CAPÍTULO III

PRISMA

Contenidos del capítulo:

PRISMA

En un futuro, PRISMA será un marco de trabajo o *framework* que incluirá un modelo, unos lenguajes, una herramienta y una metodología. En este capítulo se presentará tanto el modelo como los lenguajes de este *framework*.

PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables. Para ello, se basa en componentes y aspectos para la construcción de modelos arquitectónicos y se caracteriza por la integración que realiza del Desarrollo de Software Basado en Componentes (DSBC), del Desarrollo de Software Orientado a Aspectos (DSOA) y por sus propiedades reflexivas.

PRISMA define una metodología guiada y (semi-)automatizada para obtener el producto software final basándose en el paradigma de la prototipación automática de Balzer [Bal85]. En esta metodología, el usuario utilizará una herramienta de modelado (compilador de modelos) que le permitirá validar el modelo mediante la animación de los modelos arquitectónicos definidos en la especificación. PRISMA está basado en un lenguaje de especificación formal Orientado a Objetos llamado OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos, que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA preserva las ventajas de OASIS, garantizando la compilación de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos arquitectónicos, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

3.1 El Modelo PRISMA

El modelo arquitectónico PRISMA [Per03] integra dos aproximaciones de desarrollo de software, el DSBC y el DSOA. Esta integración se consigue definiendo los tipos mediante la composición de aspectos. La mayoría de modelos arquitectónicos analizan cuáles son las primitivas básicas para la especificación de sus arquitecturas y exponen su sintaxis y semántica. El modelo PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las necesidades de cada uno de ellos.

Un elemento arquitectónico de PRISMA puede ser analizado desde dos vistas diferentes: la interna y la externa. La vista interna (ver Figura 15) define un elemento arquitectónico como un prisma con tantas caras como aspectos considere. Dichos aspectos están definidos desde la perspectiva del problema y no de su solución, aumentando el nivel de abstracción y evitando el solapamiento de código que puede sufrir la programación orientada a aspectos.

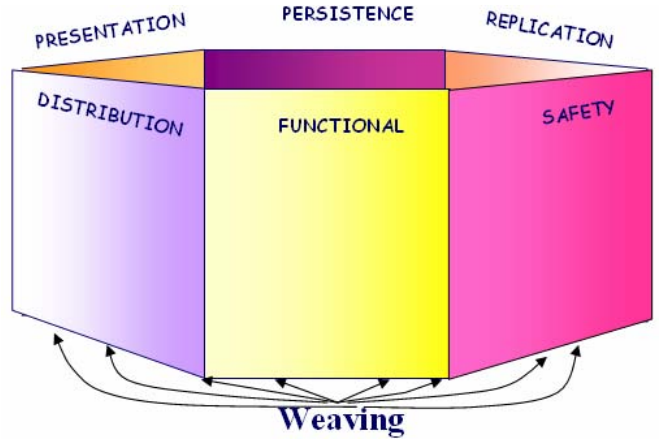

Figura 15 - Vista interna de un elemento arquitectónico PRISMA

Por otro lado, la vista externa de un elemento arquitectónico encapsula la funcionalidad como una caja negra y publica el conjunto de servicios que ofrece al resto de elementos arquitectónicos (ver Figura 16)

Figura 16 - Vista externa de un elemento arquitectónico PRISMA

Por ello, el modelo PRISMA puede analizarse desde dos perspectivas diferentes, la orientada a aspectos (vista interna) y la orientada a componentes (vista externa). A continuación se presenta el modelo desde ambas perspectivas.

3.2 Visión orientada a Aspectos

Las técnicas AOP, como han sido mostradas en los capítulos anteriores, permiten encapsular en un solo elemento (los aspectos) aquella funcionalidad que se repite a lo largo de todo el sistema. Un aspecto puede verse como la unión de un conjunto de interfaces que incluyen servicios y atributos del tipo del aspecto, más la especificación semántica de su estructura y comportamiento. No obstante, el modelo PRISMA va más allá y amplía el concepto de aspecto en el sentido de que cada aspecto representa una vista específica de la arquitectura del sistema (vista funcional,

distribución, calidad, etc.). Por ello, cada primitiva de la arquitectura está formada a su vez por varios aspectos que la describen desde diferentes puntos de vista. Algunos tipos de aspectos pueden ser:

- **Aspecto Funcional:** Captura la semántica del sistema de información mediante la definición de sus atributos, sus servicios y su comportamiento
- **Aspecto de Coordinación:** Define la sincronización entre elementos arquitectónicos durante su comunicación.
- **Aspecto de Distribución**: Especifica las características que definen la localización dinámica del elemento arquitectónico en el cual se integra. También define los servicios necesarios para implementar estrategias de distribución de los elementos arquitectónicos (como movilidad, equilibrio de carga, etc.) con el objetivo de optimizar la distribución de la topología del sistema resultante [Ali03].

Existen diferentes tipos de aspecto, definiéndose cada uno de ellos de forma independiente. El número de tipos no está limitado, ya que gracias al metanivel pueden definirse nuevos tipos. Los mostrados son sólo algunos tipos de aspectos, en otros sistemas de información puede haber otros tipos diferentes, que emergerán de los requisitos de los sistemas a los que se aplique el modelo PRISMA.

No obstante, no es suficiente con definir únicamente los tipos de aspectos, sino que también hay que enlazarlos entre sí: esto se realiza mediante los *weavings*. Los *weavings* indican que la ejecución de un servicio de un aspecto puede provocar la invocación de servicios en otros aspectos.

A diferencia de las tecnologías orientadas a aspectos vistas hasta ahora, en la que los aspectos se enlazaban al código base, en PRISMA no existe el código base como tal, sino que la funcionalidad global de una primitiva del modelo arquitectónico se define mediante la unión de los distintos aspectos que la forman. Otra diferencia importante a resaltar, es que en las otras tecnologías (como *AspectJ*) los *weavings* se definen en el mismo aspecto, resultando en una pérdida de reutilización por parte del aspecto. Es por esto que en PRISMA los *weavings* entre los aspectos se definen en el elemento arquitectónico que los integra.

