamento es uno de los más relevantes en la universidad y fuera de ella, tanto en su actividad docente como investigadora. Isidro puso un gran grano de arena que con estas palabras quiero agradecer.

Juan Carlos Casamayor Ródenas



Querido Isidro,

Permíteme ir un paso más allá de lo que han sido todos aquellos momentos compartidos en el grupo ISSI, por lo que estoy ya muy agradecido, y permíteme centrarme en un episodio que creo que coincidirás conmigo en que es también de gran relevancia.

Recuerdo aquellas primeras clases sobre Ingeniería de Requisitos e Ingeniería del Soft-

ware en lo que antes era la Facultad de Informática en la UPV. Recuerdo aquellos primeros momentos en los que transmitías la idea de que los informáticos debíamos ser ingenieros, y que para ello teníamos que hacer ingeniería. Aquello caló, puedo asegurar. No solo en mí, sino en muchos de mis compañeros de promoción. Por fin uno de nuestros profesores, tras varios años en la facultad, nos decía algo así.

También puedo recordar el caso de la cuenta bancaria, la noción de taxonomía, de reificación, las valuaciones, las descripciones lógicas, la estructura de Kripke… Nadie antes se había atrevido a descubrirme el maravilloso mundo que realmente hay detrás de nuestro trabajo. Desde luego, eso no dejaba a nadie indiferente. Sin duda alguna, cualquiera de tus exalumnos tendrá también un nítido recuerdo de lo que supuso un punto de inflexión en nuestra carrera y en nuestra percepción acerca de la profesión. Pasábamos a ser (futuribles) Ingenieros en Informática.

Quisiera expresar en estas líneas mi agradecimiento por el valor añadido aportado a mi formación. Como ves, esos recuerdos perduran, y forman parte también del legado y la impronta que has dejado en mí. Cada momento que sigo compartiendo contigo en cualquier reunión o encuentro informal, siento que sigo aprendiendo, y eso es algo tan grande por lo que te estoy muy agradecido. Muchas gracias por compartir y seguir compartiendo tus experiencias conmigo.

Alejandro Catalá

Isidro, no quería dejar pasar esta oportunidad en la que poder dedicarte unas palabras de reconocimiento a tu labor, y sobre todo, agradecimiento. Agradecimiento porque fuiste uno de los referentes que llamó mi atención y logró captarme para incorporarme al grupo ISSI (lejos quedan las clases de TSA donde contabas a alumnos de 5º cosas extrañas de investigación en un lenguaje que no comprendían ¿ontologías? ¿reflexión, reificación? ¿transformaciones de modelos?). Agradecimiento también porque, a lo largo de los años, me has transmitido el espíritu de la investigación, del trabajo duro, y la capacidad de desafiarse a uno mismo en busca de nuevos retos y reinventarse continuamente.

También agradecerte el que siempre estuvieras ahí apoyando al grupo en momentos clave o de representación (recuerdo mis primeras JISBD, en las que ante una pregunta complicada, saliste en mi ayuda para explicar una terminología que aún no dominaba).

Por último, agradecerte el que hayas estado presente en tantos momentos clave de mi vida: el día en que despedimos a mi padre (fue un gran apoyo moral para mí), mi boda (uno de los días más felices), y presidir mi tesis (ya ves, incluso te pedí que me hicieras más preguntas —quería saber tu parecer ;-)—, tras las casi 1h30 de preguntas de los otros miembros del tribunal).

En definitiva, ha sido un honor haber formado parte de la tripulación del ISSI que capitaneabas, etapa en la cual adquirí no sólo conocimientos, sino capacidades que han sido muy relevantes para mi carrera profesional.

Un abrazo,

Cristóbal Costa Soria

Vaig conèixer a Isidro al començament de la dècada dels 80's, quan ell acabava de moure's de la Universitat del País Basc a la Universitat de València i jo acabava d'aterrar a Barcelona després d'una llarga estada en cultures i conceptes universitaris molt diferents. Sense l'ajut de l'Isidro per a navegar en el «peculiar» sistema universitari espanyol, possiblement el meu pas per la universitat espanyola no hagués estat massa llarg. Per tant, no crec que la meva opinió sobre l'Isidro pugui qualificar-se com objectiva.

Dit això, des de la meva perspectiva, Isidro ha estat una de les influències més importants en el desenvolupament de la informàtica a Espanya. Físic de formacio, va ser un dels primers a veure la importància de la informática com a disciplina pròpia i al llarg de la seva trajectòria ha deixat la seva empremta por a on ha passat: Centre de Càlcul de la Universitat Complutense de Madrid, Universitat del País Basc, Centre de Càlcul de la Universitat de València, Universitat Politècnica de València, Universitat de Castilla—La Mancha, on va engegar la universitat com a Rector, i on continua com a «Rector Honorario». Una part important d'aquesta empremta ha estat plantar la llavor de grups de treball en diferent disciplines de la informàtica. Dues àrees de la informàtica on la tasca d'Isidro ha fructificat són la Concurrència i l'Enginyeria de Sistemes. Tota la tasca realitzada per Isidro es basa en una combinació d'una bona visió de futur amb un caràcter fort per a defensar el que ell creu que s'ha de fer (aquesta darrera característica pot ser un tret genètic en alguns valencians). Aquest fort caràcter fa que Isidro no deixe indiferent ningú, aixecant grans estimes entre la gent que el coneixem, però també algunes animositats.



Isidro, gràcies per tot el que has fet, i pel que encara faràs!

Josep Díaz

Isidro es un exponente señero de la mejor academia de la España pre-transición. Como a tanta gente de su generación, a Isidro le ha tocado abrir camino. Estudió en Paris cuando la gente se iba a Francia no precisamente a hacer la tesis. Estuvo en Bilbao, en los inicios de la Universidad del País Vasco. Tal vez de aquella época le venga el carácter resuelto y «tirao-pa-alante» que se suele asociar con los habitantes de aquellas tierras. También paso por Castilla—La Mancha, de cuya Universidad fue su primer Rector. Y regresó a su tierra, Valencia, donde también le tocó liderar los cambios. Isidro desbrozó caminos que luego nos han facilitado la vida a otros. Pero como todo explorador sabe, la labor de desbroce requiere de un buen machete. E Isidro tenía el suyo. Bien afilado. Le he escuchado en muchas intervenciones. Algunas públicas, en mesas redondas. Otras, más privadas, alrededor de un mantel. En todas ellas, Isidro solía poner las puntos sobre las íes. También en las íes mayúsculas… Sus análisis solían estar desprovistos del

buenismo y obsequiosidad que suele caracterizar a mucho discurso académico. Si no estoy equivocado, suyo es el sustantivo «meninismo» para reflejar ciertas prácticas universitarias. Sólo por eso, merecería un sillón en la RAE… bueno, dejémoslo en un taburete. Sus logros no se limitan al apartado investigador sino especialmente, al haber sido una voz valiente, a veces carrasposa, pero en cualquier caso comprometida con la Universidad en su mejor sentido.

Óscar Díaz



Isidro forma ya parte de la historia de la Informática en España. Fue uno de los pioneros en este campo en España, y a día de hoy aún tiene fuerzas para seguir en activo.

Si hay algo que caracteriza a Isidro por encima de todo es la solemnidad. No he conocido a ninguna otra persona que se exprese de forma tan solemne, sea cual sea el tema del que habla o el contexto en el que lo hace. Por supuesto, como cabe esperar en un personaje como él, también ha hecho acopio de una gran experiencia y saber hacer, lo cual ha sido de gran ayuda a los que hemos trabajado con él en innumerables ocasiones. Por ejemplo, de entre las actividades en las que he tenido el honor de colaborar con Isidro, su papel fue crucial tanto en la puesta en marcha de los estudios de doctorado en informática y de la escuela de verano en la Universidad de Castilla—La Mancha, como en la organización de la tercera edición del Congreso Español de Informática (CEDI), del que fue su Presidente.

De entre sus muchas frases para la historia, me quedo con esta: «la investigación no es cuestión de democracia», en la que pone de manifiesto su acertada forma de pensar, según la cual el conocimiento y saber hacer en investigación deben prevalecer sobre la opinión de la mayoría.

Y de entre sus virtudes, me quedo con su enorme capacidad de trabajo, que lejos de apoltronarse en su cátedra le ha llevado a proyectos más ambiciosos y con mejores resultados con el paso de los años.

José Duato



Se olvidan con rapidez los malos o los mediocres profesores que se limitan a transmitir informaciones, pero se guarda enorme gratitud a aquellos que nos enseñaron a comprender las materias, a asimilar conocimientos, a utilizarlos y a enriquecerlos mediante el razonamiento y la experimentación. En aquellos días donde la informática intentaba abrirse camino como enseñanza universitaria, un Isidro Ramos barbado nos introdujo en el mundo de los procesadores de lenguajes. Con él aprendimos la relación estrecha entre teoría y práctica y la idea de que la programación es algo más que poner una instrucción detrás de otra. Clases amenas, rigor matemático, implementación de los formalismos matemáticos para construir lenguajes de programación, e iniciación en la publicación de los resultados obtenidos. Esta es la imagen que emerge de aquella universidad de los 70 ocupada por la policía, con huelgas continuas y con poco espacio para la concentración académica. Isidro irradiaba —y lo sigue haciendo— saber hacer, pedagogía, profesionalidad, buen trato con los alumnos y excelentes vibraciones que nos contagiaron curiosidad, inquietud y entusiasmo para continuar los estudios más allá de los Pi-

rineos. Desde esa época de estudiante siempre he admirado de Isidro su iniciativa para formar nuevos grupos, su capacidad de trabajo, su generosidad y su apoyo a colaboradores y amigos, su honestidad y su sinceridad para señalar verdades incomodas. Y porque he sido beneficiario de todas ellas, es un placer poder expresar en este reconocimiento al verdadero impacto de Isidro, lo que siempre hemos pensado pero que por unas razones u otras nunca le hemos dicho.

Francisco Garijo



Con estas líneas quiero en primer lugar darte las gracias por todos tus sabios consejos y por todas esas discusiones en las que nos mostrabas ese otro punto de vista, ese que se basa en ver el mundo, la ciencia y la informática con los ojos de un físico. Por esas largas y constructivas discusiones cuando presentaba primero mi borrador de Proyecto Fin de Carrera, posteriormente, la que sería mi tesina de master y más tarde los contenidos de mi tesis. Esa tesis en las que recogía, junto a mis directores Silvia y Emilio, el guante que en su día arrojaste en forma de retos pendientes de solución en el ámbito del proyecto MULTIPLE.

