Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos

Cristóbal Costa Soria

Tesis de Máster

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información
Departamento de Sistemas Informáticos y Computación

Directores

Dr. Isidro Ramos Salavert
Dr. José A. Carsí Cubel

Septiembre de 2007, Valencia

Este trabajo ha sido financiado por el CICYT (Comisión Interministerial de Ciencia y Tecnología) Proyecto META TIN2006-15175-C05-01. También ha sido financiado parcialmente gracias a una beca FPI (Formación de Personal Investigadore) de la Conselleria d'Educació i Ciència de la Generalitat Valenciana concedida a Cristóbal Costa.

Agradecimientos

A Sagrario, por su amor, comprensión y apoyo incondicional.
Sin ella, nada para mí tiene sentido.

A mi madre y a mi hermano, por su paciencia conmigo
en los momentos difíciles y su apoyo.

A mi familia.

A Jennifer, Pepe e Isidro, por haber depositado su confianza,
por compartir sus conocimientos y consejos.

A Abel, Alejandro, Carlos, Javi, Jose Antonio, Elena, Manolo y Nour,
gracias por estar ahí y por hacerme pasar tan buenos ratos.

A todos los miembros del grupo de investigación ISSI,
por su atención y colaboración.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPÍTULO I

INTRODUCCIÓN

El trabajo presentado en esta tesis de máster persigue dotar a los sistemas software altamente disponibles o a aquellos que realizan misiones críticas y que por tanto no pueden ser detenidos, de la posibilidad de modificar tanto su estructura como su comportamiento sin afectar al resto de subsistemas en ejecución. En concreto, este trabajo se enmarca dentro del área de las arquitecturas software y es por tanto a este nivel en el que se proporcionan capacidades de evolución dinámica.

Esta propuesta proporciona dos grados de adaptabilidad a las arquitecturas software. Por una parte, permite especificar cómo una arquitectura software puede modificar su configuración en tiempo de ejecución en respuesta a distintas situaciones, como cambios en su entorno o como a consecuencia de la lógica de negocio del sistema. Por otra parte, esta propuesta permite describir cómo una arquitectura software puede evolucionar sus tipos arquitectónicos dinámicamente.

La estructura de este capítulo es la siguiente. En primer lugar se describe cuál es la motivación de este trabajo de investigación. En segundo lugar se presentan los objetivos del trabajo. En tercer lugar se describe la metodología de investigación seguida. En cuarto y último lugar, se presenta brevemente cada uno de los capítulos restantes.

1.1 Motivación del trabajo

1.1.1 La importancia del mantenimiento del software en los sistemas complejos

Los sistemas software cada vez son más complejos. A la vez que la tecnología ha ido evolucionando y proporcionando mayores prestaciones, los

sistemas software han incrementado su funcionalidad, y por tanto, su complejidad. Además de los requisitos funcionales del propio sistema, cada vez se demandan más requisitos no funcionales, como que el sistema sea distribuido y/o descentralizado, seguro, fiable y tolerante a fallos, altamente disponible, etc. Mayor complejidad del software tiene como consecuencia mayores tiempos de desarrollo, mayores tiempos de mantenimiento, y por tanto mayores costes económicos.

Una forma de abordar la complejidad del desarrollo de sistemas software es descomponiéndolos en subsistemas funcionales interconectados entre sí. Para facilitar su mantenimiento, dichos subsistemas deben ser reutilizables e independientes, de tal forma que las dependencias entre ellos sean mínimas y el mantenimiento de uno de ellos no afecte al resto. El enfoque de Desarrollo de Software Basado en Componentes (Component-Based Software Development – CBSD [DSo98], [Szy98]) denomina a cada uno de estos subsistemas *componentes*. La especificación de *qué* componentes integran el sistema software a desarrollar y *cómo* deben interconectarse es la denominada *Arquitectura Software* [Per92], [Gar93]. De esta forma, un sistema software se construye mediante el ensamblado e interconexión de distintos *componentes* (cada uno de los cuales puede haber sido desarrollado por distintos proveedores de software).