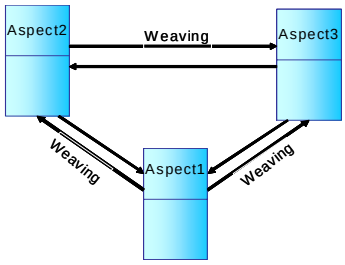

Figura 17 - Weavings entre aspectos

Como se puede observar en la figura, los aspectos se definen independientemente de la arquitectura donde se vayan a integrar, y son los *weavings* los que unen el conjunto de aspectos de un mismo elemento arquitectónico y forman la vista interna mostrada anteriormente. De la misma forma que en las tecnologías vistas anteriormente, PRISMA dispone de varios tipos de *weavings*, que son los siguientes:

- *After:* aspect1.service es ejecutado después de aspect2.service.
- *AfterIf (condition)*: igual que *After*, sólo si se cumple la condición.
- *Before:* aspect1.service es ejecutado antes de aspect2.service.
- *BeforeIf (condition)*: igual que *Before*, sólo si se cumple la condición.
- *Instead*: aspect1.service es ejecutado en lugar de aspect2.service.

Como se puede observar, los aspectos PRISMA son altamente reutilizables y favorecen la mantenibilidad, pues un cambio en un *concern* específico (i.e. el cambio en la estrategia de distribución de elementos arquitectónicos) sólo ha de realizarse en un aspecto y no en todo el conjunto del sistema. Además, la importación de COTS se expresa gracias al carácter opcional de los aspectos, lo que permite ver a las primitivas arquitectónicas como cajas negras en cuyo interior pueden incorporarse componentes COTS.

3.3 Visión orientada a Componentes

La visión externa de los elementos arquitectónicos está orientada hacia el concepto de componentes, a diferencia de la orientación a aspectos de la visión interna. El modelo arquitectónico consta de tres tipos de primitivas: *componentes*, *conectores* y *sistemas*. Y como se ha visto anteriormente, cada uno de estos tipos a su vez está compuesto por tantos **aspectos** como se consideren relevantes para definir el sistema de información con precisión.

3.3.1 Componentes

Un componente PRISMA se define como un elemento arquitectónico que captura la funcionalidad del sistema de información y no actúa como coordinador entre otros sistemas arquitectónicos. Puede verse como una parte del sistema que no se puede disgregar en partes más simples. Está formado por:

- un identificador,
- un conjunto de aspectos, que le proporcionan su funcionalidad,
- las relaciones de *weaving,* que proporcionan la cohesión entre los aspectos que lo forman,

- y los puertos de entrada y salida, cuyo tipo es una interfaz específica (un conjunto de servicios).

Los puertos son aquellos elementos que permiten la interacción del componente con los demás elementos arquitectónicos. Un puerto puede ofrecer un comportamiento servidor, cliente o cliente/servidor, en función de los servicios que lo definan. El comportamiento servidor especifica los servicios que el componente ofrece al resto de los demás elementos mientras que el comportamiento cliente especifica los servicios que puede requerir de los demás elementos arquitectónicos.

Los componentes PRISMA han de cumplir ciertas restricciones:

- todo componente debe especificar su aspecto funcional, excepto en el caso de componentes externos (COTS)
- un componente nunca puede contener un aspecto de coordinación
- un componente nunca puede contener dos aspectos del mismo tipo
- los tipos de los puertos de un componente sólo podrán ser aquellas interfaces que usen algunos de los aspectos que formen a dicho componente.

3.3.2 Conectores

Un conector PRISMA es un elemento arquitectónico que actúa como coordinador entre otros elementos arquitectónicos. El conector permite describir interacciones complejas entre componentes mediante su aspecto de coordinación. Está formado por:

- un identificador,
- un conjunto de aspectos, que le proporcionan su funcionalidad,
- las relaciones de *weaving,* que proporcionan la cohesión entre los aspectos que lo forman,
- y los roles de entrada y salida, cuyo tipo es una interfaz específica (un conjunto de servicios).

Los conectores sincronizan y conectan componentes, otros conectores o sistemas a través de sus roles, que de la misma forma que los puertos de los componentes, definen un conjunto de servicios que el conector ofrece (comportamiento servidor), que el conector necesita (comportamiento cliente) o un comportamiento mixto (cliente/servidor).

Existen una serie de restricciones que se han de tener en cuenta:

- un conector siempre ha de contener un aspecto de coordinación,
- un conector nunca puede contener un aspecto funcional,
- un conector nunca puede tener dos aspectos del mismo tipo,

- los tipos de los roles de un conecto sólo podrán ser aquellas interfaces que usen alguno de los aspectos que formen a dicho conector,
- una instancia de conector al menos ha de conectar dos elementos arquitectónicos PRISMA

3.3.3 Sistemas

En la mayoría de modelos arquitectónicos, surge la necesidad de tener mecanismos de abstracción que permitan tener elementos de mayor granularidad, aumentando la modularidad, composición y reutilización de elementos arquitectónicos. En PRISMA esto se consigue mediante los sistemas, los cuales permiten encapsular un conjunto de conectores, componentes y otros sistemas correctamente conectados entre sí. En dicha encapsulación pueden surgir propiedades emergentes.

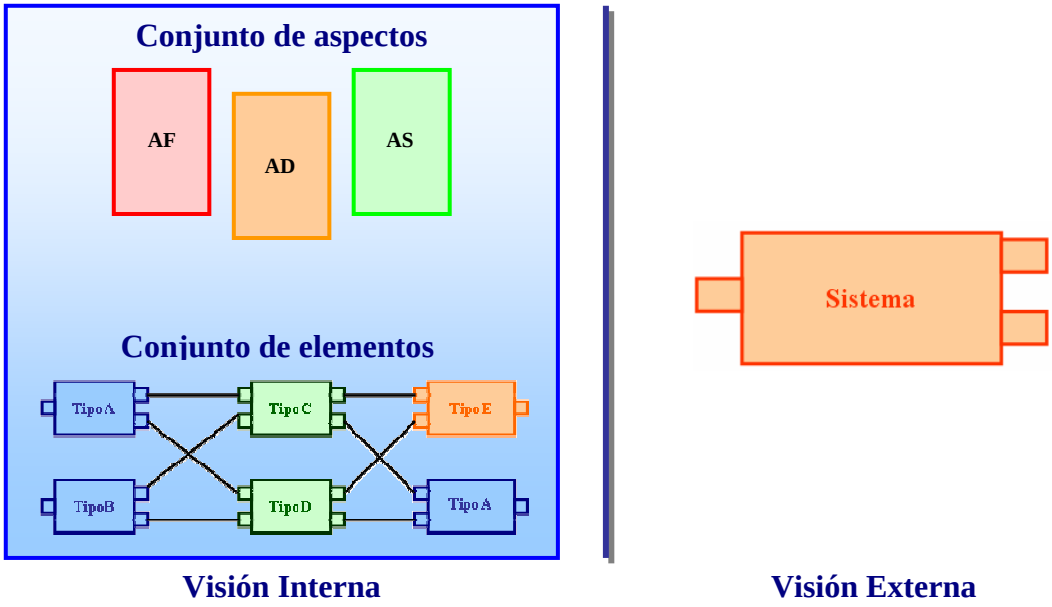

Figura 18 - Vistas de un Sistema PRISMA

Un sistema está formado por los mismos elementos y restricciones que un componente: una función de identificación, un conjunto de aspectos, una serie de *weavings* que los interconecta, y los puertos. No obstante, al ser un tipo compuesto, dispone también de un conjunto de elementos arquitectónicos que pueden o no estar conectados entre sí y realizan una determinada función para el sistema.