Para mí fue un regalo, un orgullo que fueses el presidente de mi tribunal y fueses partícipe del final de ese largo camino (y el principio de otro) en el que has participado en primera persona dejando la impronta imborrable propia de los grandes maestros, aquellos que uno recuerda durante toda la vida. Gracias también por ayudarnos a darle a nuestro trabajo ese cariz formal que a veces perdemos de vista y al mismo tiempo recordarnos que no debemos perder el rumbo de lo práctico y de lo aplicado.

Me ha fascinado durante todo este tiempo la capacidad de escuchar sin prejuicios, de analizar, de rebatir aquellas ideas con las que no estas conforme, de mantener la curiosidad como el primer día y de seguir involucrado en el grupo y en la comunidad científica.

Por último, no me gustaría que estas líneas se quedasen en la faceta académica o profesional. En estos casi cinco años me ha dado tiempo a conocerte en un ámbito más personal. Fuiste profesor mío hace muchísimo tiempo y en aquella época los alumnos te veíamos como un profesor serio, estricto y con un discurso muy elaborado. Al integrarme en el grupo, conocí esa otra faceta, la de marino, viajante y enamorado de la historia y de la ciencia en todas sus formas. En esas otras charlas informales en las que discutimos de mil y un temas y en las que uno llega a la conclusión de que al final, como buen maestro, siempre tienes algo que enseñarnos.

Javier González Huerta



Apreciado Isidro,

Parece que fue ayer cuando nos conocimos por primera vez en un tribunal de tesis de la Universidad Politécnica de Madrid y ya han pasado cerca de 20 años. Un admirado profesor que tuve en la Facultad de Matemáticas decía siempre «Es de bien nacido ser agradecido» y, no quiero dejar pasar esta oportunidad para agradecer tu esfuerzo y dedicación a la comunidad de Ingeniería del Software durante todos estos años.

Cicerón decía «No basta con adquirir sabiduría; es preciso, además, saber usarla» y,

en este sentido, creo que formas parte de una generación ilustre, con visión de futuro, y que ha contribuido firmemente al desarrollo de la Informática en general y a la Ingeniería del Software en particular. Habéis sabido usar vuestra sabiduría y conocimiento para impulsar nuestra disciplina no sólo en España sino también a nivel internacional.

Tu trayectoria y tu buen hacer siempre han sido un referente para muchos de nosotros.

Con todo mi respeto y admiración.

Un fuerte abrazo,

Juan Hernández



Más que un Profesor.

En 1991 me encontraba trabajando en la Universidad Nacional de Colombia, en su sede de la Ciudad de Medellín, en una categoría fuera de concurso público, denominada Profesor Especial. Por esta razón me solicitaron que tomara el liderazgo en un proyecto que COLCIENCIAS (Departamento Administrativo de Ciencia, Tecnología e Innovación de Colombia) le había asignado a la Universidad. Un proyecto financiado por la UNESCO a través del CREI (Centro Regional para la Enseñanza de la Informática), dentro de las efemérides del Encuentro de Los Dos Mundos o quinientos años del descubrimiento de América por los europeos. En los rubros presupuestados en ese proyecto se tenía la movilización de académicos de España y Colombia. El proyecto consistía en estudiar la emigración de software de tercera generación a software de cuarta generación buscando una manera de transferir aplicativos en software tipo espagueti a software orientado a objetos y de paso proponer una solución a la problemática que se avecinaba en el manejo de fechas por el cambio de siglo, conocida como la crisis Y2K. Le solicité al CREI asesoría sobre alguna universidad española para hacer contacto y la respuesta fue «la persona más indicada sobre el tema es el profesor Isidro Ramos, de la Universidad Politécnica de Valencia». Así, comenzó mi relación con Isidro.

Luego de los trámites de rigor se consolidó el proyecto y el profesor Isidro Ramos Salavert viajó a Medellín, Colombia, junto con otros profesores de la Universidad en diferentes momentos. En contrapartida viajé con otro compañero a Valencia, España. El proyecto fue un éxito, se produjeron varios reportes técnicos, se desarrollaron herramientas CASE y se abrió paso para otros proyectos.

Existen impresiones interesantes sobre el paso de Isidro por Colombia, de un «señor tan serio» como él parecía. Caminando por aceras de un barrio de Envigado, de repente comentó que aquí debía haber mucho dinero disponible pues las aceras estaban hechas en cerámica. Se refería a que estaba arreglando el balcón de su casa en Moraira y era muy costoso hacer allí lo que él veía aquí en la calle. En alguna ocasión solicitó que le llevaran a un mercado popular, no a uno de almacenes de grandes superficies, porque según él es una manera de conocer la cultura de la comunidad, difícil demanda que fue resuelta a satisfacción. Tuve la oportunidad de ver con Isidro el partido España—Colombia en el mundial de futbol del momento, una rareza que vale mencionar. En otra de sus visitas, estábamos mostrándole hacia arriba en la montaña una raya blanca, la cárcel en donde se encontraba el tristemente célebre personaje Pablo Escobar, y ninguno sabíamos que en ese momento dicho personaje estaba escapándose de ese recinto. Grande fue nuestra sorpresa por el enorme despliegue periodístico mundial que encontramos al siguiente día, ya que nosotros estuvimos cerca del citado lugar y no vimos tanto movimiento

policial. Todas estas anécdotas dieron pie a que se generara una relación personal más allá de lo académico. Hay un sinnúmero más de cosas simpáticas que contar.

Por otra parte, en Valencia siempre fuimos acogidos de muy buena manera por Isidro, y aunque él intentaba no dejar entreverlo, notábamos su preocupación por que estuviéramos bien atendidos, ubicados en un espacio de trabajo adecuado y no quedáramos relegados por los miembros de su equipo luego de las jornadas de trabajo. Isidro, con generosidad, no sólo me abrió la puerta de su acervo académico sino también de su vida personal. Departimos en diferentes encuentros, agradables momentos en su casa, junto con su esposa Eulalia y su hija Sofía, también en restaurantes y esporádicamente algunas copas. Llegó incluso a la complicidad del colegaje al prestar su barco para la boda de otro compañero. Por supuesto, fueron frecuentes los paseos en su auto por los alrededores de Valencia. Cuento esto porque para otras personas debe ser muy difícil imaginarse a ese «señor tan serio» en una faceta tan amable.

En la relación profesional surgieron otros proyectos entre los cuales se destaca un convenio de formación doctoral en Medellín entre la Universidad Politécnica de Valencia y la Universidad Nacional de Colombia, que luego se extendió a todo el país. En este proyecto varios profesores de la Politécnica impartieron docencia presencial en Medellín, se podía observar una rivalidad (sana) profesional entre ellos, pero siempre triunfaba la jerarquía de Isidro. En formalización de software, en orientación a objetos, en elicitación de requisitos, no tenía parangón. Isidro era la autoridad, sí tuvieron razón los funcionarios del CREI cuando nos recomendaron acercarnos a él. Este proyecto produjo varios doctores, experiencia en manejo de grupos de investigación, otros proyectos de investigación y producción de publicaciones científicas.

Puedo decir, con mucho gusto, que con el profesor Isidro Ramos Salavert se formó una sólida relación académica y profesional, pero mejor aún, una sincera y cercana amistad, que sigue vigente. Finalmente, solo puedo agradecer a Isidro por todas sus deferencias conmigo y con Colombia.

Jesús Antonio Hernández Riveros

Kaixo Isidro:

Una vez leí la siguiente frase « Tu trabajo no se ocupará de ti cuando estés enfermo. Tus amigos sí lo harán. Mantente en contacto con ellos».

Eso es lo que quiero decir, que sigas manteniendo contacto con nosotros.

Un abrazo

Arantza Illarramendi

Isidro, quiero resaltar en estas líneas cómo las cosas buenas vienen acompañadas por otras mejores, y éste ha sido el caso de mi experiencia de formación como investigador que tuve contigo. Desde tu primer contacto en México en el 2001, sentí la buenas vibras que trasmitías en tu participación sobre tu brillante idea sobre los meta-modelos, y digo brillante porque quién iba a decir que sería uno de los tópicos relevantes enarbolados por la OMG, concretamente sobre el MDA, siendo que tú fuiste quién inició con este trabajo. Aún recuerdo tu propuesta que me hiciste en el 2002 sobre el compilador de modelos, pero mi nula experiencia en la investiga-

ción en ese entonces y como suele pasar con lo novedoso, no alcancé a percibir la trascendencia de tu propuesta.

Luego de mi breve estancia por la UPV en el 2003 logré constatar tu liderazgo no sólo en el grupo que encabezabas sino más allá del DSIC y de la UPV, y fui testigo de los trabajos que impulsabas directamente con Jennifer y Artur, co-creadores contigo de tus excelentes trabajos plasmados en sus tesis doctorales: PRISMA y MOMENT.

Después, en mi estancia del 2005 al 2008 en el DSIC, no sólo fui testigo de tus logros sino que me diste la oportunidad de colaborar con mi grano de arena en parte de tus proyectos. Un mundo de trabajo excitante como exigente pero gracias a esa presión y sobre todo a tu apoyo y al del excelente equipo de trabajo con que has contado pude ver culminado con el objetivo de mi vida profesional que anhelaba.

Mas como mencioné al principio, las cosas buenas vienen con otras aún mejores, así que lo mejor que llegó con mi logro profesional fue el haber conseguido tu amistad, pues si en el «aspecto» profesional tu trabajo habla por sí mismo, este «aspecto» no brillaría tanto si no fuera por la calidad humana que te caracteriza y que transmites a tus colaboradores.

Así que Isidro, gracias por forjarme como investigador y brindarme lo más valioso de ti, tu experiencia y amistad.

Con profundo afecto y admiración

Rogelio Limón

Para Isidro Ramos

Estos pensamientos van dedicados al gran maestro Isidro Ramos, formador de formadores y maestro de maestros.

Isidro ha sido uno de los grandes impulsores de la Informática (Computación, en México) en el ámbito de Valencia en particular, y de España en general, en momentos en los cuales el arranque, la dirección y la inteligencia fueron vitales para lograr implantar esta disciplina como una nueva ciencia con vida y voz propia dentro del escenario de las escuelas universitarias profesionales, desligándola de ser dependiente de otras disciplinas como las matemáticas o la ingeniería eléctrica/electrónica.

Isidro formó parte de ese grupo de gente intelectual, con estudios de doctorado, que creó la disciplina de la Informática en un ámbito político, social, económico y educativo, hasta cierto punto, adverso, que ponía trabas para cualquier proyecto que implicará innovación, desarrollo y creación tecnológicos en el entorno gubernamental, y que requirió de un gran sacrificio, dedicación e ingenio para llevar a cabo el desarrollo de esta nueva ciencia en España, la cual ahora se ve como algo natural en nuestros días.

