BOM–Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA

María Gómez[1], Abel Gómez[1], Mª Eugenia Cabello[2], Isidro Ramos[1]
[1] Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, España
[2] Universidad de Colima, Ave. Universidad 333, 28040 Colima, México
magolac@fiv.upv.es, agomez@dsic.upv.es, ecabello@ucol.mx, iramos@dsic.upv.es

Resumen. Este documento presenta BOM–Lazy, una aproximación para desarrollar Sistemas Expertos mediante la utilización de técnicas de Desarrollo de Software Dirigido por Modelos y Líneas de Producto Software. Se ha realizado un estudio sobre la variabilidad de los Sistemas Expertos para determinar las características generales y particulares de dicho dominio. La variabilidad de tal dominio se gestiona mediante una transformación de modelos que permite obtener automáticamente diferentes arquitecturas base a partir de la arquitectura genérica de la Línea de Productos Software.

Palabras clave: Líneas de Productos Software, Gestión de Variabilidad, Transformación de Modelos, Arquitecturas Software, Sistemas Expertos.

1 Introducción

Una clase de sistemas que está cobrando gran interés en los últimos tiempos son los Sistemas Expertos (SE) [1]. Un SE es un programa software que permite simular el comportamiento de un especialista humano frente a un problema de su competencia en un determinado campo. Estos sistemas intentan codificar los conocimientos y reglas de decisión de los especialistas, incrementando la calidad y rapidez en las respuestas, dando así lugar a una mejora de la productividad del experto. La aplicación de los SE ha sido extensa en muchos ámbitos de la vida humana, algunos de los campos de aplicación son: medicina, educación, inteligencia militar, transportes, aeronáutica, agricultura, arqueología, derecho, geología, industria electrónica, informática y telecomunicaciones. Por ello, los SE empiezan a tener cada vez mayor auge, hasta el punto de ser un punto de referencia importante en la toma de decisiones, siendo de notable interés en sistemas que realizan tareas de diagnóstico. Pero el desarrollo de este tipo de sistemas es complejo, dado que los elementos básicos que conforman su arquitectura varían tanto en su comportamiento como en su estructura. Por consiguiente, existe la necesidad de soportarlos adecuadamente.

La naturaleza cambiante de la tecnología lleva a necesitar en cortos periodos de tiempo múltiples versiones de la misma o parecida aplicación. Por ello, la Ingeniería del Software debe proporcionar herramientas y métodos que permitan desarrollar una familia de productos con distintas capacidades y adaptables a situaciones variables, en contraposición a desarrollar un único producto. Ante esta situación, surge el concepto de Línea de Productos Software (LPS) [2] con la finalidad de controlar y minimizar los altos costes del desarrollo de software. Esta aproximación se fundamenta en la creación de un diseño que puede ser compartido por todos los miembros de una

familia de programas (la arquitectura genérica). De esta manera, un diseño hecho explícitamente para un producto se beneficia del software común que puede ser usado en diferentes productos, reduciendo los costes y el tiempo para construirlos. Para el desarrollo de SE se considera más adecuado el enfoque de LPS que el enfoque tradicional, puesto que este tipo de sistemas involucran una amplia gama de áreas de aplicación, con características que difieren de un caso de estudio a otro. Además las LPS facilitan el desarrollo de los productos en distintas plataformas y su uso en distintas tecnologías.