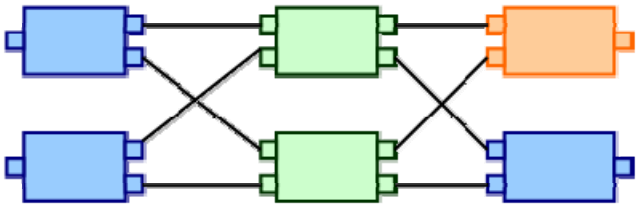

Figura 1 – Arquitectura de Software

Además de la dificultad de construir sistemas complejos, también se debe hacer frente a la dificultad de mantener dichos sistemas. Es inevitable que el sistema software, una vez en funcionamiento, no requiera ser modificado para corregir errores o incorporar nuevas características. Por una parte, a mayor complejidad, mayor es la probabilidad de que surjan errores a posteriori y que deben ser reparados. Por otra parte, como es imposible anticipar a priori todas las características que el sistema va a requerir en un futuro, posiblemente también deberá modificarse el sistema para añadir nuevas características. La etapa de mantenimiento de un sistema software es posiblemente la más importante, dado que estará presente durante toda la duración de la vida útil del sistema software. Desde el principio, los sistemas software complejos deben diseñarse para facilitar su futuro mantenimiento. Deben ser lo suficientemente flexibles para tolerar modificaciones posteriores.

Gran parte de los problemas derivados del mantenimiento del software son a consecuencia de que el código se encuentra entremezclado: muchos bloques

de código con una funcionalidad concreta (como la gestión de las comunicaciones remotas, o las conexiones con bases de datos) se encuentran repetidos y diseminados entre el código del sistema, de tal forma que cambiar dicho código tiene un alto coste. El enfoque de Desarrollo de Software Orientado a Aspectos (Aspect-Oriented Software Development – AOSD [Kiz97], [Aosd]) propone la separación de estos bloques de código entremezclado en entidades separadas denominadas *aspectos*. Dichos aspectos facilitan el mantenimiento y la reutilización, ya que son altamente independientes.

Los enfoques de desarrollo de software anteriormente citados (CBSD, Arquitecturas Software y AOSD) ofrecen las siguientes ventajas:

- Facilitan la tarea de desarrollar sistemas software complejos, puesto que permiten describir el software a un alto nivel de abstracción, en términos de *componentes* y sus interrelaciones.
- Facilitan el mantenimiento y reutilización de las entidades que componen el sistema software, puesto que éstas se definen de forma independiente, minimizando el número de dependencias con el resto de entidades del sistema.
- Facilitan la evolución del software, puesto que mediante el uso de aspectos se facilita la adaptación del código existente con nueva funcionalidad.

1.1.2 Desarrollo de software dirigido por modelos

Los enfoques presentados en las secciones anteriores proponen distintos modelos para la construcción de sistemas software: mediante el uso de aspectos (AOSD) o mediante el uso de componentes (CBSD). El enfoque de Desarrollo de Software Dirigido por Modelos (Model-Driven Development – MDD [Bey05], [Am04]) va más allá: amplía el concepto de modelo para que además de ser una herramienta conceptual para la representación de los sistemas software, permita también la generación automática de código. Esto se consigue mediante la combinación de distintos modelos, que describen distintas vistas del sistema software. MDD está incluido dentro de la propuesta de la Ingeniería Dirigida por Modelos [Sch06], que aumenta la variedad de artefactos software que pueden representarse como modelos (ontologías, modelos UML, esquemas relacionales, etc.)

La ventaja del uso de modelos para el desarrollo de software es que permiten describir sistemas software a un alto nivel de abstracción, facilitando su comprensión y eliminando detalles de implementación de bajo nivel. Además, estos modelos son independientes de la tecnología, por lo que dichos modelos pueden transformarse a las distintas tecnologías existentes

(como Java, .NET, sistemas empotrados) y lenguajes de programación (C#, Java, C++, etc.)

En la comunidad investigadora aún no existe un consenso acerca de qué modelos deberían ser utilizados para la descripción completa de un sistema software, y por tanto para la generación de código. Cada propuesta presenta diferentes modelos mediante los cuales describir y construir sistemas software, con sus ventajas e inconvenientes.

El grupo de investigación en el que se enmarca este trabajo cuenta con una larga experiencia en el campo de la generación automática de software a partir de modelos. En particular, la propuesta PRISMA [Per06] proporciona un enfoque MDD que combina las ventajas del Desarrollo de Software Orientado a Aspectos (AOSD) con las ventajas de la descripción arquitectónica de sistemas software (Arquitecturas Software). PRISMA ofrece un soporte completo para el desarrollo de arquitecturas software orientadas a aspectos independientes de la tecnología, puesto que las descripciones de arquitecturas software PRISMA pueden ser compiladas a distintas plataformas tecnológicas y lenguajes de programación mediante el uso de técnicas de generación de código. La aportación de este trabajo se basa en arquitecturas software PRISMA, con el objetivo de beneficiarse de las ventajas proporcionadas por el desarrollo de software orientado a aspectos.