Formó profesionales primero, y luego doctores en esta área, y dirigió un sinnúmero de proyectos de investigación en Informática que permitieron abrir paso a la disciplina dentro del espectro de áreas que copaban en esos tiempos los apoyos de investigación dados por el gobierno. Como persona cosechó innumerables amigos, de todo tipo y clase sociales, siendo muy apreciado y valorado en todos los ámbitos en que se desenvolvió.



Vaya mi reconocimiento y agradecimiento a este gran hombre que supo estar en el momento oportuno, y aprovecharlo, para ser parte de los fundadores de una disciplina que hoy es una de las fundamentales en nuestras vidas.

Muchas gracias por todo, profesor y amigo Isidro Ramos.

Felipe López Gamino

Isidro ha sido para mí el maestro que me ha ido guiando desde que empezó mi aventura universitaria como informático.

Acabé la licenciatura de Física en un momento que había un porvenir más bien triste, lo único a lo que se podía aspirar era a ser profesor de física y química o de matemáticas en un Instituto, ya que la puerta de la Universidad estaba cerrada, con un montón de los llamados PNNs esperando a ver como se resolvía su situación, estoy hablando del final de los 70s.

Las dos puertas más interesantes que se me abrieron entonces fueron la del Instituto Astrofísico de Canarias y la de Telefónica, pero en ambas había que saber eso de programar, y solo habíamos visto en menos de un mes en la carrera la sisáis del FORTRAN y sin probar un solo programa en un ordenador de esos en los que había que meter con tarjetas los datos y programas. Entonces apareció por primera vez Isidro para orientarme, en la Universidad de Valencia había aparecido un joven catedrático que había montado una asignatura en cuarto de Física y reorganizado el CPD donde daban unos cursos de posgrado en Informática donde competíamos con los alumnos de física en los terminales de un Eclipse para hacer prácticas, eso fue en 1983. Allí conocí como profesores a los que luego serían compañeros en el DSIC, Paco, Enrique, Arturo, Emilio, José Miguel, y como alumnos con los que me peleaba en los terminales a EncarNa, Mabel, entre otros con los que tres años después coincido en el DSIC. Antes de acabar aquello ya estaba trabajando.

Poco años después entré en el DSIC, y de nuevo apareció Isidro introduciéndonos en el estudio de las ideas de prototipación, de la base matemática que había de detrás de las abstracciones que han permitido crear el paradigma de la orientación a objetos, con todas las posibili-

dades de investigación que nos aportaba. También nos enseñó a unos físicos a movernos en el para nosotros complicado mundo de los ingenieros de la Politécnica, con su experiencia como rector de la Universidad de Castilla—La Mancha.

Cuando nos integramos en la Universidad de Alicante nos ayudó en los primeros pasos con el rector de entonces, Martín Mateo, con el que había coincido en el País Vasco, y en las reuniones de rectores. Y pese a la distancia siempre pude tener sus buenos y sabios consejos, cuando empezamos con e-learning, con las bibliotecas digitales que con Hilario también estaban trabajando dándonos interesantes contactos como Nieves Brisaboa o Ricardo Baeza Yates, para aquella aventura que fue poner en marcha la biblioteca virtual Miguel de Cervantes. Cuando empecé mi otra aventura, la empresarial, nuevamente me animó y me hizo ver lo importante que era para la dimensión académica.

Por último cuando de nuevo volví a la Universidad a tiempo completo, su estancia durante unos meses en nuestro departamento, me sirvió para por fin cerrar mi doctorado que había dejado de priorizar por los mil líos en los que he estado metido en estos anteriores 25 años.

Por eso te recuerdo Isidro, como maestro, en este tremendo aprendizaje que hemos tenido muchos en el reciclaje a informáticos, a investigadores en este nuevo campo, donde casi todo ha sido autoaprendizaje, has sido la persona que más me ha orientado en este complejo proceso.

¡Muchas gracias! y gracias por seguir en la brecha.

Manolo Marco

Unas líneas para Isidro Ramos… Isidro, ¿cómo describirte? Los inicios de la Ingeniería Informática en España; un pilar fuerte sobre el que se ha construido la comunidad de Ingeniería del Software en España; una persona enérgica, capaz de mover e integrar, inteligente, capaz de crear y construir, generoso, capaz de enseñar y de dar, humano, capaz de escuchar y empatizar…

En los inicios de mi carrera universitaria, Isidro, viniste a dar una charla a la Universidad Carlos III, en la que yo comenzaba mis estudios de doctorado. Esa tarde, cuando te lleve al hotel, me invitaste a acompañarte a cenar, ¿recuerdas? Yo no he olvidado esa tarde. No solo por la oportunidad que era, para una estudiante de doctorado, cenar con «Isidro Ramos», sino también por lo mucho que aprendí ese día hablando contigo. En el café Comercial, me contaste tus años en Madrid y un montón de anécdotas sin desperdicio. Fue, en verdad, una tarde entrañable desde la que te recuerdo con admiración y cariño.

Desde entonces, siempre has estado ahí; consejero, mentor, amigo… Ha sido un placer cada encuentro y cada conversación contigo. Esto no es una despedida, porque sigues estando y estarás muchos años con nosotros. Sólo es un buen momento para agradecerte de corazón todo el apoyo y las enseñanzas que nos has trasmitido y para expresar lo que el día a día a veces no nos permite: la admiración, el respeto y el cariño que siento por ti.

Esperanza Marcos «Cuca»

Cuando mi directora de tesis enumeró los miembros del tribunal de tesis doctoral que había elegido y quién sería el presidente del mismo, a mí no me pareció tan buena idea como a ella. El tal Isidro Ramos tenía fama de ser un Catedrático de los de antes, esculpido en mármol, aunque reconozco que hablaba desde la ignorancia pues no recordaba haber coincidido con él. Los mitos y leyendas que llegaban a un doctorando como yo hablaban de alguien que desde el tiempo de los dinosaurios había sentado buena parte de las bases de la Informática en la Universidad Española, y era fácil imaginar que así habría llegado a coleccionar un buen puñado tanto de amigos como de enemigos por todo el país. Se decía de él que hablaba muy clarito y que cuando algo le gustaba, lo decía… y cuando no le gustaba, también (y esto último no me aportaba mucha tranquilidad, la verdad). Pero bueno, si mi directora pensaba que si uno tenía un buen trabajo era recomendable que Isidro lo conociera de primera mano, pues así lo haríamos y ya veríamos que pasaría.

Bueno, y lo que pasó fue que los mitos y leyendas no estaban muy alejados de la realidad. Sin embargo, esa característica de Isidro de decir lo que piensa (y hay que reconocer que cuando algo no le gusta, lo dice con muchísimo tacto… si uno es merecedor de tales favores) llega a límites insospechados. Quiso la providencia que nuestro trabajo le gustara (los infiernos no se abrieron para tragarse a este pobre doctorando durante la defensa de la tesis) y el otrora temido Isidro se convirtió en un inesperado heraldo de nuestro trabajo a lo largo y ancho del país. Nosotros, que nos habíamos hecho ver más en el extranjero que localmente, nos encontramos con que en poco tiempo varios grupos nacionales nos invitaban a dar charlas sobre nuestra investigación. Varias veces le he manifestado a Isidro nuestro agradecimiento pero seguro que han sido pocas pues tiene buena parte de la culpa de que hayamos establecido relación con varios grupos nacionales con líneas de investigación similares.

A partir de ahí comienza una amistad no sólo con él sino con miembros de su grupo de investigación, con los que colaboro regularmente a nivel docente, investigador… y gastronómico. Pero no puedo acabar este agradecimiento hacia Isidro sin contar una de mis anécdotas favoritas sobre él. Tras acabar felizmente el acto de defensa de la tesis, antes de comer, salimos a pasear por Zaragoza para mostrarles a los miembros del tribunal las maravillas que dicha ciudad posee. Sin embargo fue Isidro quien nos hizo de (excelente) guía turístico, y nos explicó a los locales las más notables características arquitectónicas de La Basílica del Pilar, la Seo, los restos romanos, y de cada uno de los monumentos zaragozanos que encontramos, lo cual nos hizo sospechar si no habría sido testigo directo de la construcción de todos ellos, pues la profusión de detalles desconocidos para los comunes mortales era enorme. Pero la guinda del pastel estaba aún por llegar. Cuando entramos al restaurante elegido para comer, situado en la Plaza de Aragón, Isidro nada más cruzar la puerta pregunta a quien nos recibió: «Perdone si me equivoco pero este edificio es de estilo pompeyano, ¿verdad?». Un silencio de varios segundos nos envolvió a todos, especialmente al nuevo Doctor («¿¿ha dicho pompeyano??»), hasta que la señora respondió sorprendida: «Pues sí, está en lo cierto, aunque la verdad es que hasta ahora nadie nos lo había dicho». Ése es Isidro. Espero que nunca cambies, un abrazo.

Eduardo Mena

Estimado compañero y amigo Isidro:

Aún recuerdo el primer contacto que tuvimos cuando me presenté a una oposición en el Centro de Cálculo de la Diputación de Valencia. Tú estabas como miembro del tribunal, había superado la primera prueba pero a la segunda no puede presentarme porque me coincidía con algún examen o algo así. El caso es que cuando fui a justificarlo ahí estabas tú, hablamos de ello y la casualidad hizo que nos encontráramos, de nuevo ese mismo día, en la estación esperando el tren en la «Fonteta de San Lluís». El retraso habitual de los trenes que viajaban de noche nos proporcionó un largo y grato rato de charla.

¿Quién iba a imaginar que volveríamos a vernos en el *poli*? ¿Que tú serías mi director de tesis doctoral? ¡Cosas del destino! A lo largo de estos años han sido muchos los momentos compartidos en adorable compañía, el fantástico viaje a Berlín, ¡Ay! ¡Qué buenos recuerdos! Y tú siempre ahí, con ese buen hacer, ese saber estar, ese facilitar las cosas con generosidad.

Sinceramente, he recibido más que he dado, gracias por todo lo que me has aportado y que la vida te siga tratando bien, incluso mejor si es que es posible.

Un fuerte abrazo

Lidia Moreno

«He not busy being born is busy dying»,
Bob Dylan, It's Alright, Ma, Bringing It All Back , 1965

El documental que Martin Scorsese dedicó a Bob Dylan llevaba por título «No Direction Home», cuya traducción literal puede ser «Sin dirección» o «Sin hogar», pero que en el contexto del cantautor de Minnesota significa que se trata de un individuo impredecible, incon-



formista e imposible de etiquetar. Por eso me ha venido a la mente al pensar en Isidro Ramos en el momento de su jubilación formal, que no vital.