Este trabajo integra diferentes espacios tecnológicos [3]: Sistemas Expertos (SE), Líneas de Producto Software (LPS) y la iniciativa *Model-Driven Architecture* (MDA) [4] propuesta por el *Object Management Group* (OMG) como soporte tecnológico para integrarlos. MDA se basa en la separación de la descripción de la funcionalidad del sistema (*modelo independiente de plataforma*, PIM) de su implementación en plataformas software específicas (*modelo específico de plataforma*, PSM). MDA propone definir y usar modelos a diferentes niveles de abstracción como los principales activos del proceso de desarrollo de software. Sobre estos modelos se pueden realizar manipulaciones (transformaciones de modelos), para refinarlos de forma sucesiva y obtener el sistema software final. El estándar propuesto por el OMG para la definición de estas transformaciones se denomina *Query / View / Transformations (QVT)* [5]. De esta manera, en nuestra aproximación, que llamamos BOM–Lazy emplearemos todo el utillaje tecnológico proporcionado por MDA para representar y gestionar la variabilidad de la LPS mediante la construcción de modelos de dominio y su transformación en modelos arquitectónicos.

La estructura de este documento es la siguiente: la sección 2 presenta la variabilidad de los SE a través de diferentes casos de estudio. La sección 3 describe cómo es gestionada la variabilidad en nuestra aproximación. La sección 4 explica la implementación realizada y la sección 5 presenta las conclusiones.

2 Variabilidad en los Sistemas Expertos

Para mostrar nuestra aproximación se ha elegido un subconjunto dentro de los sistemas expertos: aquellos que se emplean para realizar diagnósticos. El diagnóstico de una entidad consiste en interpretar el estado de ésta a través de sus propiedades. Se ha realizado un estudio de campo [6] sobre los sistemas de diagnóstico para identificar las características de su variabilidad. Este estudio ha permitido conocer el comportamiento y la estructura de los SE en varios dominios específicos. A lo largo del artículo se emplearán dos casos que resultarán paradigmáticos: los sistemas empleados para el *diagnóstico médico* y los sistemas para el *diagnóstico educativo.*

En el caso del diagnóstico médico la entidad a diagnosticar es el paciente y el resultado es la enfermedad que padece. En el proceso se realiza un diagnóstico clínico, que debe coincidir con el diagnóstico de laboratorio, para finalmente obtener un diagnóstico integral con la participación de ambos. Así, se distinguen tres funcionalidades básicas del sistema «realizar diagnóstico de laboratorio» (1), «realizar diagnóstico clínico» (2), y «obtener resultados del diagnóstico» (3). La primera de ellas (1) es empleada por el llamado «técnico de laboratorio», mientras que las dos

últimas (2 y 3) se emplean por parte del «médico». En este caso, las propiedades de las entidades cambian durante el proceso del diagnóstico,lo que conlleva la existencia de varias hipótesis que deberán ser validadas para obtener la más adecuada, mediante un razonamiento diferencial.

En el caso del diagnóstico educativo la entidad a diagnosticar es un programa educativo de postgrado en el que se evalúan diversos criterios de calidad, y el resultado del diagnóstico es el nivel de desarrollo que tiene dicho programa. Las propiedades de las entidades no cambian durante el proceso del diagnóstico por lo que se genera una sola hipótesis aplicando un razonamiento deductivo. En este caso, el sistema experto sólo realiza la acción «obtener nivel de desarrollo» invocada por el usuario que emplee la herramienta.

2.1 Arquitectura genérica de los Sistemas Expertos

En las LPS existen partes comunes y partes variables de los productos. La parte común está representada por la arquitectura genérica, que capta la funcionalidad compartida. La parte variable involucra las características particulares adicionales que definen los productos concretos, representados por las arquitecturas base.

La arquitectura genérica de los Sistemas Expertos viene dada por un modelo modular compuesto tradicionalmente por tres módulos básicos (Fig. 1):

- El módulo «*Inference Motor*», queestablece el proceso de inferencia y toma decisiones.
- El módulo «*Knowledge Base*» que contiene el conocimiento del dominio de aplicación.
- El módulo «*User Interface*» que establece la interacción hombre-máquina.