La especificación de la conexión entre estos elementos y el propio sistema se realiza mediante los **bindings**, que son los enlaces entre los puertos del sistema y los puertos (o roles en caso de ser conectores) de los elementos que encapsula. Permiten mantener un enlace entre los distintos niveles de granularidad de los elementos arquitectónicos que forman un sistema.

Por otra parte, las conexiones entre los distintos elementos que contiene (componentes, conectores y attachments) se realizan mediante los **attachments**, que establecen la conexión entre puertos de componentes y roles de conectores.

3.4 Lenguaje de definición de Arquitecturas

El Lenguaje de Descripción de Arquitecturas de PRISMA (ADL) está basado en OASIS [Let98], como ya se comentó anteriormente. A diferencia de otros modelos, el ADL (Architecture Definition Language) de PRISMA está dividido en dos niveles de especificación: el *nivel de definición de tipos* y el *nivel de configuración*.

El nivel de definición de tipos de PRISMA potencia la reutilización y combina el DSBC y DSOA. Este nivel permite definir los tipos necesarios para especificar un sistema arquitectónico. Dichos tipos se guardan en una librería para posteriormente reutilizarlos en la definición de distintos modelos arquitectónicos. Sus ciudadanos de primer orden son: interfaces, aspectos, componentes y conectores.

El nivel de configuración permite definir las instancias y especificar la topología del modelo arquitectónico. Para ello, en primer lugar se han de importar todos aquellos tipos (conectores, componentes y sistemas) definidos mediante el lenguaje de definición de componentes, que se necesiten para un determinado modelo arquitectónico. Después, se ha de definir el conjunto de instancias necesarias de cada uno de los tipos importados. Finalmente, se debe especificar la topología, interconectando adecuadamente las instancias del modelo.

Esta diferenciación de nivel de granularidad proporciona importantes ventajas frente a la mezcla de distintos niveles de granularidad dentro de la especificación. Una de ellas es que permite gestionar de forma independiente los tipos y las topologías específicas de cada sistema, incrementando la reutilización y obteniendo un mejor mantenimiento de las librerías de tipos. Además, permite discernir claramente entre la evolución de tipos (nivel de definición de tipos) y la evolución arquitectónica (nivel de configuración), teniendo meta-eventos diferentes en cada lenguaje para hacer evolucionar a los tipos o a las conexiones existentes entre sus instancias.

3.4.1 Definición de Tipos

A continuación se describirá la sintaxis de los ciudadanos de primer orden de PRISMA: interfaces, aspectos, componentes y conectores. Los sistemas también se definen en este nivel con el objetivo de que se puedan almacenar conjuntamente con los demás elementos y favorecer la reutilización.

3.4.1.1 Interfaces

Las interfaces son el mecanismo usado en PRISMA para establecer canales de comunicación entre los componentes, conectores y sistemas. Una interfaz describe la visibilidad parcial que tienen el resto de elementos, del estado y comportamiento del tipo que define la semántica de dicha interfaz. Por lo tanto, una instancia sólo puede solicitar y consultar aquellas propiedades y servicios publicados en sus interfaces. De este modo, una interfaz actúa como mecanismo de seguridad y permite que los componentes, conectores y sistemas sean vistos como cajas negras, preservando un alto nivel de abstracción.

Las interfaces definen los servicios sin tener en cuenta los puertos, roles o aspectos que las van a utilizar, con el objetivo de favorecer la reutilización. No obstante, se ha de tener en cuenta que todos los servicios que define una misma interfaz han de pertenecer al mismo tipo: funcional, distribución, etc. pues serán implementadas por completo por un aspecto de un tipo concreto, y no puede haber solapamiento entre aspectos de tipos diferentes.

A continuación, se presenta la interfaz *ICreditCardTransactions*, que define las operaciones típicas para el manejo de cuentas bancarias, y *IMobility*, que define los servicios de movilidad que puede contener un componente.

```
Interface ICreditCardTransactions
   Withdrawal(input quantity: decimal, output newMoney: decimal);
   Balance(output newMoney: decimal);
   ChangeAddress(input newAdd: string);
End_Interface ICreditCardTransactions;

Interface IMobility
  Move(input newLoc: LOC);
End_Interface IMobility;
```

Como se puede observar, los servicios definen los parámetros que son de entrada mediante la palabra reservada *input*, mientras que los de salida se definen mediante la palabra reservada *output*.

3.4.1.2 Aspectos

Para definir los aspectos se ha definido una sintaxis específica basada en OASIS [Let98]. La plantilla genérica se presenta a continuación:

```
1   tipo_aspecto Aspect nombre using interface1, ... interfacen;

2   Attributes
      [Constant | Variable]
        <nombre_atributo1> : <tipo_atributo>;
        ...
        <nombre_atributon> : <tipo_atributo>;
      [Derived]
        <fórmula_derivación>

3   Constraints
      static <restricciones_estáticas>

    Services
4     Begin [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio)];
        Valuations
          <fórmulas_observabilidad_atributo>

      [in | out] <nombre_servicio> (<arg_servicio>)
        [as <nombre_servicio> (<arg_servicio)];
5       Valuations
          <fórmulas_observabilidad_atributo>

      End [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio)];
        Valuations
          <fórmulas_observabilidad_atributo>

6   Preconditions
      <fórmula_precondición_evento>

7   Triggers
      <fórmula_disparo_evento>

8   Operations [transaction]
      <fórmula_transacción>

9   Played Roles
      <fórmula_played_role>

10  Protocols
      <fórmula_protocolo>

    End_ tipo_aspecto Aspect name;
```

Tabla 2 - Plantilla de un aspecto PRISMA

La cabecera de un aspecto (*sección 1*) define el tipo del aspecto (funcional, distribución, etc.), el nombre que lo identifica y las interfaces a las que se da semántica. La *sección 2* define la lista de atributos que almacenan el estado de un aspecto. Para cada atributo se indica el nombre y el tipo de valores que va a contener. Puede ser de tres tipos:

- **Constante:** su valor se establece en la inicialización y no es modificado
- **Variable:** su valor puede cambiar cuando ocurra una acción relevante (como la invocación de un servicio o la activación de un disparador)
- **Derivado:** su valor se especifica en términos de los valores de otros atributos, mediante una fórmula de derivación.