Porque Isidro ha hecho muchas cosas en su vida y creo que en todas ha dejado huella, desde su paso por el Centro de Cálculo de la Universidad Complutense a su labor de Rector de la Universidad de Castilla—La Mancha, pasando por su intensa dedicación a la universidad. Tampoco estoy seguro de que todo ello haya sido muy planificado: según han surgido desafíos se ha enfrentado a ellos con su carácter pausado, decidido y creativo, aprendiendo de cada opción supongo, pero dejando su impronta y marcando el camino a los que han venido detrás. En una entrevista el llamado peregrino del rock afirmaba cínicamente «Para ser honesto, no tengo ninguna esperanza en el futuro y sólo espero tener suficientes pares de botas para cambiarme». Afortunadamente no le ha faltado el calzado a un personaje como Isidro que siempre ha mirado al futuro con escepticismo e ironía, pero con determinación.

Y es que cuesta destacar una virtud en Isidro por encima de las demás, aunque me atreveré a decir que lo que más me ha sorprendido a mí personalmente es su capacidad de seguir teniendo una producción investigadora, asistiendo a congresos y reuniones científicas y creando escuela de forma profunda a lo largo de su vida y a pesar de su intenso paso por la administración. «Entonces era demasiado viejo, ahora soy más joven», cantaba Bob Dylan y le sienta a la perfección. Tampoco puedo olvidarme de cómo ha dado cuerpo completamente en España a la disciplina de la Informática en general y a la Ingeniería de Software en particular.

Soy incapaz de imaginar a Isidro inactivo y menos a su cerebro, donde sus neuronas seguirán en incansable acción. A qué va a dedicar su energía ahora, pues «la respuesta está en el viento», eso sí, en el viento de ese Levante tan querido para él.

Y es que, ¡caray!, sin duda estamos ante un gran tipo del que me honro disfrutar de una sensación de amistad. ¡Gracias, Isidro! En la duda que nos regalaba Dylan y que inicia este homenaje, «el que no está ocupado en nacer está ocupado en morir», tú te encuentras claramente posicionado y eso nos ilumina a todos los que te hemos conocido.

Juan José Moreno Navarro



Isidro, este escrito va dedicado a ti, a contarte quién has sido en mi vida y qué ha significado para mí conocerte.

Todavía recuerdo aquel comienzo del verano de 2002 en que viniste a la escuela de verano en Albacete. Yo me encontraba perdida, buscando una solución a aquel ir y venir de tesis, sin un rumbo fijo, sin unos objetivos claros. Sólo hizo falta unas palabras para contarte mi situación y tú me ofreciste tu ayuda. Me ayudaste a dar, por primera vez, esos primeros pasos en lo que finalmente resultó ser el objetivo de mi tesis. Conseguiste que por fin se hiciera la luz en aquellos momentos en que todo parecía oscuridad a mi alrededor.

Debo darte las gracias no sólo por ayudarme a dar esos primeros pasos, sino también por darme libertad para elegir mi camino. Libertad para acertar y equivocarme, y sobre todo para escucharme, cualquiera que hubiera sido el resultado. Libertad necesaria para conseguir crecer, y fortalecer esa nueva parte de mí, pero acompañándome en el proceso, advirtiéndome de los peligros y emociones del camino que se ha de recorrer para llegar al final de una tesis.



Por todo, ello no puedo otra cosa que darte las gracias por lo que me has ayudado a ser, por haber tenido el placer de encontrarte en el camino de mi vida, y por ser simplemente tú.

¡¡Gracias Isidro!!

Elena Navarro Martínez

Cuatro episodios relevantes en mi carrera académica

Conocí a Isidro hace más de cuarenta años. Aunque sólo en el primer año hemos coincidido en la misma universidad, Isidro ha jugado un papel relevante en determinados momentos de mi carrera, como explico en los siguientes cuatro episodios. Sin embargo, he de decir que creo que yo no fui un caso aislado. Estoy seguro que otras personas podrían contar historias similares. Si algo le ha caracterizado ha sido su papel de animador, provocando a la gente para que trabajara y prestándoles su ayuda cuando hiciera falta.

Episodio 1 – Primeras experiencias

Conocí a Isidro en el curso 73/74. Yo estudiaba quinto curso de Matemáticas en la U. Complutense, en la especialidad que entonces se llamaba de «Cálculo Automático» y él era el profesor de la asignatura «Compiladores y Sistemas Operativos» de esa especialidad.

Isidro acababa de volver de Nancy, donde había hecho una estancia larga (creo que de

un año) y venía con un cierto ímpetu revolucionario. El primer día de clase nos propuso hacer las clases de modo no convencional: en vez de la típica clase magistral, él nos presentaría algunas ideas que discutiríamos entre todos. El objetivo era que, a partir de esas discusiones, nosotros hiciéramos el diseño de un pequeño compilador y, después, de un pequeño sistema operativo. No habría examen, sino que las notas se pondrían a partir del trabajo realizado. Como se podía esperar, a todo el mundo le pareció muy bien.

Después de llevar casi toda la carrera aburrido con las clases que recibíamos (que motivaban que no fuera un asistente asiduo), recuerdo las primeras clases de Isidro interesantes, estimulantes y entretenidas. Lamentablemente, la cosa no duró mucho. Aunque éramos un curso participativo, según pasaron algunas semanas, una parte mayoritaria de la clase comenzó a ponerse nerviosa porque consideraban las clases demasiado poco estructuradas y no conseguían hacerse a la idea de cómo abordar el trabajo que teníamos que hacer. Como consecuencia, se organizó una asamblea en que la mayoría votó por volver a la estructura tradicional (clases tradicionales seguidas de un examen), dejando el trabajo como algo opcional. El resultado de la asamblea se le comunicó a Isidro. Imagino que no le gustó mucho, pero aceptó el resultado de la asamblea. Las clases ya no volvieron a ser tan divertidas, pero siguieron siendo lo suficientemente interesantes como para que fueran una de mis motivaciones para dedicarme a esto.

Episodio 2 – Encuentro en un pasillo

Durante el curso 73/74, Isidro ganó unas oposiciones de Profesor Agregado (figura de profesorado entre los Catedráticos de Universidad y los ahora Profesores Titulares) en Bilbao. Como consecuencia, el siguiente curso ya no seguiría en la Complutense.

Mientras tanto, yo simultaneaba mis estudios de quinto curso con una beca de IBM, donde tenía que trabajar cuatro horas diarias. Al acabar el curso tenía que decidir qué haría. Por una parte, tenía la posibilidad de quedarme en IBM, por otra parte, José Fernández Prida profesor de «Teoría de la Computabilidad» me ofrecía hacer la tesis con él. Para ello, me ofrecía un contrato de Ayudante con dedicación «normal» (era la dedicación más baja, por debajo de la dedicación «exclusiva» y la dedicación «plena»). Eso suponía ganar 1.500 pesetas al mes, menos de lo que me pasaban mis padres para mis gastos mientras estudiaba (2.000 pesetas), menos de la mitad de lo que me pagaba IBM como becario (4.000 pesetas) y 12 veces menos que el sueldo de entrada en IBM (18.000 pesetas). Para mí, la decisión para mí fue clara: hacer la tesis y tener menos dinero que cuando estudiaba.

Cuando menos lo esperaba, Isidro resolvió mi problema económico. Un día de Septiembre de 1974, iba por un pasillo de la Facultad de Matemáticas cuando me encuentro con Isidro que inmediatamente me pregunta si he terminado la carrera. Cuando le respondo que sí, me convence de que tengo que encargarme de la asignatura de «Compiladores y Sistemas Operativos». Me lleva al Director del Departamento y rápidamente se acuerda que me harán un contrato de Profesor Encargado de Curso (equivalente a lo que hoy es un asociado). Como consecuencia, pasaría a cobrar 6.000 pesetas al mes, ¡un lujo!

Episodio 3 – Isidro head hunter

Cinco años después, en diciembre de 1979, depositaba mi tesis doctoral. Mientras llegaba el momento de la lectura (sería en mayo), me preguntaba qué haría el año siguiente. En ese momento, yo estaba bastante a disgusto en la Facultad de Matemáticas de la Complutense. Por una parte, éramos como de segunda clase, porque nuestra especialidad era sistemáticamente cuestionada. Por otra parte, me explotaban de mala manera: tenía que dar doce horas de clase a la semana en cuatro asignaturas diferentes («Tecnología de la Programación», «Compiladores», «Sistemas Operativos» y «Bases de Datos»). Me habían prometido que cuando leyera la tesis me

harían un contrato mejor y me reducirían la carga docente. El problema es que yo no me lo creía. Estaba seguro de que, llegado el momento, me dirían que era imposible cambiarme el contrato (lo que, de hecho, ocurriría).

Una vez más, Isidro vendría al rescate. Un día de primavera me llama y me dice que en la nueva Facultad de Informática que había en Barcelona estaban buscando a alguien para una plaza en el Departamento de Programación de Computadores y que yo podía ser un buen candidato. Conclusión, el siguiente curso era profesor en la Facultad de Informática de Barcelona.

Episodio 4 – El contubernio hispano-francés

En el año 80 había varios centros de poder en la informática académica española. En los temas más cercanos al hardware, el poder estaba en manos de los catedráticos de automática como José García Santesmases en la Complutense o Gabriel Ferraté en la UPC. En el ámbito más cercano al software, Rafael Portaencasa (entonces rector de la Politécnica de Madrid) pretendía hacerse con el poder. En concreto, con la legislación de entonces, este poder consistía en tener la capacidad para decidir el resultado de cualquier concurso de plazas en informática en cualquier universidad española. En aquella situación, nosotros, un grupo de no funcionarios, con Isidro a la cabeza (el único funcionario), que trabajábamos en el ámbito del software, éramos unos mindundis.

En aquel contexto, Isidro decidió que debíamos organizarnos para constituir una especie de contrapoder. Además, Isidro y Josep Díaz, llegado un par de años antes de USA, consiguieron el apoyo del profesor Maurice Nivat, que entonces era uno de los líderes indiscutibles de la informática teórica francesa y europea. El grupo hizo una reunión fundacional en *Can Rigau*, una masía cerca de Girona, que fue seguida de otra reunión en París.

Consecuencias de esta alianza fueron, entre otras actividades: la organización de varias Jornadas Hispano—Francesas de Informática Teórica y Programación; la organización en Peñíscola, en 1981, del congreso *International Conference on Formalization of Programming Concepts*, al que asistieron investigadores de primera línea mundial, como John Backus, que había recibido hacía poco el premio *Turing*; y la organización en 1983 del ICALP, que es el congreso anual de la *Asociación Europea de Informática Teórica* (EATCS). A nivel personal, gracias a esta alianza, hice una estancia en París en el grupo de Nivat.