Fig. 1. Arquitectura genérica de los Sistemas Expertos

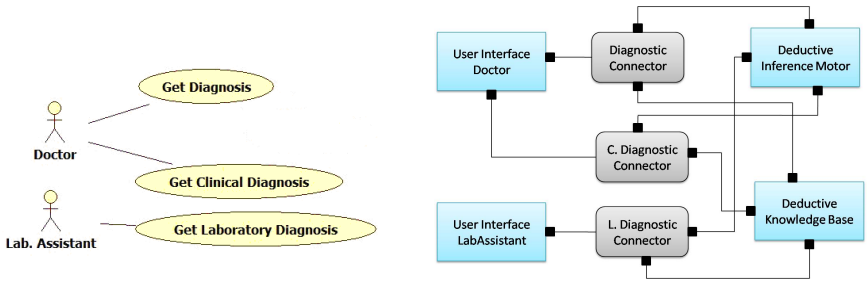
En el proceso de desarrollo de una aplicación concreta miembro de la LPS, ésta se deriva a partir de la arquitectura genérica, pero existen características particulares adicionales que definen el producto concreto. Esto implica la creación de una arquitectura base específica cada vez que se desarrolla un producto de la línea. Sin embargo se ha detectado que la arquitectura base que implementa una arquitectura genérica no es única puesto que este tipo de sistemas varían tanto en estructura como en comportamiento como se explica a continuación.

2.2 Variabilidad en la estructura de los Sistemas Expertos

Para mostrar que los elementos arquitectónicos de un SE varían en su estructura, se han modelado los requisitos funcionales que el producto final ha de satisfacer mediante un diagrama de casos de uso de UML. Estos diagramas muestran las

distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno. En particular, la estructura de los elementos arquitectónicos varía según el número de casos de uso considerados, así como el número de actores y de los casos de uso a los que accede un actor.

En la Fig. 2 se muestra el diagrama de casos de uso del dominio del diagnóstico médico junto con su correspondiente arquitectura base.



(a) (b)

Fig. 2. Diagrama de casos de uso del diagnóstico médico (a) y arquitectura base (b).

Como se observa en la Fig. 2.a, el sistema experto para el diagnóstico médico de nuestro caso de estudio tendrá dos actores, *doctor* y *asistente de laboratorio*. El primero de ellos empleará la aplicación para obtener un diagnóstico clínico así como un diagnóstico final; y el segundo de ellos empleará la aplicación para obtener el diagnóstico de laboratorio. Estos casos de usos inciden en la arquitectura base final de la aplicación (Fig. 2.b) donde dispondremos, por ejemplo, de un módulo de interfaz de usuario por cada actor de nuestro diagrama.

2.3 Variabilidad en el comportamiento de los Sistemas Expertos

La variabilidad en el comportamiento entre los elementos arquitectónicos de un SE depende del dominio de aplicación y el tipo de razonamiento usado. Como se presentó en el apartado anterior, el razonamiento puede ser *deductivo* o *diferencial* y determinará el proceso de inferencia a seguir. El proceso de inferencia es estático si las entidades tienen las mismas propiedades durante todo el proceso del diagnóstico y una sola hipótesis. Por el contrario si las propiedades de una entidad cambian durante el proceso del diagnóstico y existen diferentes hipótesis, el proceso es dinámico. De

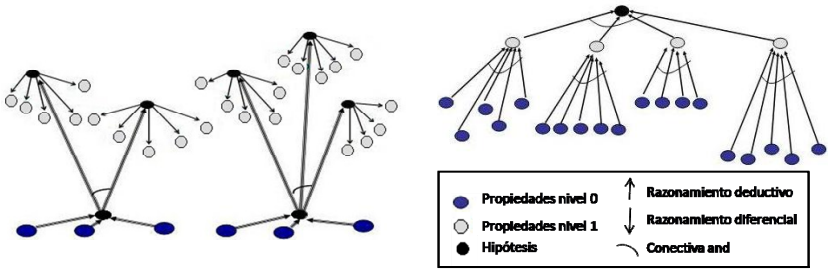


Fig. 3. Grafos descriptivos de los procesos de inferencia del diagnóstico médico (izquierda) y diagnóstico educativo (derecha).