1.1.3 Evolución del software en tiempo de ejecución

Pese a las ventajas que los enfoques anteriores proporcionan para facilitar la descripción de un sistema software, su mantenimiento y su reutilización, la modificación de un sistema tras su implantación es una tarea costosa, debido a que los cambios deben instalarse en el sistema una vez ya está en funcionamiento y detenerlo no siempre es posible. Tradicionalmente, el mantenimiento (o evolución) del software se ha realizado mediante la modificación del código fuente afectado (modelos en el caso de enfoques MDD), su compilación y posterior reinicio (total) del sistema con los cambios incorporados.

Hasta hace dos décadas la totalidad de los sistemas implantados en la industria eran mayoritariamente estáticos. Los sistemas de naturaleza dinámica eran muy difíciles de desarrollar y mantener, por lo que tan sólo existían algunos prototipos en el ámbito de la investigación. En la actualidad aún predominan los sistemas estáticos, pero poco a poco, a medida que la tecnología es más potente, los sistemas de naturaleza dinámica se van extendiendo: son más flexibles y soportan mejor los cambios. Aunque su desarrollo sigue siendo muy complejo, existen numerosos prototipos que demuestran la viabilidad de la construcción de dichos tipos de sistemas.

Los sistemas con necesidades dinámicas pueden ser de varios tipos [Cue02]:

- *Sistemas Multi-Agente y Sistemas Distribuidos capaces de expresar algún grado de movilidad.*

  Son sistemas que por su naturaleza móvil, modifican frecuentemente las conexiones entre sus elementos.

- *Sistemas Auto-organizados.*

  Modifican tanto las conexiones entre sus elementos como los elementos que los componen para adaptarse a situaciones concretas. Dichos cambios son especificados en tiempo de diseño. Como ejemplo, están aquellos sistemas capaces de recuperarse frente a fallos o capaces de configurarse en función del sistema en el que se instalan.

- *Sistemas Continuos, de Tiempo Real y/o Tolerantes a Fallos.*

  Son sistemas que pueden requerir cambios frecuentemente, pero no pueden ser detenidos para su actualización. También son denominados en la literatura como *Software-Intensive Systems*. Como ejemplo de este tipo de sistemas, están aquellos que se encargan del control de infraestructuras críticas, como las militares, la energía, relativas a la salud, telecomunicaciones o transporte. También se incluyen dentro de esta categoría los sistemas empotrados para campos específicos de la industria (automoción, aviónica, etc.)

- *Sistemas Abiertos*

  Su estructura cambia de forma constante. Pueden añadirse nuevos elementos no contemplados en tiempo de diseño para extender o modificar su funcionamiento. Un ejemplo de este tipo de sistemas son los sistemas operativos, puesto que permiten "introducir" nuevos elementos para extender su funcionamiento.

Es por la demanda creciente de este tipo de sistemas que surge la necesidad de incorporar el soporte a la evolución en tiempo de ejecución. El objetivo es que los sistemas software sean capaces de soportar cambios en su arquitectura sin necesidad de detenerse completamente. Para ello, los subsistemas afectados deberán poder ser modificados en tiempo de ejecución, mientras que el resto de subsistemas serán ajenos a dicho proceso y podrán continuar trabajando normalmente.

Además de la necesidad de incorporar dinamismo en los sistemas software, también es necesario poder *modelar* dicho dinamismo. Muchas de las metodologías de desarrollo de software actuales ya incorporan técnicas de generación automática de código a partir de modelos (según el enfoque MDD). Sin embargo, pocas propuestas incorporan algún grado de dinamismo en los sistemas software generados, siendo incorporado posteriormente y de forma manual. Para la descripción completa de un sistema software dinámico, deben modelarse también las capacidades y

necesidades de dinamismo del sistema a generar, aspecto que apenas ha sido tratado en el campo del MDD.

1.2 Objetivos del trabajo

El principal objetivo de este trabajo es proporcionar a sistemas software complejos la capacidad de evolucionar en tiempo de ejecución, comprendiendo tanto la modificación de su topología como la modificación de los elementos que la integran. Esto se ha realizado mediante el uso de aspectos, por las ventajas que éstos proporcionan para la adaptación y mantenimiento del software.