No sé en qué medida nuestra alianza sirvió para algo en el sentido en que pensábamos, esto es, adquiriendo un cierto peso en la naciente comunidad académica en Informática. Lo que seguro que es cierto es que todas estas actividades nos dieron visibilidad internacional e impulsaron nuestra investigación.

Fernando Orejas

La etapa universitaria es, sin duda, una de las que más marcan la vida de una persona. Es donde se afianzan los valores, donde se adquieren las bases que regirán la vida profesional, donde se anclan, definitivamente, el carácter, las ideologías y los pensamientos. Para los que hemos optado por la vida en la academia, por la docencia y la investigación, el tándem entre aprender y enseñar se convierte en el compromiso sobre el que pivota gran parte de nuestra actividad intelectual, científica, docente, profesional y, también, por qué no decirlo, personal.

A lo largo de mi andadura universitaria he tenido el placer de conocer, trabajar y colaborar con maestros de gran talla humana y humanística que, de algún modo, han influido

positivamente en ese compromiso, han marcado la senda de un modelo académico-científico y han apostado con rotundidad por una Institución pública de Educación Superior, por la Investigación y el conocimiento.

Este es el caso del profesor Isidro Ramos, artífice y promotor innegable de una estirpe de académicos y científicos nacionales e internacionales que jalonan las principales instituciones universitarias nacionales e internacionales.

Manuel Palomar



Isidro, aprovecho estas líneas para desearte lo mejor en esta etapa de tu vida, en la que seguro que seguirás tan activo y tan creativo como siempre. Es el momento de recoger los frutos de tantos esfuerzos y tantas realizaciones. Esta obra de teatro sublime que es la vida nos da muchas sorpresas. Pero por encima de todo, con el paso de los años uno puede hacer el grato ejercicio de mirar hacia atrás y percibir con claridad que hay intangibles que siempre son motivo de agradecimiento, porque conforman los pilares de lo que fuimos, de lo que somos y de lo que hacemos. Recuerdo ahora todo lo aprendido durante la realización de mi tesis doctoral, de la que fuiste supervisor y maestro. Todo ese conocimiento adquirido contigo durante esos años de trabajo intenso forma parte de esos pilares de la vida, que han sido, son y serán los cimientos de toda mi actividad profesional y científica. Los buenos recuerdos se acumulan al recordar las primeras publicaciones, los viajes, congresos, reuniones de trabajo, HP y Bristol, las aventuras iberoamericanas, las discusiones científicas profundas, incursiones empresariales… Abordamos juntos desafíos ciertamente exigentes. Unos pilares realmente robustos, que ahí han quedado y aquí están para siempre. Recibe pues desde estas líneas en este momento especial para ti mi homenaje en forma de sincero reconocimiento y agradecimiento por ello.

Óscar Pastor



Hola Isidro:

Llegado este momento, no podía desaprovechar la oportunidad de darte las gracias por ser mi maestro y mi mentor, por enseñarme tantas cosas, pero muy especialmente por contagiarme la fascinación por la investigación, la preocupación de hacer bien las cosas y el empuje de luchar por mi vocación. Hemos pasado un montón de horas discutiendo sobre el fascinante mundo de la Ingeniería del Software, el modelado o las arquitecturas software, creando ideas y fundamentos nuevos, ha sido un camino realmente apasionante. Has conseguido que sea «semánticamente precisa» y que «mis modelos siempre sean conforme a su metamodelo», frases que van siempre conmigo. ¡¡¡Mil gracias por todos esos momentos buenos que hemos compartido!!!

Besos,

Jennifer Pérez



Querido Isidro,

Te conocí en el año 1996 en las «I Jornadas de Trabajo en Ingeniería del Software» que impulsaste junto a Miguel en Sevilla y que supusieron el germen de las JISBD, que contribuyeron a que la Ingeniería del Software en España diera un salto de calidad fundamental. Poco después viniste a la UC3M a evaluar la calidad de la titulación de Informática, causando mucho «temor» por tus posibles conclusiones… y a la que volverías unos años más tarde para ayudar a Adoración a conseguir su cátedra, lo que algunos no entendieron en su momento, pero que como en tantas cosas, el tiempo te ha dado la razón.

Al poco tiempo me entrevistaste para mi incorporación a la UCLM, hecho que cambió definitivamente mi vida académica. En esta universidad ya pude enterarme del esfuerzo titánico que hiciste para su creación, de la que te puedes sentir muy orgulloso; así como de las «leyendas» que corren sobre esa época del Rector en su moto por los caminos de Castilla—La Mancha y que nada tienen que enviar a las del Quijote… ;-)

Nos ayudaste desde el principio para que pudiéramos tener éxito en la UCLM, y nos acogiste en los proyectos MENHIR y DOLMEN que, nunca mejor dicho, supusieron las «piedras fundacionales» de nuestra investigación. Nos enseñaron a investigar y nos permitieron conocer y colaborar con mucha gente de la UPV, Sevilla, Murcia, Valladolid, Granada, etc.

Además, como buen Rector conoces la importancia de la «liturgia» que encierran los actos académicos, y nos enseñaste a saber presidir y «estar» en las tesis doctorales, tribunales de oposiciones, congresos, etc. Yo sigo intentando imitarte y espero llegar a tu maestría algún día.

Por otro lado, a nivel personal, eres un ejemplo no sólo por tu entrega en el ámbito académico, acudiendo a ayudar siempre a cualquiera que te lo pida, sino también por tu inago-

table capacidad de evolución y de entusiasmo. Nos has dado muchas lecciones y yo las transmito como valiosas «best practices» a quien me lo permite… podría ponerte miles de ejemplos, pero cuando hay asuntos delicados (muy habitual desafortunadamente en nuestro mundo universitario) y tenemos que tomar partido, siempre recuerdo tu frase de «me siento muy orgulloso por los amigos que tengo, pero más aún de los enemigos», … es una pena que no figuren también en el libro ;-)

Me alegra mucho poder estar presente aquí, para expresarte mi admiración y mi agradecimiento por todo lo que has hecho y deseo que sigas haciendo. Un abrazo muy fuerte.

Mario Piattini Velthuis



Veinte años después…

Hace ya un buen número de años, más de los que seguramente Isidro o yo mismo querríamos reconocer, coincidimos en La Habana con motivo de la última reunión de la *Red Iberoamericana de Tecnología del Software* (RITOS). Se pretendía culminar una etapa en la que se asentarían las bases de la colaboración entre muchos grupos de investigación iberoamericanos en diversos ámbitos de la tecnología software. Sin embargo, gracias al empuje e iniciativa de Isidro Ramos, aquella reunión de clausura significó en realidad el nacimiento de un nuevo proyecto que aún hoy día, casi veinte años después, tiene su continuidad, aunque sea indirecta.

Desde la reunión anual del año anterior de la red, en San José de Costa Rica, veníamos estudiando la posibilidad de organizar un proyecto de colaboración de grupos iberoamericanos en el ámbito más específico de la Ingeniería del Software. Esta idea cristalizó en la propuesta de una nueva *Red Iberoamericana en Ingeniería de Ambientes Software*, que se bautizó con el acrónimo IDEAS. La redacción de la propuesta se realizó casi íntegramente en una habitación del hotel Habana Libre durante la última reunión de la red RITOS, en marzo de 2016. Allí nos encontrábamos Isidro Ramos, Carlos Heuser (de la Universidad de Porto Alegre, Brasil) y yo mismo, sin la heroicidad que caracterizó a los tres protagonistas de la novela de Alejandro Dumas, que ha inspirado el título de estas líneas, pero con un enorme entusiasmo ante las posibilidades del proyecto que empezaba a surgir. Aunque ya había tenido ocasión de conocer las habilidades oratorias de Isidro y su enorme capacidad para transmitir sus ideas de forma organizada, aún en intervenciones improvisadas, no dejaba de sorprenderme esa facilidad que demostraba para generar propuestas. De este modo, de esa habitación de hotel surgió el germen y un borrador de proyecto de lo que derivó en la Red IDEAS, que ya desde su creación aglutinó a un buen número de grupos de investigación iberoamericanos en Ingeniería del Software, y que fue reuniendo el esfuerzo y colaboración de otros muchos en años posteriores.

Fue en 1998, cuando las reuniones de la red se empezaron a organizar a modo de taller: *Workshop Iberoamericano en Ingeniería DE Ambientes Software*. Y el workshop sobrevivió a la red, continuando organizándose incluso sin el apoyo económico de la misma y convirtiéndose en el referente iberoamericano de los congresos en Ingeniería del Software. Después de 10 años de andadura, en 2009 el congreso se reinventa, uniendo esfuerzos de otros eventos iberoamericanos, pasándose a denominar *Congreso Iberoamericano en Sofware Engineering* (CIbSE).

Aunque las numerosas iniciativas posteriores que Isidro acometió hicieron que su vinculación con el workshop IDEAS fuese siendo cada vez más esporádica, he querido relatar cómo

nació y el protagonismo de Isidro en el nacimiento de un evento que considero de gran transcendencia y que mantiene su influencia en la actualidad.

Sirva este relato como ejemplo de la relevancia del papel de Isidro Ramos en muchas de las iniciativas que han permitido avanzar la investigación en diversos campos de la Informática en España, como reconocimiento a su labor durante todos estos años y como testimonio de un investigador que se siente afortunado por haber tenido la oportunidad de compartir excelentes momentos de trabajo y ocio con un gran compañero y mejor amigo.

Ernesto Pimentel



Isidro, tú ya lo sabes, pero hay cosas que, de todos modos, viene bien que le repasen a uno/a de vez en cuando, así que aquí van unas cuantas verdades. Tú le has marcado el camino a mucha gente. Yo he sido una de las afortunadas. Gracias a ti me conocieron y valoraron. Gracias a ti supe siempre qué hacer y cómo hacerlo. Fuiste, y vas a seguir siendo, guía ético y estético, generador y difusor de ideas y opiniones en esto que llamamos «Academia en Informática» y especialmente en la familia SISTEDES.

Isidro, te vamos a seguir necesitando, no desaparezcas. Has sido fundamental para todos nosotros. Fundamental, que viene de fundamento. Ese fundamento que es cimiento y al mismo tiempo conformador del edificio. Eres, en definitiva, una irrepetible buena suerte que tuvimos y queremos seguir teniendo.