esta manera, el diagnóstico médico requiere un razonamiento diferencial y un proceso de inferencia dinámico (Fig. 3.a); mientras que el diagnóstico educativo requiere un razonamiento deductivo y un proceso de inferencia estático (Fig. 3.b) [7]. Así, por ejemplo, la arquitectura base para los sistemas expertos del dominio del diagnóstico médico incorporarán un componente «*Differential Inference Motor*» cuyo comportamiento diferirá del de los sistemas expertos de la familia del diagnóstico educativo, que incorporarán un componente «*Deductive Inference Motor*». En la Fig. 3. se muestran los procesos de inferencia del diagnóstico médico y educativo a través de un grafo.

3 Gestión de la Variabilidad

La gestión de la variabilidad es la base para el desarrollo de los SE. La variabilidad entre los productos de una LPS puede ser expresada en términos de características [8]. En BOM–Lazy (Baseline Oriented Modeling–Lazy) las características observadas en el proceso del diagnóstico y los requisitos del usuario son considerados como puntos o fuentes de una primera variabilidad, y adicionalmente deben de ser consideradas las características del campo de aplicación a través de una segunda variabilidad, tratando así la variabilidad en dos fases. La primera variabilidad es tratada a través de puntos de variabilidad plasmados en un *Modelo de Características*, el cual permite diseñar la arquitectura base concreta. La segunda variabilidad es manejada decorando una determinada arquitectura base con las características del dominio de aplicación, lo cual determina el producto final. En lo que resta de documento se describirá únicamente la gestión de la primera variabilidad, que está relacionada con el proceso del diagnóstico y los requisitos del usuario (determinando el comportamiento y la estructura del SE como se describe en la sección 2), siendo la gestión de la segunda variabilidad un proceso semejante pero partiendo de artefactos más refinados.

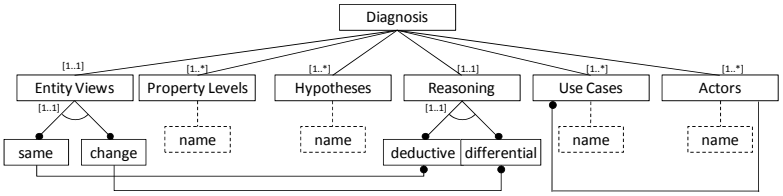


Fig. 4. Diagrama de Características de la LPS.

Las características de la primera variabilidad se representan en el *Modelo de Características* de la LPS del diagnóstico y son captadas como instancias en un *Modelo Conceptual del Dominio* (MCD). Estas características son utilizadas para configurar las arquitecturas base. La Fig. 4 muestra el *Modelo de Características* de la LPS de los SE de diagnóstico utilizando una variante de la notación extendida de Czarnecki-Eisenecker [9], donde se permite la clonación de características. En este caso, además, la notación ha sido enriquecida con el uso de atributos que sirven para identificar de forma unívoca las características que han sido clonadas.

De esta manera, el manejo de la variabilidad del dominio puede abordarse mediante una transformación de modelos, que recibe como entrada una instancia del

modelo de la variabilidad del dominio y el modelo de la arquitectura genérica de los SE, produciendo como salida el correspondiente modelo de la arquitectura base.

4 Implementación

En esta sección se presenta el desarrollo de SE mediante técnicas de transformación de modelos para el tratamiento de la variabilidad. En BOM–Lazy la transformación involucrada en la construcción de las arquitecturas base de la LPS se denomina «T1», y viene determinada por la *primera variabilidad* apuntada en la sección 3. Esta transformación genera una arquitectura base concreta (el modelo componente–conector), a partir del modelo modular (la arquitectura genérica, que describe las partes comunes de los sistemas expertos) y una instancia del modelo de variabilidad que captura la variabilidad del dominio concreto (MCD), como muestra la Fig. 5.