La *sección 3* define las restricciones de integridad que deben cumplirse a lo largo de toda la vida de las instancias del tipo del aspecto. Son fórmulas basadas en el estado del aspecto y se pueden clasificar en estáticas o dinámicas, dependiendo de si se refieren sólo a un estado o relacionan diferentes estados, respectivamente. Las restricciones dinámicas se caracterizan por el uso de operadores temporales como *sometimes* o *always*.

La *sección 4* define los servicios que implementa el aspecto. Todos los servicios de la interfaz o interfaces a las que da semántica el aspecto deben definirse, además de aquellos servicios propios y no públicos del aspecto. Existen diferentes tipos de servicio: el servicio *begin* permite inicializar atributos, el servicio *end* permite liberar la memoria asociada al aspecto y finalmente, los servicios de modificación y consulta. Es importante resaltar que los servicios *begin* y *end* no indican que un aspecto sea instanciable, sino que hacen referencia al servicio de creación y destrucción del elemento arquitectónico al cual el aspecto pertenecerá. Además, cada servicio de modificación o consulta puede tener un comportamiento servidor (*in*), comportamiento cliente (*out*) o un comportamiento servidor y cliente (*in/out*). El comportamiento servidor se caracteriza porque el aspecto proporciona un servicio que puede ser requerido por otros aspectos, mientras que el comportamiento cliente indica que el aspecto va a requerir dicho servicio a otro aspecto/componente.

De la misma forma que se ha visto en la definición de los servicios en la interfaz, los argumentos de los servicios que incluye un aspecto pueden ser de entrada (*input*) o de salida (*output*), pero no devuelven ningún valor.

Por otro lado, el lenguaje ADL también proporciona un mecanismo de renombrado de servicios, mediante el operador *as*, con lo que se consigue que el nombre del servicio definido en una interfaz no tenga que coincidir con el nombre del servicio definido en el aspecto.

La semántica que define el comportamiento de los servicios se especifica mediante las *evaluaciones* (*sección 5*), que permiten definir el cambio de estado de los atributos del aspecto ante la ejecución de un servicio determinado. Se definen mediante fórmulas en lógica dinámica del tipo "$\varphi \rightarrow [a]\phi$" y se interpretan como: "si en un determinado estado del aspecto se satisface φ y ocurre la acción 'a', en el estado inmediatamente posterior se satisface ϕ". Se puede no declarar la condición de evaluación φ asumiéndose *true*.

Las *secciones 6 y 7* definen respectivamente las precondiciones y los disparadores (*triggers*). Las precondiciones definen las condiciones que se deben cumplir para que un servicio se pueda ejecutar, mientras que los disparos permiten especificar la ejecución de un servicio cuando se cumple una determinada condición.

La *sección 8* define las operaciones y transacciones. Una operación es un servicio no elemental y no atómico, en el que se define una serie de servicios que se van a ejecutar secuencialmente. Para que estos servicios se ejecuten de forma atómica se emplea la palabra reservada *transaction.* Por ejemplo, la definición de una operación transaccional para el envío de noticias a una lista de correo en PRISMA sería:

```
Operations
  transaction EnviarNoticias(input listaCorreo: String, input
                  listaNoticias: StringList):
    ENVIARNOTICIAS = SeleccionarListaCorreo(listaCorreo).NOTICIAS;
    NOTICIAS =  SeleccionarNoticias(listaNoticias, coleccion).ENVIO;
    ENVIO = envioSimultaneo(coleccion).CONFIRMACIÓN;
    CONFIRMACION = confirmacionLlegada.ENVIARNOTICIAS;
```

En el ejemplo se puede observar cómo se realiza el mapeo de parámetros de la transacción a cada uno de los servicios y cómo se especifica el siguiente estado alcanzado *nombre_servicio.nuevo_estado*.

Por último, en la *sección 9 y 10* se especifican los *Played_Roles* y los Protocolos. El *Protocolo* define un proceso describiendo un conjunto de acciones cuya ocurrencia es posible. Está compuesto por un conjunto de procesos que establecen los servicios y transiciones a otros procesos posibles. Además, también permite especificar las prioridades de ejecución de cada uno de los servicios implicados.

Por otra parte, un *Played_Role* es una proyección del protocolo que define el comportamiento parcial de un rol o papel determinado. Por lo tanto, todo *Played_Role* debe ser compatible en signatura y proceso con el protocolo de forma que, dicho papel o rol se desempeña dentro del comportamiento global del protocolo y sus cálculos son un subconjunto de los cálculos del proceso.

Un *Played_Role* es una vista parcial del protocolo que tiene sentido por sí solo, un comportamiento específico y restrictivo que es posible asociarlo posteriormente a un puerto o rol de un componente o conector. Dicha asociación permite definir de forma exacta el comportamiento que debe ejercer un puerto o rol además de los servicios que publica, establecidos por la interfaz que lo tipa.

Ambos se basan en el *Cálculo Poliádico* [Mil91], el cual permite describir de forma sencilla la ejecución de procesos y servicios susceptibles de ejecutarse concurrentemente. Una parte de su sintaxis es la siguiente:

x,y,z	Nombre (interfaz, servicio, played_role, parámetro, identif.)
x .y	Jerarquía de nombres, tal que x = interfaz e y = servicio

All | *	Todos los played_roles
x(m, n)	Vector de nombres tal que m, n son parámetros
P,Q,R	Proceso
P ::= x?(m,n).P	Prefijo de entrada o acción de recepción
| x!(m,n).P	Prefijo de salida o acción de envío
| P + Q	Selección no determinista
| P ∧ Q	Conjunción
| P || Q	Composición paralela (concurrencia de procesos)
| P → Q	Composición secuencial
| x(m,n):*k*.P	*k* indica la prioridad del servicio x, tal que *k* = 0, 1, ...
| *if* b *then* P *else Q*	Alternativa (derivado)
| *case*(b ▹ P→Q)	Alternativa múltiple
| |x=y| P	Comparación o *matching*