Somos amigos, yo quiero seguirte recibiendo en A Coruña, quiero seguir disfrutando de tu compañía, en cuantas más ocasiones mejor, aunque sea cambiando la rueda pinchada de un escarabajo al anochecer, a lado de una selva llena de tarántulas, ¿te acuerdas? En aquella ocasión la investigación —bueno, la investigación de Chichen Itzá— casi nos cuesta un disgusto, pero ya nos enseñaste que la vida, dedicada a la investigación, no está exenta de peligros. En fin, Isidro, sigamos viviendo peligrosamente…

Nieves Rodríguez Brisaboa



PARA ISIDRO RAMOS

Dos palabras: ayuda y confianza. Son las primeras que me han venido a la mente al pensar en Isidro.

Isidro, has sido un «disparador» de eventos significativos en mi vida. En momentos clave, decisiones, palabras o acciones tuyas cambiaron sustancialmente mi porvenir, en el ámbito profesional, y a veces también en el personal.

Tuya fue la decisión que me permitió regresar a mi tierra chica, La Mancha. Estabas a un lado de la mesa, presidiendo el tribunal, y yo al otro como candidato. Lo único que yo sabía de ti es que eras un catedrático muy prestigioso que venía de Valencia. Sin conocerme, decidiste confiar en mí, un chaval de 25 años, y me encargaste montar y dirigir los servicios informáti-

cos de la Universidad de Castilla—La Mancha, que empezaba y donde todo estaba por hacer. Gracias por la inmensa oportunidad de aprender que me ofreciste. Tuyas fueron también las palabras que, años después, me animaron a dar el salto a la Academia y, poco después, entrar en el mundo de la investigación. Gracias por ayudarme a descubrir mi vocación.

Isidro, Castilla—La Mancha, universidad y región, deben estarte muy agradecidas. Aceptaste venir a «la estepa manchega», nada menos que a crear una universidad. Tu esfuerzo para que la criatura adoptara buenos y saludables hábitos desde pequeñita dio sus frutos. Varios decenios después esa universidad, por la que casi nadie daba un duro, está posicionada entre las 20 primeras del país en alumnos y en producción científica.

Has sido un pionero de la Informática en España, jugando un papel clave en el crecimiento y consolidación en el ámbito académico de la ingeniería del software y de los sistemas de información. Recuerdo cuando en 1998 abriste hueco a nuestro incipiente grupo de investigación en el proyecto nacional MENHIR. Dos años después, tus extraordinarios dominio del lenguaje, habilidad conceptual y cultura quedaron sobradamente demostrados cuando nos explicaste, jugando con las siglas y el significado arqueológico de las palabras, por qué después del proyecto MENHIR haríamos el DOLMEN. La comunidad de docentes e investigadores en ingeniería del software y en sistemas de información tenemos mucho que agradecerte.

Quiero acabar con una última anécdota. No he olvidado ni olvidaré aquel día, allá por 1986, en que tú, mi Rector, catedrático de Informática, prestigioso experto en bases de datos, me diste efusivamente las gracias cuando te mostré un programa que había hecho en dBase III para resolver uno de los muchos quebraderos de cabeza que la escasez de recursos humanos provocaban. En ese momento decidí adoptarte como mi Maestro.

Tres palabras: ayuda, confianza y gratitud.

Francisco Ruiz González

Hace ya bastantes años que conozco a Isidro Ramos, tanto en su faceta de profesor como en la de investigador. Mi apreciación sobre él ha variado con el tiempo, siempre se valoran mejor las cosas teniendo un poco de perspectiva temporal.

Hay personas que son una fuente de inspiración, por sus pensamientos e ideas que nos hacen siempre reflexionar y mejorar en nuestro trabajo. Pienso que Isidro es una de ellas, alguien al que ha merecido la pena conocer, tanto en su faceta personal, como en su trabajo de profesor e investigador.

Esperando que disfrutes de una jubilación tranquila, un fuerte abrazo.

Juan Sánchez

Si tuviera que resumir lo que siempre he admirado de Isidro es su capacidad de trabajo. Me resultaba impactante dejar en su casillero versiones iniciales de artículos para congresos y volver por la tarde y encontrarme de vuelta el documento corregido y lleno de comentarios enriquecedores. Ese entusiasmo y dedicación por las cosas bien hechas siempre fue para mí un

punto de referencia que he tratado de llevar presente desde mi marcha de la Politécnica de Valencia hace ya catorce años. Mis palabras son por tanto de simple y sencillo agradecimiento por su entrega, por su visión adelantada de las cosas, por su capacidad para conectar ideas y poner en marcha nuevas iniciativas, y por la flexibilidad y libertad que siempre nos ofreció para dirigir el trabajo investigador hacia aquellos temas en los que nos encontrábamos mejor identificados.

Muchas gracias Isidro.

Pedro Sánchez Palma

La historia de la Informática en España ha sido compleja y en muchos casos errante. Anteriormente a los años 80 sólo existían pequeños núcleos de investigación y algunas especialidades dentro de algunas Facultades de Matemáticas y Física. Es en el año 1980 cuando se crean las Facultades de Informática en Madrid, Barcelona y San Sebastián y a mediados de los 80 se crean facultades por toda España.

La Informática se creó con una visión científica y en un primer momento no tuvo interés por una visión ingenieril. Se creó como una licenciatura y, posteriormente, se convirtió en una ingeniería pero sin cambiar esencialmente el enfoque. En 1990 se establecen los estudios de Ingeniería Informática y poco a poco las Facultades de Informática han ido evolucionando a Escuelas Técnicas de Ingeniería Informática. En los últimos años se está produciendo un cambio acelerado en las escuelas de Informática hacia el enfoque cada vez más ingenieril.



La evolución de la informática en España en los últimos 30 años ha sido formidable: en su extensión geográfica, en el número de facultades, posteriormente convertidas en escuelas técnicas, en el número de departamentos de informática, en el número de profesores y de investigadores, en la creación de nuevas líneas de investigación, en la conexión internacional de la investigación española en informática…

Algunas líneas de investigación no existían en los 80 y hubo que ponerlas en marcha. En 1996 se iniciaron en Sevilla las *Jornadas de Ingeniería del Software*. Hasta el año 2005 no existe ninguna estructura organizada de la investigación en informática en España. En 2005 se crea el *Congreso Nacional de Informática* (CEDI) y la *Sociedad Científica Informática de España* (SCIE).

Este cambio tan inmenso haya sido posible porque ha habido personas que lo han impulsado. La generosidad de algunos, orientando la puesta en marcha de nuevos grupos e investigación, creado nuevas líneas de investigación, estando disponibles para jornadas agotadoras de evaluación de proyectos, para tribunales de tesis, para tribunales de todo tipo, lo ha hecho posible.

De todas las personas que han hecho posible la maduración acelerada de la informática en España hay una que destaca sobre las demás: Isidro Ramos. Creo que le debemos mucho todos pero especialmente aquellos, que como yo tuvimos que poner en marcha centros, líneas y grupos de investigación nuevos, en un entorno confuso y convulso.

Miguel Toro

Querido amigo:

Es difícil resumir en unas pocas palabras las sensaciones y situaciones que durante todos estos años hemos compartido. Cuando, a principios de los años 80 empezamos con nuestra escuela universitaria, esta región era un desierto académico, al menos en relación con la Informática. Tu ayuda, siempre generosa y desinteresada, contribuyó a salir de esta situación. Personalmente, me diste la oportunidad de conocer y trabajar en uno de los campos que considero más interesantes de la informática, la ingeniería del software, de la que, una vez superadas álgebras, categorías y morfismos, tanto he disfrutado. De forma más general, también te he considerado siempre un referente de lo que debe ser un profesor universitario. Personalmente, te doy las gracias por toda esta ayuda y por todo lo que me has enseñado, quisiera ser capaz también de transmitir a otros al menos parte de todo esto, y del amor a la profesión y a la universidad. Y para que lo demás no suene a epitafio, cambio el registro y sigo en panocho.



EL EMPIECE:

Según mi filosofía cierta, sé d'el amor que tiés a las llenguas de los guiris de más p'allá o de más p'acá de Perineos (tó en er güen sentío), pos sin tapujos ni monsergas contenúa esta plática en la llengua er panocho, ¡mi llengua maere!, la der paradiso d'España, futuro Reino e Murcia, y abora que'stá e moa er pidir la endependencia, y por si s'esparrama Murcia por el norte, desanchando la frontera, y llegar hista la Valencia, pos que pué ser de utiliá el esturrie de lo murciano, que nunca se conoce bastante.

Renchío de gran arbullo, encomencipio p'ajraecerte Isidro una sinfinitú c'hayas sío mi compañero, vaya mi reconocencia a ti, y argo muncho más importante que m'abotarga e gozo, qu'és l'amistá apretá que sustentamos. Isidrico er Güeno, isfruta y saca la panza, que jufilación no es que tó s'acabe, que viene de júfilo, qu'és alegría, qu'és gozar con ansia. Lo que desembucho no creas que son sermones e vieja, es la mesmitiquia verdá.

Arrejúntate con las güenas compañas de toa la vía, cuídate a pico rollo, ¡no esfarates tu pacencia! Y como güen zagal qu'éres, que no zanganearás, goza de lo qu'ensueñas, estrújate la sesera, estudia con mucha ensa con libros moernos, revistas, cicropedias y chalaico con la intrené esa… y'estarás como er torero, al que l'han dao las dos orejas.

Y ya, LA REMATAERA:

Isidro, sirva esto como riverencia a ti, y que nos esfisemos a luego, a luego, y munchas veces, desenrobinando tuercas, aligenciándonos vínicos e Bullas, Jumilla o Yecla, y si es arrodeaos de zagalas, mejor qu'ellas sean majas y potrancas, que premurgo entusiasmao qu'en asuntos e bragueta, de la lébido del seso, entavía eres zanguango. Y gozar del divertimento con güenos corriquios o parvá e colegas, asín sean los manchegos, castellanos, andaluces e gallegos, d'otros que me denje y'hasta que si fueran de Cartagena, pos tamién.

Asín que c'a cochino a su camino y c'alimal a su bancal. En mi nombre y'en el de la flor y nata c'a viníO a estar contigo, y c'an rellenao este tormo: un abrazo con el arma.

HE DICHO. El Ambrosio.

Ambrosio Toval

Isidro, seguramente no sabes que te vi por primera vez cuando opositabas a una plaza de Agregado junto con Antonio Vaquero. Yo acababa de entrar como becario y vosotros dos erais entonces la referencia para los que empezábamos. Como sabes, no he estado en tu órbita más cercana, pero sí que hemos tenido muchos encuentros y compartido algunas actividades. Particularmente, te agradezco que me llamaras para participar en la red RITOS, lo que nos permitió compartir un par de viajes por Sudamérica. Pero nuestro principal marco de colaboración han sido las JISBD. Tu liderazgo en la creación y desarrollo de estas jornadas ha sido fundamental para la consolidación de las mismas.