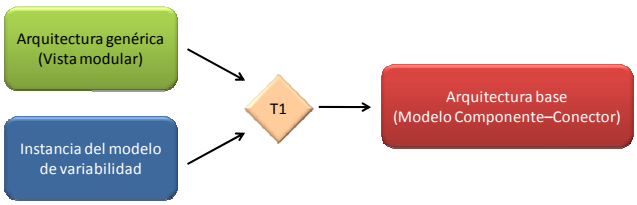


Fig. 5. Elementos involucrados en la transformación T1.

La Fig. 6 muestra el metamodelo simplificado de la vista modular empleado para representar las arquitecturas genéricas. En él se muestra que un modelo contiene un conjunto de módulos (quienes a su vez pueden contener diferentes funciones), que se vinculan a otros módulos mediante una serie de relaciones —en este caso sólo se muestra la relación de *Uso* por simplicidad—.

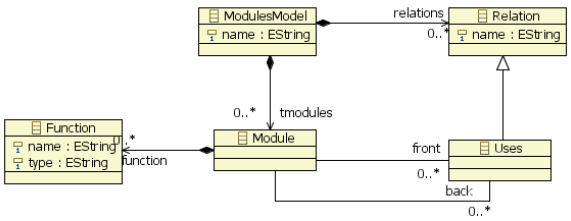


Fig. 6. Metamodelo simplificado de la vista modular.

Para desarrollar la familia de productos concretos dentro de la línea de los SE de diagnóstico debe configurarse la arquitectura genérica con las características particulares de dicho dominio. Esto se realiza mediante el modelo de características presentado en la sección 3, que finalmente se ha representado mediante un diagrama de clases de UML por simplicidad y para facilitar su uso en las herramientas de modelado. La Fig. 7 muestra cómo se representa el modelo de características del dominio de los SE para el diagnóstico (Fig. 4) según esta representación basada en los trabajos realizados en [9]. Así, una instancia de nuestro modelo de variabilidad se corresponderá con un conjunto de objetos que serán instancias de dicho modelo.

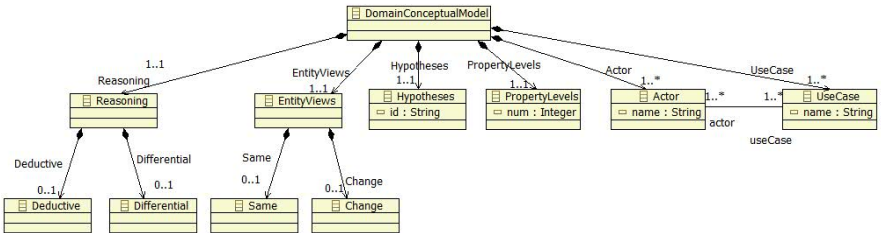


Fig. 7. Modelo de variabilidad expresado en términos de un diagrama de clases.

Por último, la Fig. 8 muestra el metamodelo simplificado de Componente–Conector, donde un modelo contiene un conjunto de componentes y un conjunto de conectores. Los componentes proporcionan una serie de servicios a través de un conjunto de puertos, y los conectores se encargan de conectar mediante sus roles los diferentes puertos entre diversos componentes.

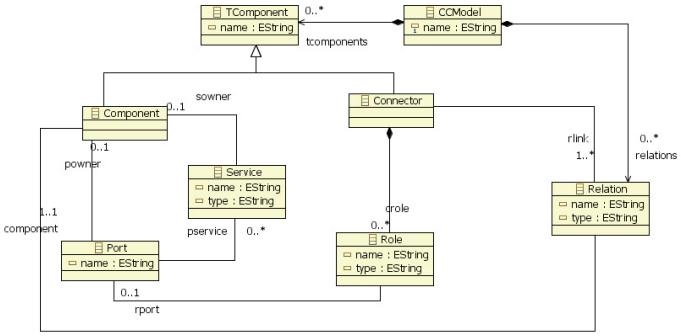


Fig. 8. Metamodelo simplificado de Componente-Connector.