La evolución de sistemas software puede aplicarse a distintos niveles de granularidad, que varía desde un nivel de granularidad más amplio (e.g. cambios que afectan a nivel de arquitecturas software), un nivel medio (e.g. cambios que afectan a nivel de clases) o un nivel más fino (e.g. cambios que afectan a métodos, variables o líneas de código). Cuanto más amplio es el nivel de granularidad, mayor es el nivel de abstracción y por tanto mayor independencia de la plataforma de desarrollo. En este trabajo se abordará la evolución de sistemas software desde el nivel de las arquitecturas software, debido a que niveles más finos son, o bien dependientes del lenguaje de desarrollo elegido (el nivel medio), o bien dependientes de la tecnología (el nivel fino).

Específicamente, este trabajo toma como base arquitecturas software PRISMA [Per06], cuyos componentes son descritos mediante aspectos. Partiendo de esta base, se garantiza que el mantenimiento y reutilización de dichos sistemas es más sencilla que con otras aproximaciones.

Este trabajo pretende dos objetivos principales. Por una parte, facilitar el modelado y especificación de arquitecturas software dinámicas y adaptables, en vistas a su incorporación futura a una herramienta, ya existente, de desarrollo dirigido por modelos (MDD). Por otra parte, determinar qué mecanismos deben ser modelados y cuáles deben ser proporcionados por la tecnología.

Se persigue dotar a las arquitecturas software de dos grados de dinamismo:

- *Reconfiguración Dinámica.*

  Capacidad de modificar su topología (configuración) en tiempo de ejecución, lo que implica: (i) crear/destruir instancias de elementos arquitectónicos (componentes/conectores), y (ii) crear/destruir conexiones entre elementos arquitectónicos. Esta capacidad es fundamental para la construcción de sistemas auto-organizados y sistemas móviles.

- *Evolución Dinámica de Tipos Arquitectónicos*.

  Capacidad de modificar la descripción de la arquitectura en tiempo de ejecución, lo que implica: (i) añadir/eliminar tipos de elementos arquitectónicos, (ii) añadir/eliminar tipos de conexiones entre elementos arquitectónicos, y (iii) modificar tipos de elementos arquitectónicos mientras sus instancias se encuentran en ejecución. Este es el grado de dinamismo más potente. Esta capacidad es necesaria para la construcción de sistemas abiertos y sistemas continuos y de tiempo real.

1.3 Metodología de investigación

La metodología de trabajo aplicada para satisfacer los objetivos propuestos en este trabajo de investigación sigue una estrategia metodológica clásica llamada "estrategia de investigación por alcanzabilidad". Esta metodología parte de una hipótesis genérica y conceptual que se presenta como una contribución en el área en que el trabajo de investigación ha sido desarrollado. Esta hipótesis se basa en el análisis previo del estado del arte en el que la contribución del trabajo está justificado.

Este trabajo parte de la siguiente hipótesis:

> *¿Facilitan los conceptos de desarrollo orientado a aspectos el modelado y construcción de arquitecturas software dinámicas y adaptables?*

Desde este punto de inicio, el principal objetivo del trabajo de investigación es estudiar si mediante el uso de los conceptos de separación de *concerns* es más sencillo describir sistemas software dinámicos y adaptables, facilitando el mantenimiento de dichos sistemas.

1.4 Organización

Este trabajo se ha organizado en cinco capítulos. A continuación se describe brevemente el contenido de cada uno de los capítulos siguientes:

- Capítulo 2: Contexto

  Este capítulo describe el contexto en el que se enmarca este trabajo: las arquitecturas software, el desarrollo de software orientado a aspectos (AOSD) y el enfoque PRISMA.

- Capítulo 3: Reconfiguración dinámica de arquitecturas software

Este capítulo describe cómo soportar la reconfiguración dinámica de arquitecturas software a través de técnicas de desarrollo de software orientado a aspectos. Por una parte, se presenta cómo especificar en un lenguaje de descripción arquitectónica un sistema software capaz de modificar su topología en tiempo de ejecución. Por otra parte, se describe qué mecanismos debe proporcionar la infraestructura de ejecución para soportar la reconfiguración dinámica de arquitecturas software.