Trabajador incansable, entusiasta y dinamizador, eres sin duda uno de los investigadores informáticos más conocidos e influyente de España. Has creado escuela por las diferentes universidades por las que ha pasado, influido en gran parte de las decisiones sobre política universitaria que se han tomado en nuestra disciplina y unido, principalmente a través de la conocida JISBD, a la mayoría de los grupos españoles de Ingeniería y Desarrollo del Software.

Isidro, me alegro y enorgullezco de haber colaborado contigo, sé que todavía tienes

ganas y energía para seguir trabajando y espero que lo sigas haciendo por el bien de todos. Te deseo lo mejor, un fuerte abrazo.

Jose María Troya



La primera vez que vi a Isidro fue en San Sebastián, en Septiembre de 1997, en las Jornadas de Ingeniería del Software (JIS). En ese momento yo era un reciente alumno de doctorado, y para mí era el primer congreso nacional al que asistía. Iba bastante cohibido y preocupado pues sabía que iba a presentar mi primer artículo delante de la comunidad española, era consciente de mi inexperiencia y bisoñez, y todo me imponía muchísimo. Todo el mundo parecía saber de todo, y los «grandes nombres» eran figuras de gran prestigio, sobre las que circulaban todo tipo de historias.

Cuando llegué al congreso se estaba celebrando una mesa redonda sobre investigación en Ingeniería del Software, y uno de los panelistas estaba respondiéndole con contundencia a una asistente sobre una pregunta sin sentido que había formulado. La verdad es que cuando lo vi me impuso muchísimo, me terminó de asustar completamente, y con pavor no pude más que preguntarle al que estaba a mi lado: «¿Quién es el que está hablando? ¡Menuda personalidad!» La persona a la que le pregunté me miró sorprendido y me dijo «¿De verdad no sabes quién es? ¡Es el profesor Isidro Ramos, catedrático de la Politécnica de Valencia!».

Así que ése era el gran hombre… Todos me habían contado muchas historias sobre sus profundos conocimientos científicos, sus publicaciones, su formación y sus estancias en Francia, su experiencia en gestión e investigación, su labor como pionero de la Informática en España, su dimensión internacional, su genio y fuerte personalidad, sus logros científicos y profesionales (había creado departamentos y grupos de investigación en vario sitios, había tenido puestos de alta gestión, había sido rector de una Universidad, …). Todo el mundo le temía y admiraba, por partes iguales: con él no valían medias tintas; era una persona directa y firme, con conocimientos y autoridad, y que esperaba lo mejor de cada uno. No le valía otra cosa. La primera impresión corroboró, e incluso aumentó, mi preconcebida idea sobre él. Por supuesto, a partir de ese momento traté de evitarlo todo lo posible durante el congreso, tal era el respeto y admiración que sentía por él (y también mucho susto, he de admitir).

No fue hasta varios años después, durante otras JISBD, cuando lo conocí en persona (tras perder el miedo a hablar con él). Para mi sorpresa, muy lejos de ser esa figura inaccesible y autoritaria que me esperaba, Isidro resultó ser una persona absolutamente humana, cercana, llena de sensibilidad, cultísimo, y muy cariñoso. Hablaba bien de todo el mundo, y mostraba un respeto exquisito por los demás, y estaba muy preocupado porque toda la comunidad informática, de la que él era precursor, se desarrollase y fortaleciese. Desde ese momento mi admiración por él ha ido siempre en aumento, y hemos coincidido, tanto en España como (sobre todo) en el extranjero, en numerosos congresos, tribunales, plazas, y comisiones. He tenido la suerte de poder hablar con él de muchas cosas y compartir visiones sobre nuestra disciplina. Y he de decir que si algo me impresiona sobremanera de Isidro no son (solo) sus logros profesionales, sino sobre todo su dimensión humana.

De hecho, cuando surgió la oportunidad de escribir estas líneas estuve considerando varias opciones a la hora de ocupar el poco espacio que nos dieron. Sin embargo, pronto decidí que no iba a dedicarlo a resaltar sus grandes logros académicos, científicos y profesionales, pues de ello ya se ocupa el resto de este libro, y también lo harán las crónicas oficiales. Lo que quizá

no recojan son algunas de las facetas de Isidro que, desde mi perspectiva, son las que verdaderamente lo hacen un gran hombre, un gran científico y un gran gestor. En particular, su espíritu extraordinariamente joven, mucho más joven que el de la mayoría de nosotros, y la ilusión que muestra en todas las cosas que hace. Ojalá todos tuviéramos esa fuerza y ese empuje, o su gran capacidad de trabajo y su espíritu de sacrificio. Como he repetido en más de una ocasión, «yo cuando sea joven quiero ser como él» :-). También quiero destacar lo que ha hecho por todos nosotros, y su disponibilidad para ayudar a todo el mundo. Cuántas veces lo he llamado para preguntarle una duda o pedirle un consejo sobre un tema delicado, y me ha cogido el teléfono al instante a pesar de estar en los Alpes, navegando por el Mediterráneo, visitando a su hija en Francia o de viaje en un congreso. Isidro siempre ha estado ahí, cuando se le necesitaba, de una forma generosa y absolutamente desinteresada.

En cualquier caso, no creo que Isidro se jubile. Ya se lo he escuchado varias veces, pero personas tan activas y valiosas como Isidro no pueden (ni deben) jubilarse. Yo al menos no quiero que lo haga. Podemos permitirle que deje de ir al trabajo, pero no que deje de trabajar. Y menos cuando aún tiene muchas cosas que hacer, muchas cosas que enseñarnos. Estoy seguro de que tendrá muchos planes para los años venideros, pero también estoy seguro de que entre esos planes está el de seguir formando parte de nuestra comunidad, ayudándonos a seguir creciendo, a seguir desarrollando la disciplina que él hace ya unos años muy sabiamente comenzó a construir. Dudo que nadie haya hecho tanto por la Informática en España como él, ni que nadie lo haga en un futuro cercano. En este sentido, todos le estamos eternamente agradecidos y queremos que siga con nosotros. Así que, más que un epílogo, este libro me gustaría que constituyera un prólogo de lo que Isidro aún tiene por hacer.

Antonio Vallecillo

Tuve mi primer contacto con Isidro Ramos, como alumno, en la segunda mitad de los 80. Cuando me dio clases. Creo que los alumnos no tienen conciencia de la relevancia de sus profesores. Es algo en lo que tenemos que trabajar. Yo, por aquel entonces, no sabía quién era Isidro Ramos.

Hoy, como director, he tenido la oportunidad de reunirme con colegas de todas las escuelas de informática en España y he tenido constancia del reconocimiento y valor que se le atribuye a Isidro Ramos. A su persona y a su legado.

Isidro es uno de los referentes cuando se habla de la informática en España. Es mencionado en cada evento sobre los orígenes de la informática y su nombre se asocia a la generación de grupos de investigación y departamentos.

Isidro es un valor que siempre tendrá nuestra escuela y nuestra universidad.

Este es el legado que nos queda. Que no es poco.

Eduard Vendrell

Querido Isidro

A estas alturas de nuestras vidas, lo que más apetece evocar son los recuerdos de los buenos ratos que hemos compartido en las actividades post-trabajo por todos los lugares a donde nos ha llevado nuestra profesión informática, desde aquel inicial café concierto de Barcelona, cuando todavía era tu estudiante y trataba de desentrañar el análisis semántico, a la última cena en septiembre pasado en la plaza cerca de tu casa en Valencia, en donde me hablabas de tus trabajos de modelado en PLN con los colegas de Alicante (fiel a ti mismo: teoría, teoría y teoría).

Por el medio ha habido muchos más, claro está, con frecuencia variable, porque ambos hemos sido viajeros impenitentes. Pero siempre he sentido que encontraba fácilmente el hilo con el que reconectar. En esencia, la capacidad de renovación y la curiosidad polifacética, son los dos rasgos que más admiro en tu trayectoria vital, y que con afecto, quiero dejar reflejados en estas líneas.



Felisa Verdejo



Conozco a Isidro desde hace más de 25 años. Siempre me sorprendió su capacidad de hacer cosas y su interés por mejorar la Universidad, independientemente de las circunstancias adversas con las que tuviera que enfrentarse. De él aprendí que la política universitaria puede tener buenos fines y ser beneficiosa para cumplir los objetivos que tiene la Universidad y que no solo tiene que ser el arte de gestionar recursos administrativos sin finalidad alguna. También aprendí a luchar por las cosas en que creía hasta el final y no darlas por perdidas ante la primera dificultad.

Durante todo este tiempo que he compartido departamento con Isidro me he sentido orgulloso de ser compañero suyo y he comprendido que ser un buen profesor universitario está por encima de las mezquindades y críticas provincianas del día a día.

Es sorprendente que 25 años después y con una larga y brillante carrera profesional y científica a sus espaldas Isidro nos siga dando una lección de lucha y de amor a la ciencia y a la Universidad.

Por todo ello, Isidro, recibe mi más sincero agradecimiento.

Antonio M. Vidal

LA CONEXIÓN LATINOAMERICANA

RITOS: Informativo Extra

Ricardo Baeza-Yates, José A. Pino

Agosto de 1992

En la fotografía adyacente se aprecia un hermoso ejemplar de *Ramos Floriferæ*, junto a su descubridor, el Adelantado don Isidro Ramos y Salavert, comendador de Valencia.

El insigne naturalista venía de un agotador viaje por tierras áridas cuando llegó al fértil valle amazónico de Mangaratiba. Allí admiró largamente un excelente ejemplar de *Christina Passionaria*, en una danza ritual, antes de quedar subyugado por la belleza de la planta descubierta. Fascinó al Adelantado la fuerza de la planta, el brillo de las hojas y la rareza del extremo de los frutos.

El ayuntamiento de Valencia ha declarado que en las Fallas anuales se recordará las formas del magno descubrimiento botánico, para gloria de la provincia. Mas aún, se pondrá un ejemplar de *R. Floriferæ* en la Plaza Mayor de la ciudad, para deleite de grandes y pequeñoos. En declaraciones a la prensa local, el Comendador ha afirmado que espera que decenas de miles de conciu- dadanos hagan cola para ver este monumento, y que aún lleguen turistas de lugares tan distantes como Lepe.