4.1 Patrones de transformación

Para definir las relaciones entre metamodelos, y manteniéndonos dentro del marco de MDA, hemos empleado el lenguaje Query/Views/Transformations (QVT) [5] propuesto por el OMG. La transformación establece correspondencias entre los elementos de los modelos origen y los elementos del modelo destino. La tabla 1 muestra de forma resumida las relaciones de correspondencia identificadas junto con los elementos implicados en cada una de ellas.

Tabla 1. Correspondencias entre elementos

RELATION	*CLASES INVOLUCRADAS EN LA TRANSFORMACIÓN*		
	Modelo Variabilidad	*MM Modular*	*MM C-C*
ModulesModel2ComponentsModel	-	ModulesModel	CCModel
UseCase2Connector	UseCase	-	Connector
Module2Component	Actor EntityViews / Reasoning	Module	Component
Module2RolePort	UseCase	Module	Role, Port
Function2Service	-	Function	Service
Function2Relation	-	Function	Relation

Para establecer las correspondencias en las relaciones se han tenido en cuenta *buenas reglas de diseño software* [10] con el objetivo de incrementar la calidad del diseño de la arquitectura generada en la transformación. Algunos criterios y decisiones de diseño descritos por las reglas de transformación son los siguientes:

- *ModulesMode2ComponentsModel.— «Un diseño debe presentar una organización jerárquica que haga un uso inteligente del control entre los componentes del software».* Con el objetivo de satisfacer este criterio, la regla transforma el elemento raíz del metamodelo modular (*ModulesModel*), contenedor del resto de elementos que conforman el metamodelo, en el elemento raíz del metamodelo de Componente–conector (*CCModel*) asignándole el mismo nombre.
- *UseCase2Connector.— «El diseño debe ser modular, es decir, se debe hacer una partición lógica del Software en elementos que realicen funciones y subfunciones especificas».* Los casos de uso constituyen una división del sistema basada en funcionalidad. Por tanto, para cumplir este criterio, la regla transforma cada caso de uso en un conector coordinador que une a todos los componentes de ese caso de uso. Se estable así una relación uno a uno entre casos de uso y conectores.

  En la Fig. 9 se muestra la implementación (mediante QVT gráfico) de la regla *UseCase2Connector*. Esta regla crea, por cada uno de los casos de uso del modelo de variabilidad, un conector en el modelo Componente–Conector, tomando el nombre del caso de uso seguido de la palabra «*Connector*». El dominio modular (*mdomain*) y el dominio del modelo de variabilidad (*dcmdomain*) son de tipo «*checkonly*» (indicado con una «*C*» adjunta a la flecha del dominio). Este modificador comprueba que al menos existe algún objeto que valida el patrón definido en el dominio. Si no existe tal objeto, y puesto que la transformación se ejecuta en el sentido del dominio *ccdomain*, la aplicación de la regla no produce efectos en tales dominios. Por su parte el dominio componente–conector (*ccdomain*) es de tipo «*enforce*» (marcado con una «*E*»). Este modificador provoca que el patrón del dominio deba poderse aplicar siempre. En caso de que no haya objetos que validen el patrón, la regla deberá crear un objeto tal que sí lo cumpla, haciendo cierta la aplicación de la regla. Por último, la cláusula «*where*» indica que la relación *Module2Component* es postcondición de la regla *UseCase2Connector*, y que ésta última se debe poder aplicar correctamente tras la aplicación de *UseCase2Connector*.
- *Module2Component.—* La arquitectura genérica de los SE está formada típicamente por tres módulos. Esta regla transforma los módulos *Base de Conocimiento* y *Motor de Inferencia* en los componentes *Base de Conocimiento* y *Motor de Inferencia* respectivamente, estableciendo su tipo (deductivo o diferencial) que viene dado por los elementos *Reasoning* y *EntityViews* del modelo