- Capítulo 4: Evolución dinámica de tipos arquitectónicos

  Este capítulo describe cómo soportar la evolución dinámica de tipos arquitectónicos, es decir, la modificación en tiempo de ejecución de la descripción de los elementos arquitectónicos (los tipos) y de todas sus instancias. Por una parte, se describe cómo poder especificar en un lenguaje de descripción de arquitecturas un sistema software capaz de evolucionar sus tipos. Por otra parte, se describen los mecanismos necesarios que debe proporcionar la infraestructura de ejecución para soportar esta adaptabilidad dinámica.

- Capítulo 5: Conclusiones y trabajos futuros

  Este capítulo presenta las conclusiones de este trabajo y los trabajos futuros que pretenden abordarse tanto a corto como a largo plazo.

CAPÍTULO II

CONTEXTO

2.1 Arquitecturas Software

El incremento en el tamaño y complejidad de los sistemas software actuales ha llevado a la comunidad informática a reconocer el análisis de la estructura del software como una etapa importante en el ciclo de vida del software. Es por esto que, como resultado de las últimas décadas, una nueva área de investigación denominada *Arquitectura Software* ha emergido para tratar específicamente esta fase del desarrollo del software.

Un diseño inapropiado de la arquitectura de un sistema software conduce al fracaso de los sistemas software grandes y complejos. Por esta razón, el diseño, la especificación y el análisis de la estructura de estos sistemas se ha convertido en un aspecto crítico del desarrollo de software [Gar01]. Las arquitecturas software se presentan como una solución para el diseño y el desarrollo de este tipo de sistemas, permitiendo describir la estructura de un sistema software ocultando los detalles de bajo nivel y abstrayendo a un alto nivel las características importantes [Per92]. Esta estructura es normalmente representada en términos de elementos computacionales y sus interacciones. Como resultado, las arquitectuas software hacen más sencillos y comprensibles los sistemas software [Gar95].

La disciplina de arquitectura software permite realizar las siguientes funciones: analizar y describir las propiedades de los sistemas a un alto nivel de abstracción, validar los requisitos software, estimar el coste de los procesos de desarrollo y mantenimiento, reutilizar software, y establecer las bases y guías para el diseño de los sistemas software complejos [Per92]. Al mismo tiempo, las arquitecturas software deben ser adaptables y proporcionar soporte para la reutilización de los elementos arquitectónicos y para la descripción parcial o total de las especificaciones de las arquitecturas software. De este modo, los nuevos diseños no empiezan desde cero y sólo las características específicas de los nuevos sistemas son creados desde cero [Per92].

Sin embargo, a pesar del intento del IEEE para estandarizar la disciplina de arquitecturas software [IEE00], aún no se ha logrado llegar a un consenso sobre la definición de arquitectura software y sobre los distintos conceptos a utilizar en este campo. Existen distintas definiciones del concepto de arquitectura software, muchas de ellas genéricas y no excluyentes pero también incompletas, poco explícitas e imprecisas. Una de las primeras definiciones fue la de Perry y Wolf:

> *"Una arquitectura software es un conjunto de elementos arquitectónicos (o diseños) con una forma particular"* [Per92]

El Software Engineering Institute (SEI) propone la definción de Garlan y Perry como la más indicada para el concepto de arquitectura software:

> *"La estructura de los componentes de un programa/sistema, sus interrelaciones, y principios y guías que gobiernan su diseño y evolución a lo largo del tiempo"* [Gar95]

Por su parte, el estándar ANSI/IEEE 1471-2000 propone una variación de la definición original de Garlan y Perry:

> *"Arquitectura es definida por la práctica habitual como la organización fundamental de un sistema, expresada en términos de componentes, las relaciones entre ellos y el entorno, y los principios que gobiernan su diseño y evolución".* [IEE00]

Ambas definiciones son sencillas y breves, pero carecen de las propiedades de completitud y exactitud. Finalmente, otra definición general es la propuesta por D'Souza:

> *"Una arquitectura es una abstracción de un sistema que describe las estructuras de diseño y relaciones, la reglas que lo rigen, o los principios que son (o pueden ser) utilizados en varios diseños."* [Dso98]

Las descripciones de arquitecturas software son especificadas de una manera formal mediante el uso de Lenguajes de Descripción de Arquitecturas (Architecture Description Languages, ADL). A pesar de la diversidad de los distintos ADL propuestos hasta la fecha, la mayoría de ellos comparten un conjunto de elementos común para diseñar la estructura de sistemas software. Los elementos que proporcionan una base común para la descripción de arquitecturas software son introducidos a continuación.