Tabla 3 - Sintaxis de Pi-Cálculo Poliádico

Para ilustrar estos conceptos, se muestran el conjunto de aspectos (funcional, distribución y coordinación) de un sistema bancario sencillo. El primero de ellos es el aspecto funcional *BankInteraction,* que implementa la funcionalidad básica de una cuenta bancaria.

```
Functional Aspect BankInteraction using ICreditCardTransactions
  Attributes
    Constant
      numberId: decimal;
    Variable
      address: String;
      money: decimal;
  Constraints
    static { money >= 0}
  Services
    Begin(input accountId: decimal);
      Valuations
        [in begin(accountId)] numberId = accountId;
    in/out Withdrawal(input quantity: decimal,
                output newMoney: decimal);
      Valuations
        [in Withdrawal(quantity, newMoney)] money = money - quantity;
    in/out Balance(output newMoney: decimal);
      Valuations
        [in Balance(newMoney)] newMoney = money;
    in/out ChangeAddress(input newAdd: String);
      Valuations
        [in ChangeAddress(newAdd)] address = newAdd;
    end;
  Preconditions
    in withdrawal(quantity)
      if quantity <= money;
  Protocols
    BANKINTERACTION = begin.TRANSACTION;
    TRANSACTION = ICreditCardTransactions.Withdrawal?(quantity,
  newMoney).TRANSACTION +
    ICreditCardTransactions.Balance?(newMoney).TRANSACTION +
    ICreditCardTransactions.ChangeAddress?(newAdd).TRANSACTION +
```

```
    end;
End_Functional Aspect BankInteraction;
```

Como puede observarse, se han definido una serie de atributos, uno de los cuales es constante (el número de cuenta, que se define en el momento de creación). Se han definido los servicios correspondientes para sacar dinero de la cuenta (*Withdrawal*), para la consulta de saldo (*Balance*), y para cambiar la dirección (*ChangeAddress*). Obsérvese cómo se ha definido la semántica de dichos servicios mediante la lógica dinámica.

Se ha definido una restricción (sección *constraints*) mediante la cual no se permite extraer dinero si la cuenta no dispone de saldo, y una precondición (sección *preconditions*) para el servicio *Withdrawal* por la cual se establece que sólo se podrá sacar dinero si se dispone de dicha cantidad en la cuenta.

En la sección *Protocols* se han definido dos procesos: BANKINTERACTION, que es el proceso inicial, y TRANSACTION. Obsérvese la sintaxis en Pi-Cálculo Poliádico para indicar que la petición (comportamiento servidor, mediante el símbolo ' ? ') de cualquier servicio (*Interfaz.Nombre_Servicio*) volverá al proceso TRANSACTION.

En segundo lugar se muestra el aspecto de distribución y el aspecto de coordinación. Ambos son importantes, pues formarán parte de los componentes que formarán el sistema que se construirá como ejemplo. La finalidad del aspecto de distribución *ExtMbile* no es más que indicar que el componente que lo utilice puede distribuirse a una nueva máquina.

```
Distribution Aspect ExtMbile using IMobility;
  Attributes
    Location: LOC NOT NULL;

  Services
    begin(input initialLOC: LOC);
      Valuations
        [in begin(initialLOC)] Location = initialLOC;
    in Move(input newLoc:LOC);
      Valuations
        [move(newLoc)] Location:= newLoc;
    end;

  Protocols
    CREATION = begin.EXTMBILE;
    EXTMBILE = IMobility.Move?(newLoc).EXTMBILE + end;
End Distribution Aspect ExtMbile;
```

La finalidad del aspecto de coordinación es coordinar las peticiones entre componentes que demanden el servicio *Withdrawal* o *Balance* al componente que actúe de servidor:

```
Coordination Aspect BankCoordination using ICreditCardTransactions
  Services
    in/out Withdrawal(input quantity: decimal, output money: decimal);
    in/out Balance(output money: decimal);
```

```
    begin;
    end;

  Played_Roles
    CLIENT =
      (ICreditCardTransactions.Withdrawal?(quantity, money)
       →
       ICreditCardTransactions.Withdrawal!(quantity, money) )
    +
      (ICreditCardTransactions.Balance?(money)
       →
       ICreditCardTransactions.Balance!(money) )

    SERVER =
      (ICreditCardTransactions.Withdrawal!(quantity, money)
       →
       ICreditCardTransactions.Withdrawal?(quantity, money) )
    +
      (ICreditCardTransactions.Balance!(money)
       →
       ICreditCardTransactions.Balance?(money) )

  Protocol
    BANKCOORDINATION = begin.COORD;
    COORD =
      (CLIENT.Withdrawal?(quantity, money) →
       SERVER.Withdrawal!(quantity, money) →
       SERVER.Withdrawal?(quantity, money) →
       CLIENT.Withdrawal!(quantity, money)).COORD
      +
      (CLIENT.Balance?(money) →
       SERVER.Balance!(money) →
       SERVER.Balance?(money) →
       CLIENT.Balance!(money)).COORD + end;

End_Coordination Aspect BankCoordination;
```

Con este ejemplo, más complejo que los anteriores, puede observarse la potencia expresiva del *Pi-Cálculo Poliádico*. Este aspecto se utilizará en un conector para coordinar dos componentes, uno que se comportará como cliente y el otro como servidor. Para ello, se han definido dos *played_roles*: uno para el comportamiento CLIENT y otro para el comportamiento SERVER. El *played_role* CLIENT describe el flujo de entradas y salidas para el comportamiento cliente. Por ello, cuando se reciba una petición del servicio *Withdrawal* (flujo de entrada, indicado mediante el símbolo '?') la siguiente acción válida sólo podrá ser devolver el resultado de dicha petición (flujo de salida, indicado mediante el símbolo '!'). En cambio, el *played_role* SERVER modela el otro lado de la comunicación: cuando se reciba un flujo de salida (invocar un servicio), la única acción válida sólo podrá ser un flujo de entrada (obtener los resultados de dicho servicio). Los símbolos '!' deben verse como flujos de *salida* (*output*) y los '?' como flujos de *entrada* (*input*).

Por otra parte, el protocolo modela el proceso global: coordina y sincroniza los servicios de los distintos *played_roles*. En la Figura 19 - Esquema de funcionamiento de un protocolo se puede observar el esquema de funcionamiento de un protocolo de forma gráfica. Cuando llegue una

petición de *Withdrawal* (?) al aspecto de coordinación, la siguiente acción que se realizará es transmitir dicha petición (!) al componente que actúa como servidor (el componente *Banco*). Entonces, el aspecto esperará hasta que se reciba la respuesta por parte del servidor (`SERVER.Withdrawal?`), tras lo cual se retransmitirá al componente cliente (`CLIENT.Withdrawal!`). Una vez se enviase al cliente, el siguiente estado válido sería COORD, y podría procesarse otro servicio diferente.