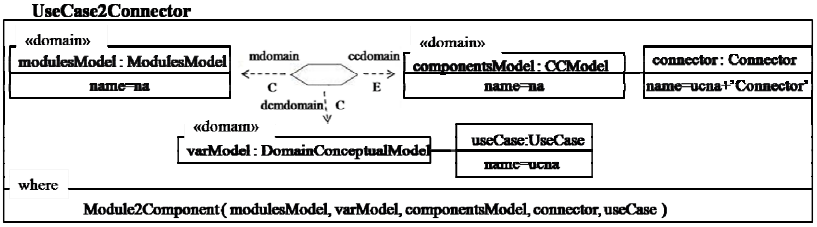


Fig. 9. Representación gráfica de la regla de transformación UseCase2Connector.

de variabilidad. Cualquier otro módulo adicional de la arquitectura genérica dará lugar a un componente que lo represente con el su mismo nombre en la arquitectura base final.

Con el objetivo de satisfacer el siguiente criterio de diseño: *«El diseño debe conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y el entorno exterior»*, el módulo *Interfaz de Usuario* será transformado en tantos componentes *Interfaz de Usuario*, como actores aparezcan en el modelo de variabilidad. De esta forma se establece una relación uno a uno entre actores en el modelo de entrada e interfaces en el modelo de salida.

- *Module2RolePort.*— Con la finalidad de satisfacer el criterio de diseño: *«Unicidad de la base de conocimiento*», que indica que el conocimiento debe ser único para todo el sistema, en nuestra arquitectura base existe un único componente *Base de Conocimiento*. Por cada caso de uso existirá una vista distinta de la *Base de Conocimiento*, se realiza la unión de las vistas en una mediante la adición de puertos a la *Base de Conocimiento* que la unen con cada conector a través de diferentes roles. Esta regla crea en cada componente un puerto y su correspondiente rol en cada conector. Por tanto, se establece una relación uno a uno entre casos de uso y puertos en la *base de conocimiento*, lo que implica una relación uno-a-uno también con los roles de los correspondientes conectores. Esta regla también crea los puertos en los componentes *Motor de Inferencia* e *Interfaces de Usuario* y sus correspondientes roles en los conectores.

5 Conclusiones

Para ilustrar la aproximación se ha elegido el dominio de los Sistemas Expertos de diagnóstico. El desarrollo de este tipo de sistemas es complejo puesto que los elementos que componen su arquitectura varían tanto en comportamiento como en estructura. Esta situación produce varias arquitecturas base en la LPS que comparten una arquitectura genérica. La aproximación presentada en este documento utiliza *QVT–Relations* como lenguaje de transformación de modelos para la gestión de la variabilidad de la LPS. Esta aproximación mejora el desarrollo de los SE aplicando técnicas de LPS al construir un diseño que comparten todos los miembros de una familia de programas. De esta manera, un diseño específico puede ser usado en diferentes productos, reduciendo costes, tiempos de producción, esfuerzo y complejidad. La aplicación de técnicas de MDA en el desarrollo, permite la construcción de sistemas independientes de plataforma, abordados desde la perspectiva del problema y no de la solución, lo que proporciona aplicabilidad en diferentes dominios. Además se ofrece una funcionalidad conjunta y coordinada de todos los espacios tecnológicos contemplados, que constituyen las técnicas actuales más avanzadas en desarrollo de software. Cabe destacar que, aunque en este artículo sólo se ha descrito la gestión de la variabilidad de la primera fase (como se comenta en la sección 3), tras la obtención de la arquitectura base el proceso de desarrollo continúa, decorándola y obteniendo una arquitectura PRISMA [11]. PRISMA es un modelo arquitectónico orientado a aspectos que dispone de la herramienta PRISMA-MODEL-COMPILER [12] que permite generar automáticamente código ejecutable en C# .NET. De esta manera, esta propuesta abarca el proceso de desarrollo completo.