2.1.1 Componente

El concepto de *componente* es la base para las arquitecturas software y el que todos los ADL comparten. Un componente es un elemento computacional que permite a los usuarios estructurar la funcionalidad de los sistemas software. Proporcionan un alto nivel de encapsulación y sólo es

posible interactuar con ellos a través de sus interfaces. La mayoría de los ADLs permiten definir más de una interfaz para cada componente. La *interfaz* o las múltiples interfaces de un componente define(n) la funcionalidad que el componente requiere y proporciona. En este sentido, los componentes son considerados como cajas negras.

El concepto de componente no es solamente usado en el campo de las arquitecturas software. Por esta razón, muchas veces es difícil conocer el significado exacto del término, y no existe un consenso acerca de la definición de componente y cómo identificar los componentes que conforman un sistema software. Existen dos tendencias: una es orientada a implementación y la otra es más genérica. La primera engloba aquellas definiciones que relacionadas con el concepto de que un componente es paquete de código [DSo98], mientras que la segunda define un componente como un artefacto diseñado para ser reutilizado. Esta seguna definición es abstracta y genérica, por lo que un componente puede ser un caso de uso, una clase o cualquier otro elemento que emerja durante el proceso de desarrollo. La definición de componente en el campo de la arquitectura software se encuentra también en esta categoría.

Existen numerosas definiciones de componente, pero la más extendida es la propuesta por Szyperski:

> *"Un componente software es una unidad de composición con solamente interfaces especificadas contractualmente y dependencias del contexto explícitas. Un componente software puede ser utilizado independientemente y es susceptible de composición por terceras partes."* [Szy98]

Otra definición muy extendida es la de Bertrand Meyer:

> *"Un componente es un elemento software (unidad modular) que satisface las tres condiciones siguientes:*
>
> - *Puede ser utilizado por otros elementos software, sus "clientes"*
> - *Posee una descripción oficial de uso, que es suficiente para un cliente para utilizarlo*
> - *No está atado a un conjunto fijo de clientes".* [Mey03]

Por último, la definición de D'Souza proporciona también una definición genérica de componente, aunque apuesta por las tendencias orientadas a implementación:

> *"Un componente es un paquete coherente de artefactos software que pueden ser desarrollados independientemente y desplegados como una unidad, y que pueden ser compuestos con otros componentes para construir algo más grande".* [DSo99]

2.1.2 Conector

El concepto de *conector* emerge de la necesidad de separar la interacción de la computación y con ello obtener componentes más reutilizables y modulares y mejorar el nivel de abstracción de las descripciones de las arquitecturas software.

Los conectores representan las interacciones de los sistemas software. Definen el proceso de coordinación entre componentes, esto es, las reglas que guían la interacción entre componentes. Las interfaces son el modo de interactuar con ellos y dichas interfaces representan los roles en que cada uno de los componentes participa en el proceso de coordinación. En su trabajo, Mary Shaw [Sha94] presenta la necesidad de emplear conectores debido al hecho de que la especificación de sistemas software mediante protocolos de coordinación complejos es muy difícil sin la noción de conector. Además, defiende la idea de considerar conectores como ciudadanos de primer orden de los ADLs, definiendo el concepto de conector de esta forma:

> *"Los conectores son los lugares de las relaciones entre componentes. Median interacciones, pero no son "cosas" a conectar (ellos son, mejor dicho, los que conectan). Cada conector tiene una especificación de protocolo que define sus propiedades. Estas propiedades incluyen reglas sobre los tipos de interfaces que son capaces de mediar, garantías acerca de las propiedades de la interacción, reglas acerca del orden en que las cosas ocurren, y compromisos acerca de la interacción como ordenación, rendimiento, etc."* [Sha94]

Sin embargo, otras aproximaciones y sus respectivos ADL (como Darwin [Mag95] o Leda [Can01]) prefieren la ausencia del concepto de conector porque distorsionan la natualeza composicional de las arquitecturas software. Dichas aproximaciones emplean el concepto de conector como la mera conexión entre dos componentes, y por tanto no lo definen como ciudadano de primer orden. No obstante, progresivamente el concepto de conector está siendo aceptado por la comunidad y cada vez más trabajos acerca de conectores complejos están emergiendo.