El gran descubridor está actualmente dedicado a hacer experimentos con la nueva especie. En un artículo pronto a ser publicado en Nature, afirma que esta planta no necesita agua para permanecer fresca y sana. Ademas conjetura, aunque todavía no ha podido demostrarlo, que la planta no cambia de tamaño. Esta última característica ha sido aprovechada



Figura 1: Ejemplar de *Ramos Floriferæ*

por un industrioso catalán que ha comenzado a producir pequenños ejemplares de *R. Floriferæ*, especialmente para las peceras que según se dice abundan en la región de Isla Cristina. Los lugareños asocian la propiedad del invariante del tamaño de la planta con la eternidad y por ende, la eterna juventud.

La planta se ha hecho popular en muchas partes, y como el nombre científico es un poco difícil, se le ha comenzado a llamar también *Ramos Ritos*. Incluso se sabe de un novelista que ha comenzado a escribir una historia de un caballero valenciano que sale a buscar la *R. Ritos* para ofrecérsela a su amada doña Eulalia, como ofrenda en el quinto centenario del descubrimiento del Nuevo Mundo. Y por supuesto, en Latinoamérica no se habla de otra cosa.



Querido Isidro:

Queriendo estar contigo en este momento, recuerdo todas nuestras aventuras latinoamericanas, especialmente en Cuba, pero también en Bolivia, Brasil, Chile, Colombia, Guatemala y tantos otros países donde nos encontramos gracias a CYTED. También recuerdo con cariño las aventuras valencianas, navegando en el Mediterráneo y luego bañándonos en caletas escondidas junto al peñón de Ifach.

Y lo de aventurero no es una casualidad, ya que es una manifestación más del Isidro científico, del Isidro capitán (en la fotografía de octubre de 2009), ese al que le gusta explorar todo, tal como a mí. Por esto sé que ahora es sólo el comienzo de otra aventura, donde seguirás navegando en múltiples aguas, como si fueran tierras lejanas en distintas latitudes.

Así que mi deseo para este día es que encuentres las mejores rutas de vida para seguir explorando todos tus mundos y de paso seguir conociéndote a ti mismo.

Un abrazo muy grande y fuerte.

Ricardo Baeza Yates

Querido Isidro,

Não nos conhecemos há muito tempo; só há 25 anos quando durante o Congresso do CLEI junto com José Antonio Cordero, nos apresentou a idéia do CYTED. Os seus grandes feitos do período anterior só ouço dos relatos dos que já te conheciam. E são relatos de grande admiração, que só reforçam a minha visão destes 25 anos. Que carreira estupenda construíste! Que exemplo para os acadêmicos jovens que tiveram a sorte de serem guiados por você! Para mim, tem sido uma honra poder desfrutar do seu conhecimento e um prazer compartilhar contigo idéias e momentos divertidos em todas os nossos encontros!

Dentre as suas diversas qualidades, a que mais me chama a atenção é a sua generosidade. A disponibilidade de dar o melhor de si para ajudar os que necessitam desde uma direção a palavras de apoio, especialmente no meio acadêmico. Citando Albert Camus «A verdadeira generosidade para com o futuro consiste em dar tudo ao presente». Tenho certeza que muitos ex-alunos e colegas reconhecerão e te agradecerão nos seus relatos por esta grande virtude

Eu comecei a observar esta sua virtude inicialmente em suas interações com os acadêmicos que compunham a Rede RITOS e posteriormente com os acadêmicos da Espanha, particularmente os alunos e professores do grupo ISSI. É muito fácil perceber este reconhecimento nos colegas que o rodeiam mais de perto, como José Hilário, Mari Carmen, Patricio e Pepe Carsi, apenas para citar alguns.

Eu desfrutei desta generosidade quando me ensinaste Valencia; quando apoiaste a minha estada como Professor Visitante na UPV; quando intermediaste com Angela a minha residência temporária; quando várias vezes me hospedaste na sua casa; enfim quando ofereceste a sua companhia nas inúmeras vezes que nos encontramos. Quanto eu tenho aprendido neste convívio! Quanto eu me sinto privilegiado por ter tua amizade!



Nesta nova etapa da vida acadêmica que se inicia desejo a ti a mais completa tranquilidade. Que consigas observar o dia-a-dia à distância sem se envolver diretamente. Que possas se controlar para deixar que problemas aconteçam sem se angustiar. Que possas se convencer que você já fez muito mais do que o desejável. Que possas manter o teu imprescindível apoio no nível conceitual, sem se desgastar. Em resumo, que um novo Isidro, ainda melhor, surja deste status que alcançastes! E que seja feliz, muito feliz!

Do seu amigo fraterno,



Marcos Borges

Isidro Ramos, uma pessoa solidária

Tive o prazer de conviver por quase uma década com Isidro, especialmente dentro de algumas das redes CYTED, ao longo da década de 90 do século passado. Estivemos juntos em vários países da Iberoamérica, ministrando cursos e oficinas. Também tive o prazer de visitar seu grupo algumas vezes em Valência.

Isidro sempre demonstrou ser alguém dotado de liderança acadêmica, capaz de montar grupos de pesquisa sólidos e duráveis e do formar os recursos humanos de qualidade que hoje atuam nas mais variadas instituições. Isto é sobejamente demonstrado por este volume.

Entretanto, a impressão duradoura que tenho de Isidro é a solidariedade, principalmente para quem está em situação de desvantagem. Para ilustrar, recordo dois eventos, ambos acontecidos durante reuniões de nossa rede CYTED.

Estas reuniões ocorriam em locais escolhidos da Iberoamérica, que tivessem condições, não só de sediar a própria reunião, mas também receber cursos e oficinas ministrados por alguns dos pesquisadores que compunham a rede. Muitas vezes, a reunião acontecia em um local em que não havia pesquisa em Computação estabelecida. Mesmo assim, normalmente convidávamos docentes locais para apresentar seus trabalhos. A ideia era tentar identificar pontos para alguma futura colaboração. Em uma destas reuniões, estivemos em um local sem nenhuma tradição de pesquisa na Informática. Por alguma razão, os docentes locais decidiram escolher como palestrante um aluno de seu curso de graduação. Ele nos falou sobre um novo paradigma de programação que havia criado. Claramente lhe faltava orientação e acesso à bibliografia, pois tudo que pensava estar criando já existia com outro nome na literatura. Terminada a apresentação, eu imaginei que não haveria perguntas. Afinal, o que já havia sido dito fazia pouco sentido. Não parecia correto ridicularizar o jovem esforçado frente a plateia. O normal seria passar ao próximo palestrante. Neste momento, Isidro tomou a palavra e, de forma elegante, fez várias perguntas e sugestões ao aluno, dando lhe um caminho para seguir seu trabalho e incentivo para tal. Tenho certeza de que este jovem até hoje tem boas recordações da intervenção de Isidro. Este é o docente solidário.

O outro evento ocorreu durante uma reunião que fizemos em La Habana. La Habana é uma cidade linda e hospitaleira e aproveitamos para, fora de nossas reuniões, conhecer um pouco da cidade. Nestas nossas incursões turísticas éramos acompanhados por docentes da Universidade local. Em uma noite, fomos assistir ao Ballet Nacional de Cuba. Na época, em muitos locais turísticos, havia preços diferenciados para estrangeiros (mais caros) e nacionais (muito mais baratos). Fomos ao Ballet acompanhados de algumas docentes locais. Após ingressar, ficamos sabendo que as docentes locais não poderiam ficar sentadas conosco, visto que seus ingressos lhe davam apenas direito a assentos distantes do palco, enquanto que os nossos ficavam em uma localização privilegiada. Recordo até hoje a indignação de Isidro com esta injustiça. Ali mesmo, ele iniciou um protesto para mudar aquela situação e somente parou quando a segurança do tetro acomodou as docentes locais junto aos demais membros do grupo. Este é o cidadão solidário.

Carlos A. Heuser

Querido Isidro:

Aunque hace tiempo que no conversamos, siempre te recuerdo afectuosamente por la amistad que tuve el privilegio de tener contigo. Recuerdo la ocasión en que nos conocimos: ese seminario en Buenos Aires al que nos invitaron; y en el que dejamos claro que no se podía disfrazar proyectos académicos fraudulentos como si fueran auténticos. En tu caso, ¡la valentía para denunciar el fraude te costó un reclamo diplomático!

¡Cómo olvidar nuestras aventuras conjuntas con CYTED! Las reuniones en varias partes de América: Quito, Cartagena de Indias, La Paz, La Habana. Y la famosa reunión de Mangaratiba, en la que disfrutamos de la vida y de la vegetación tan exuberante :-)… ¡Qué ganas de volver a esos tiempos, un poco inocentes, pero llenos de nuestro entusiasmo por hacer cosas y apoyar a otros!

Recuerdo también tus visitas a Chile y a nuestra casa en calle Puerto de Caldera, Las Condes. ¿Sabes que ese sector ahora está lleno de edificios altos? Mi casa (que por cierto, también era tuya) desapareció bajo las retroexcavadoras. Mi familia se mudó a un sector más lejano del centro de la ciudad. Recuerdo que mis hijos te apreciaban mucho por tu espíritu jovial y entretenido. Me imagino que también les gustaba tu dicción tan correcta. Hoy mis hijos ya tienen sus respectivas familias. Tengo dos nietas, que me llenan de alegría. Si nos quisieras visitar, seguro que también apreciarían tu personalidad afectuosa, como sus padres. Nuestra casa está más lejos del centro, pero aún en Santiago.

Querido amigo, me he enterado que ahora pasas a una nueva etapa en la relación con tu querida universidad en Valencia. Recibe entonces mi aprecio, amistad por siempre y admiración. Tu vida académica será motivo de estímulo para muchos y para mí en particular. Adriana me dice que también te envía sus felicitaciones y afecto.

Recibe un gran abrazo de

José A. Pino (Pepe)

APÉNDICE

PUBLICACIONES NO HALLADAS

APÉNDICE

PUBLICACIONES NO HALLADAS

Pese a la intensa búsqueda, los siguientes artículos no han podido ser localizados:

Ramos, I. Equivalencia semántica entre ejecuciones de programas no deterministas. Primeras Jornadas Hispano-francesas de Informática Teórica y Programación. Bordeaux, 1981.

Ramos, I. Programa de actuación en Informática en una corporación local. Jornadas de Informática en la Administración Local (JIAL-82), 1982.

Ramos, I. Evolución del Logicial- Publi. Real Academia de Ciencias, 1987.

Ramos, I. Pastor, O.; González del Rio, A. Herramientas CASE para fases iniciales de desarrollo del Software. Congreso Nacional de Instituciones de Educación en Informática, Sociedad Mexicana de Computación en la Educación. 1989.