Por otra parte, en aproximaciones tradicionales, el conjunto de arquitecturas base se define e implementa en la fase de diseño de la LPS (ingeniería del dominio) y permanece inmutable a lo largo del ciclo de vida de la aplicación de la LPS (ingeniería de la aplicación). En la aproximación de BOM–Lazy el empleo de la transformación T1 permite trasladar el proceso de creación de las arquitecturas base a la fase de ingeniería de la aplicación. Esto permite definir el conjunto de arquitecturas base mediante una serie de reglas que codifican patrones de buenas prácticas de diseño, así como otras decisiones de diseño de forma genérica, evitando tener que definir todas ellas de forma explícita. A medida que aumenta el tamaño de la LPS, BOM–Lazy se convierte en la aproximación más adecuada para abordar la gestión de la variabilidad, puesto que sería inviable tener construidas a priori las arquitecturas para todos los posibles productos de la LPS. Así, en nuestra aproximación, el principal esfuerzo realizado en la fase de ingeniería del dominio se realiza codificando y formalizando el conocimiento adquirido mediante un conjunto de reglas declarativas (el conocimiento se almacena de una forma explícita), y no en desarrollar de forma extensiva todas las posibles combinaciones de arquitecturas base (el conocimiento se almacena de una forma implícita). Esto redunda en una mayor eficiencia en la fase de ingeniería de la aplicación, donde partiendo del conocimiento almacenado de forma explícita se obtiene cada arquitectura precisamente en el momento en que realmente se necesita.

References

1. Giarratano, J., and Riley, G., *Expert Systems: Principles and Programming*. Fourth Edition (Hardcover). Ed. Course Technology. ISBN: 0534384471, 842 pages, Boston, 2005.
2. Clements P. and Northrop L.M., *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison Wesley, 2002.
3. Kurtev, I., Bézivin, J., Aksit, M.: *Technological Spaces: An Initial Appraisal*. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.
4. Model Driven Architecture: MDA. http://www.omg.org/mda.
5. OMG: *MOF 2.0 QVT Final Adopted Spec*., http://www.omg.org/docs/ptc/05-11-01.pdf
6. Cabello Ma. Eugenia., *BOM: Una aproximación MDA basada en Líneas de Producto para el desarrollo de aplicaciones*. Tesis doctoral, Universidad Politécnica de Valencia, 2008.
7. Cabello, M.E., Ramos, I. *Expert Systems development through Software Product Lines techniques*. In Information Systems Development: Towards a Service Provision Society. Papadopoulos, G.A., Wojtkowski, W. (et al.) (Eds.) Springer-Verlag. ISBN 978-0-387-84809-9. Due: Sept, 2009.
8. Kang K., Kim S., Lee J., Kim K., and Shin E. *FORM: A Feature Oriented Reuse Method with Domain Specific Reference Architectures*. Annals of Software Engineering, Vol. 5, 1998, pp. 143-168.
9. Czarnecki K. and Kim C. H. P. *Cardinality-based feature modeling and constraints: A progress report*. In International Workshop on Software Factories, San Diego, California, Oct 2005. http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf
10. Pressman, R.S., "*Ingeniería del Software. Un enfoque práctico*". 6ª ed. Mc Graw Hill 2006.
11. Pérez J., *PRISMA: Aspect-Oriented Software Architectures*. PhD. Thesis in Computer Science, Polytechnic University of Valencia, Spain, pages 397, Dec. 2006.
12. Cabedo R., Pérez J., Carsí J.A. y Ramos I., *Modelado y Generación de Arquitecturas PRISMA con DSL Tools*, Actas del IV Workshop DYNAMICA, Archena, Murcia, 2005.