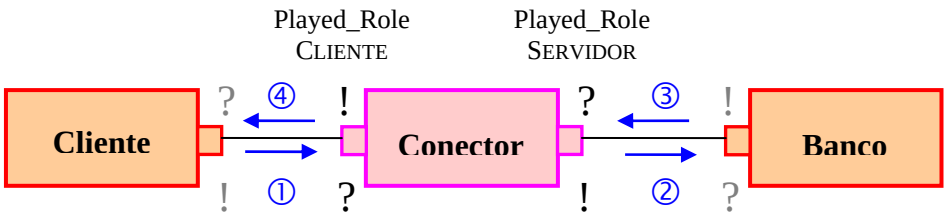

Figura 19 - Esquema de funcionamiento de un protocolo

Si se quisiese permitir la ejecución de otro servicio (como *Balance*) mientras el aspecto de coordinación espera la respuesta del servidor, se indicaría mediante el uso del operador '||' (composición paralela), en lugar del operador '+' (selección no determinista), con lo que se especificaría la ejecución concurrente de servicios.

3.4.1.3 Componentes y Conectores

La plantilla de un componente y de un conector, muy similares entre sí, están formadas por cuatro partes básicas: la cabecera, la definición de puertos o roles de comunicación, los aspectos que lo forman y los *weavings*, que los entretejen entre sí. A continuación se muestra la plantilla genérica, y se describirá cada sección en detalle:

```
1   [Component | Connector]_type <nombre_Componente|Conector>

2   [Port | Role]
      <nombre_i> : <interfaz_i>
3     [Played_Role <nombre_aspecto>.<nom_playedRole_del_aspecto>];
    End_[Port | Role];

4   [Functional | Coordination] Aspect Import <nombre_aspecto_j>;
    <tipo_aspecto_k> Aspect Import <nombre_aspecto_k>;

5   [Weaving
        <nombre_aspecto_k>.<nombre_servicio_k>
            <operador_weaving>
        <nombre_aspecto_n>.<nombre_servicio_n>;
    End_Weaving;]

6   Initialize
      New(<argumentos>) {
        <nombre_aspecto_i>.begin(<args_constructor_i>);
      }
    End_Initialize;
```

```
7   Destruction
      Destroy() {
        <nombre_aspecto_i>.end(<args_destructor_i>);
        <nombre_aspecto_i+1>.end(<args_destructor_i+1>);
      }
    End_Destruction;

    End_[Component | Connector]_type <nombre_Componente|Conector>;
```

Tabla 4 - Plantilla de un componente/conector PRISMA

La cabecera (sección 1) se compone de la palabra reservada *component_type* (para el caso de los componentes) o de *connector_type* (para el caso de los conectores), seguida del nombre que reciba el componente o conector, y que lo identificará unívocamente en la librería PRISMA.

Los puertos de comunicación (o roles en el caso de conectores) se definen en la sección 2, indicando qué interfaz implementan. Además, se puede asociar un played_role al puerto (sección 3) para especificar el comportamiento del conjunto de servicios que forman parte de la interfaz. El played_role pertenece al aspecto indicado mediante la notación punto `<nombre_aspecto>.<nom_playedRole_del_aspecto>`. De esta forma, pueden crearse varios puertos con la misma interfaz pero con played_roles diferentes, con lo que cada puerto con idéntica interfaz tendrá un comportamiento diferente: el especificado por el played_role.

La importación de los aspectos (sección 4) se realiza indicando el tipo y el nombre del aspecto. Al menos debe importarse un aspecto funcional, en el caso de los componentes, o un aspecto de coordinación, en el caso de los conectores. También cabe recordar que no pueden importarse dos aspectos del mismo tipo dentro del mismo componente o conector.

El *weaving* entre los aspectos (sección 5) se define al importar cada aspecto, indicando con qué aspecto van a entretejerse sus servicios. Éstos se declaran especificando el nombre del aspecto en el cual están definidos mediante la notación: `<nombre_aspecto>.<nombre_servicio>`. El *weaving* entre los servicios se realiza mediante `<operador_weaving>`, que indica si la sincronización se realiza *after* (después), *before* (antes) o *instead* (en lugar de), como se mostró en el apartado 3.2.

Por último, las secciones 6 y 7 describen el proceso de creación y destrucción de los componentes/conectores. En la sección *Initialize* se especifica cómo el servicio *New* desencadena la ejecución del servicio *Begin* de los distintos aspectos que forman el componente, mientras que la sección *Destruction* desencadena la destrucción de los aspectos. Además, en el servicio *New* pueden indicarse los parámetros que necesitan los aspectos para su inicialización, que a su vez pueden ser proporcionados por el elemento arquitectónico que cree el componente/conector.

A continuación, se muestran dos ejemplos de componentes: el componente *ATM*, y el componente *Account*, que importa el aspecto funcional definido anteriormente y realiza el *weaving* del servicio *ChangeAddress* con el servicio *Move* del aspecto de distribución:

```
Component_type ATM
   Port
     VISACreditCard_port: ICreditCardTransactions;
     AccountTrans_port: ICreditCardTransactions;
   End_port;

   Functional Aspect import BankInteraction;
   Distribution Aspect import ExtMbile;
   Initialize
     New(accountId: decimal, initialLocation: LOC) {
       BankInteraction.begin(accountId);
       ExtMbile.begin(initialLocation);
     }
   End_Initialize;

   Destruction
     Destroy() {
       BankInteraction.end();
       ExtMbile.end();
     }
   End_Destruction;
End_Component_Type ATM;
```

```
Component_type Account
   Port
     Account_port: ICreditCardTransactions;
   End_port;

   Functional Aspect import BankInteraction;
   Distribution Aspect import ExtMbile;

   Weaving
     BankInteraction.ChangeAddress(newAdd: string)
          before ExtMbile.Move(newAdd: string);
   End_Weaving;

   Initialize
     New(accountId: decimal, initialLocation: LOC) {
       BankInteraction.begin(accountId);
       ExtMbile.begin(initialLocation);
     }
   End_Initialize;

   Destruction
     Destroy() {
       BankInteraction.end();
       ExtMbile.end();
     }
   End_Destruction;
End_Component_Type Account;
```

También se muestra un conector cuya función sería la de coordinar entre sí los dos componentes definidos anteriormente:

```
Connector_type ATMAccount
   Role
     ATM_role: ICreditCardTransactions,
```

```
      Played_Role BankCoordination.CLIENT;
    Account_role: ICreditCardTransactions,
      Played_Role BankCoordination.SERVER;
  End_role;

  Coordination Aspect import BankCoordination;
  Distribution Aspect import ExtMbile;

  Initialize
    New(initialLocation: LOC) {
      BankCoordination.begin();
      ExtMbile.begin(initialLocation);
    }
  End_Initialize;

  Destruction
    Destroy() {
      BankInteraction.end();
      ExtMbile.end();
    }
  End_Destruction;
End_Connector_Type ATMAccount;
```

Como puede observarse, el conector define dos roles con la misma interfaz, pero mediante los played_roles puede distinguir las peticiones de servicio que recibe (comportamiento cliente de los componentes a él conectados) y los servicios ofrecidos (comportamiento servidor). La semántica de coordinación de dichos played_roles vendrá definida por el aspecto de coordinación que lo define.