2.1.3 Puerto

El concepto de *puerto* está relacionado con los elementos arquitectónicos, componentes y conectores. Los puertos son los puntos a través de los cuales los elementos arquitectónicos pueden interaccionar con el resto de la arquitectura software. Son las partes en las que la interfaz de un elemento arquitectónico se divide. Su principal función es preservar la vista de caja negra de los elementos arquitectónicos y publicar el comportamiento ofrecido y requerido por los elementos arquitectónicos. Han sido utilizados de diferentes formas: algunas aproximaciones consideran un puerto como un servicio y otras como un proceso con distintos servicios. Esta última forma

de definir los puertos no solamente define los servicios de los puertos, sino que también las condiciones de cómo y cuándo los servicios pueden ser requeridos y proporcionados.

2.1.4 Conexión

Las *conexiones* son utilizadas para restringir cómo los distintos elementos arquitectónicos pueden interaccionar y cómo son organizados respecto al resto de elementos de la arquitectura [Per92]. Establecen los canales de comunicación entre los elementos arquitectónicos: conectan el puerto de un componente con el puerto de un conector o de otro componente, dependiendo si los conectores son considerados ciudadanos de primer orden o no, respectivamente. En algunos ADL, las conexiones entre elementos arquitectónicos son denominadas *attachments*.

2.1.5 Sistema

Muchas aproximaciones arquitectónicas necesitan proporcionar mecanismos de abstracción. Dichos mecanismos permiten definir elementos de una mayor granularidad e incrementar la modularidad, composición y reutilización de los sistemas software. La composición del software permite reducir la complejidad del proceso de desarrollo de sistemas software [Nie95].

Estas necesidades y ventajas han conducido a una amplia variedad de modelos arquitectónicos y sus respectivos ADL a proporcionar el concepto de componente complejo. Un componente complejo es un componente que está compuesto por otros elementos arquitectónicos. Ejemplo de ello son los modelos propuestos por Darwin [Mag95], ACME [Gar00], o ArchWare ADL [Oqu04a], [Oqu04b]. Son denominados *sistemas*.

Los sistemas representan configuraciones arquitectónicas constituidas por conectores y componentes dispuestas de una forma jerárquica. Por esta razón, un sistema puede estar compuesto de otros subsistemas [And03].

2.1.6 Relaciones de composición

Las *relaciones de composición* emergen con los sistemas, debido al hecho de que es necesario para los sistemas comunicarse con sus elementos arquitectónicos. Estas conexiones son distintas a las proporcionadas por los attachments, debido a que son usadas para conectar elementos arquitectónicos de distintos niveles de granularidad. Como resultado, la semántica de dichas conexiones es composicional, mientras que la de los

attachments no lo es (conectan elementos del mismo nivel de granularidad). Estas relaciones son denominadas *bindings*.

Los bindings establecen los mappings entre las interfaces internas y externas de un sistema [Gar01]. Como resultado, los bindings establecen una conexión entre el puerto del sistema y el puerto de uno de sus elementos arquitectónicos.

2.2 Desarrollo de Software Orientado a Aspectos

La Programación Orientada a Aspectos (*Aspect Oriented Programming, AOP)* surge como solución a ciertos problemas detectados en el modelo de programación orientado a objetos (POO). En el paradigma POO, toda tarea específica debe ser responsabilidad de una clase o de un pequeño número de clases agrupadas de alguna forma lógica. Sin embargo, existen ocasiones en las que determinados servicios se utilizan en el seno de diversas clases y no tienen suficiente entidad para incluirlos en una clase específica, lo que acaba provocando repetición de código a lo largo de toda la aplicación. Ejemplos de estos servicios o bloques de código son los dedicados a la sincronización o a la optimización de los accesos a los recursos, a la persistencia de los datos, al tratamiento de excepciones, al registro de auditorías (logs), etc. Todos estos servicios o bloques de código son características o temas de interés dentro del sistema software (*concern*). IEEE define el concepto de *concern* como:

> *"…aquellos intereses que pertenecen al desarrollo del sistema, su operación o cualquier otro aspecto que es crítico o importante para uno o más stakeholders"* – ANSI/IEEE Std 1471-2000 [IEE00]

Usualmente estos *concerns* se encuentran diseminados entre distintos artefactos software (por ejemplo, en la POO, estos artefactos serían clases), dando lugar a los denominados *crosscutting concerns* (asuntos de interés entremezclados). Los *crosscutting concerns* son asuntos de interés que se encuentran repetidos en todos aquellos artefactos software a los que afectan, y además se entremezclan con otros *concerns* que también afectan al mismo artefacto software. Esto aumenta el volumen del código y dificulta el mantenimiento de aquellos artefactos software afectados por distintos *concerns*. Además, al estar los *concerns* diseminados por el sistema, también se dificulta el mantenimiento de estos *concerns*.