3.4.1.4 Sistemas, Attachments y Bindings

La plantilla genérica para definir un sistema es la mostrada en la

Tabla 5:

```
1   System_type <nombre_sistema>

2     Ports
        <nom_puerto_i> : <interfaz_i>;
        <nom_puerto_i+1> : <interfaz_i+1>;
      End_Ports

3     Variables
        <nom_Var_i> : <tipo_componente | tipo_conector>;
        ...
      End_Variables

4     Attachments
        <nom_Var_i>.<nom_puerto> ←→ <nom_Var_k>.<nom_rol>;
        ...
      End_Attachments;

5     Bindings
        <nom_Var_i>.<nom_puerto> ←→ <nombre_sistema>.<nom_puerto>;
        ...
      End_Bindings;
```

```
6     Initialize
        New() {
          <nom_Var_i>= new <(nom_componente | nom_conector)>(<args>);
          ...
        }
      End_Initialize;

7     Destruction
        Destroy() {
          <nom_Var_i>.destroy();
          ...
        }
      End_Destruction;
    End_System_type <nombre_sistema>;
```

Tabla 5 - Plantilla de un sistema PRISMA

En la cabecera de un sistema (sección 1) se indica el nombre identificativo de dicho tipo, que lo identificará dentro de la librería de tipos PRISMA. La definición de puertos (sección 2) permite indicar los puertos de comunicación que tendrá dicho sistema con el resto de elementos arquitectónicos. Mediante los *bindings* se establecerá la relación entre los puertos del sistema y los componentes que les darán semántica a los servicios de dichos puertos.

La definición de variables (sección 3) permite especificar los componentes y/o conectores que formarán parte del sistema. Cada componente importado se asociará con una variable, cuyo tipo será el del componente a importar. La construcción/destrucción de dichos componentes se desencadenará en el momento de creación/destrucción del sistema. Estas operaciones se especifican en la sección *Initialize* y *Destruction* (secciones 6 y 7), respectivamente, y permiten indicar los parámetros que necesitará cada componente para su creación.

La definición de un sistema especifica las relaciones de conexión (*attachments*) y composición (*bindings*) entre los elementos arquitectónicos que contiene. Como se definió anteriormente, los *attachments* establecen la conexión entre los puertos de los componentes y los roles de los conectores, mientras que los *bindings* definen la composición entre el sistema y los componentes o subsistemas que contiene. Esto se define en las secciones 4 y 5. Los *attachments* se especifican mediante una relación entre dos variables, cuyos tipos son un componente y un conector, y a través de la notación punto se indica el puerto y el rol que se va a conectar:

```
<variable: tipo_componente>.<puerto> ←→ <variable: tipo_conector>.<rol>
```

Los *bindings* se especifican de la misma forma, sólo que en lugar de definir la relación entre componentes y conectores, se define entre un componente o conector y un sistema.

Como ejemplo, en la siguiente figura se muestra el sistema *BankSystem* que encapsula los componentes y conectores definidos anteriormente, junto con la especificación PRISMA:

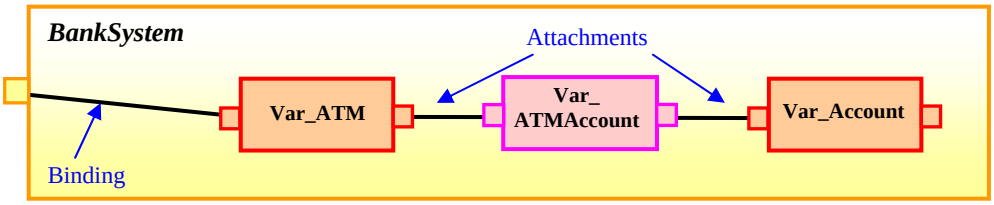

Figura 20 - Ejemplo BankSystem

```
System_type BankSystem
   Ports
      CreditCard_port : ICreditCardTransactions;
   End_Ports

   Variables
      Var_ATM : ATM;
      Var_Account : Account;
      Var_ATMAccount : ATMAccount;
   End_Variables

   Attachments
      Var_ATM.AccountTrans_port ↔ Var_ATMAccount.ATM_role;
      Var_Account.Account_port  ↔ Var_ATMAccount.Account_role;
   End_Attachments;

   Bindings
      Var_ATM.VISACreditCard_port ↔ BankSystem.CreditCard_port;
   End_Bindings;

   Initialize
      New() {
         Var_ATM = new ATM(accountId: decimal, initialLocation: LOC);
         Var_Account=new Account(accountId:decimal,initialLocation:LOC);
         Var_ATMAccount = new ATMAccount(initialLocation: LOC);
      }
   End_Initialize;

   Destruction
      Destroy() {
         Var_ATM.destroy();
         Var_Account.destroy();
         Var_ATMAccount.destroy();
      }
   End_Destruction;
End_System_type BankSystem;
```

Como puede observarse, se ha definido un sistema cuyo único puerto, *CreditCard_port*, está relacionado con el puerto *VisaCreditCard_port* del componente ATM, mediante una relación de *binding*. Además, los componentes que encapsula el sistema están unidos entre sí mediante relaciones de *attachment*, uniendo el puerto *AccountTrans_port* del componente ATM con el rol *ATM_role* del conector ATMAccount, y el puerto *Account_port* del componente *Account* con el rol *Account_role* del conector. Con esto se está especificando que todas las peticiones de servicio que lleguen al sistema se redirijan al componente *ATM*, que decidirá cómo procesar dichas peticiones. A su vez, este componente puede requerir

comunicarse con el componente *Account*, y lo hará a través del conector *ATMAccount*, que coordinará las comunicaciones entre ambos.

3.4.2 Configuración Arquitectónica

El nivel de configuración del lenguaje permite definir las instancias y la topología del modelo arquitectónico final. En la tabla siguiente se puede observar la plantilla genérica, muy semejante a la del sistema visto en el punto anterior:

```
1   Architectural Model <nombre_modelo>
2     Variables
        <nom_Var_i>: <tipo_componente | tipo_conector | tipo_sistema>;
      End_Variables

3     Attachments
        <nom_Var_i>.<nom_puerto> ←→ <nom_Var_k>.<nom_rol>;
      End_Attachments;

4     Initialize
        New() {
          <nom_Var_i>= new <(nom_componente | nom_conector)>(<args>);
        }
      End_Initialize;

5     Destruction
        Destroy() {
          <nom_Var_i>.destroy();
        }
      End_Destruction;
    End Architectural Model <nombre_modelo>;
```

Tabla 6 - Plantilla de un modelo arquitectónico PRISMA

Como se puede observar, las secciones que forman parte de un modelo arquitectónico son muy similares a las de los sistemas. La cabecera (sección 1) define el nombre del modelo arquitectónico. La cláusula *Variables* (sección 2) define los componentes, conectores o sistemas que forman el modelo arquitectónico, que a la vez que importa los tipos de la librería de PRISMA, las asocia con un nombre de variable para posteriormente poderles hacer referencia en las siguientes secciones. La especificación de *Attachments* (sección 3) especifica, al igual que en los sistemas, la conexión entre los diferentes elementos arquitectónicos que conforman el modelo.

El bloque *Initialize* (sección 4) especifica cómo instanciar los elementos arquitectónicos que conforman el modelo, proporcionando los valores de inicialización que requieran sus constructores. Pueden seguirse dos aproximaciones. La primera de ellas consiste en especificar en el modelo arquitectónico las instancias que lo forman, proporcionando los correspondientes valores de inicialización. El problema que plantea esta aproximación es que cuando se quieran cambiar los valores de inicialización

deberá modificarse la especificación del modelo arquitectónico. La otra aproximación consiste en definir el modelo arquitectónico como un tipo, es decir, no especificar los valores de inicialización que tomarán las instancias, tan sólo la secuencia de llamadas a los constructores de los elementos arquitectónicos que forman el modelo arquitectónico. Más tarde, cuando se decidiese instanciar el modelo arquitectónico definido, se proporcionarían los valores de inicialización requeridos (ver ejemplo). Con esto, se favorece la reutilización del modelo arquitectónico definido. Además, la inicialización puede ser anidada, es decir, pueden especificarse los valores de inicialización de componentes encapsulados dentro de sistemas, con el objetivo de evitar que los constructores de dichos sistemas acumulen una larga lista de inicialización.

Por último, el bloque de destrucción (sección 5) define qué secuencia de destrucción de los componentes debe seguirse, y si es necesario, indicar servicios adicionales que deban especificarse, como por ejemplo, la persistencia de los datos.

Como ejemplo de modelo arquitectónico, va a tomarse únicamente el sistema definido en el apartado anterior:

```
Architectural Model SimpleBankSystem
   Variables
      Var_BankSystem : BankSystem;
   End_Variables

   Initialize
      New() {
         Var_BankSystem = new BankSystem();
      }
   End_Initialize;

   Destruction
      Destroy() {
         Var_BankSystem.destroy();
      }
   End_Destruction;
End Architectural Model SimpleBankSystem;
```

Como se puede observar en el ejemplo, la definición del modelo arquitectónico está formada únicamente por un único sistema, el definido anteriormente. El bloque de inicialización únicamente define cómo construirlo, pero no lo instancia, con el objetivo de favorecer la reutilización del modelo arquitectónico. La instanciación del modelo arquitectónico se haría de la siguiente forma:

```
BANKSYSTEM = new BankSystem {
   ATM1 = new ATM(123456789, new LOC("tcp://garbi.dsic.upv.es"));
   ACCOUNT1= new Account(987654321,new LOC("tcp://sigil.dsic.upv.es"));
   ATMAccount1 = new ATMAccount(new LOC("tcp://garbi.dsic.upv.es"));
}
```

Obsérvese cómo se han ido anidando los constructores para inicializar los componentes que forman el sistema. El *Framework* de PRISMA sería el encargado, mediante reflexión de código, de obtener los componentes que

forman el sistema y de pedir al usuario los parámetros de inicialización necesarios, bien de forma gráfica mediante interfaces de usuario, o bien por consola. La otra alternativa consistiría en definir el constructor del sistema final con tantos argumentos como argumentos tuvieran cada uno de los constructores de los componentes/conectores que forman el sistema.

3.5 Otros aspectos avanzados de PRISMA

La especificación del modelo PRISMA abarca otras áreas que no se han cubierto en este capítulo por no incluirse entre los objetivos del presente proyecto. No obstante, la especificación PRISMA también incluye un lenguaje gráfico, basado en UML 1.5, para diseñar la arquitectura del sistema de forma amigable para el usuario. Pueden encontrarse más detalles en [Per03]. En la actualidad, otras líneas de trabajo están definiendo plugins gráficos para diseñar modelos PRISMA en la herramienta Microsoft® Visio 2003, aunque en un futuro también se diseñarán para Rational Rose® y ArgoUML.

Por otra parte, una característica importante que aporta PRISMA es que está concebido para permitir la evolución del modelo en tiempo de ejecución: el modelo arquitectónico podrá reconfigurarse dinámicamente sin necesidad de detener la ejecución de la aplicación. Podrá evolucionar tanto la estructura como el comportamiento de los distintos elementos arquitectónicos que forman el modelo PRISMA, mediante la definición de un metanivel para convertir los componentes en datos y poderlos modificar.

Finalmente, también existe otra línea de investigación que se encarga del estudio de las propiedades de distribución del modelo y de extenderlo para permitir la movilidad, replicación y comunicación distribuida de los elementos arquitectónicos. Para ello, se han definido patrones de distribución que describen situaciones en las cuales la arquitectura software necesita reconfigurar su topología de localización dinámicamente bien moviendo o reemplazando sus elementos arquitectónicos para hacer frente a problemas de distribución, nuevos requerimientos del sistema o otros cambios en el rendimiento de la arquitectura software. Entre los múltiples patrones que se recogen en [Ali03], a continuación se citan algunos: Invocación de servicio, exceso de invocación de un servicio, exceso del volumen de datos intercambiado con un origen, exceso del tiempo de latencia de un mensaje, etc.