AOP propone encapsular cada *crosscutting concern* en una entidad separada, el *aspecto*, de forma que se localicen los cambios relacionados con un *concern*. Esta separación evita el código entremezclado y permite la reutilización del mismo aspecto en diferentes artefactos software (objetos,

componentes, módulos, etc.) En la Figura 2 puede observarse cómo el agrupar los diferentes *crosscutting concerns* en entidades separadas favorece el mantenimiento y la evolución de dicha funcionalidad a consecuencia de una mayor modularización del código.

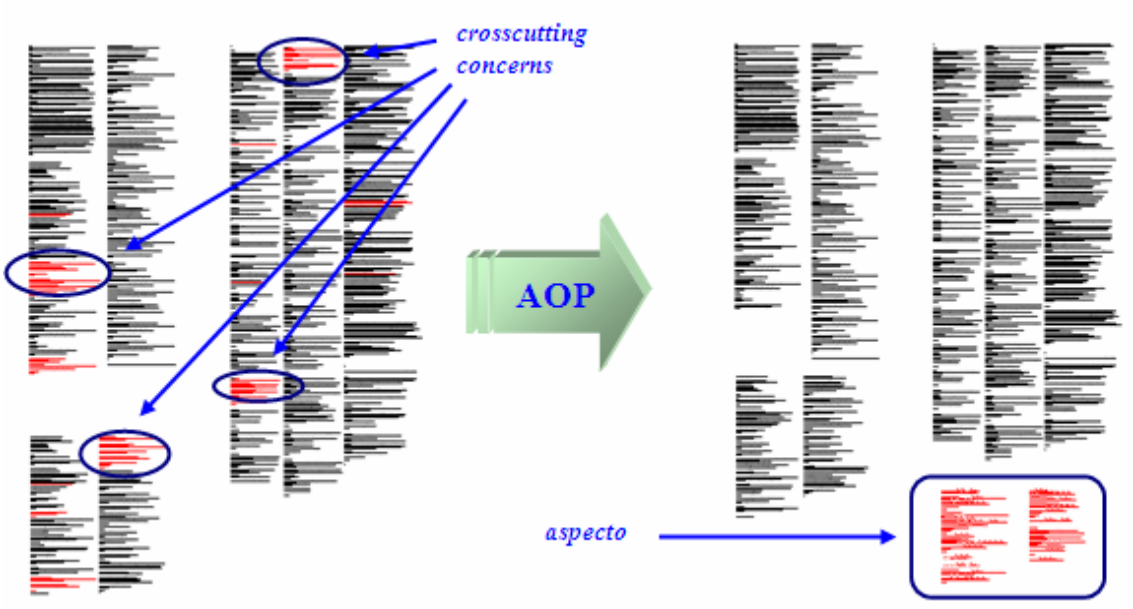

Figura 2 – Mejoras de la modularidad con AOP

AOP fue introducido en la comunidad investigadora por los trabajos de Gregor Kiczales [Kic97], [Kic01]. Su propuesta se materializó en el lenguaje de programación AspectJ [AsJ03], [AsJ07], desarrollado en Palo Alto Research Center (PARC) y financiado conjuntamente por Xerox, por el National Institute of Standards and Technology (NIST) y por el DARPA. Posteriormente fue transferido a la comunidad de software abierto e integrado en el proyecto Eclipse, con el objetivo de ampliar la comunidad de usuarios. Como resultado, en la actualidad AspectJ es el lenguaje más ampliamente utilizado para la programación orientada a aspectos (AOP).

Aunque AOP emergió a nivel de implementación, su uso se ha extendido a cada una de las etapas del ciclo de desarrollo software, dando lugar a lo que se denomina como Desarrollo de Software Orientado a Aspectos (Aspect-Oriented Software Development, AOSD [Aosd]). El AOSD está teniendo un gran auge en la comunidad informática debido a que su código modular consigue que los costes de desarrollo, mantenimiento y evolución del software se reduzcan. Por este motivo, cada vez es mayor la tendencia hacia la incorporación del concepto de aspecto en las primeras fases del ciclo de vida, como la etapa de requisitos y la de diseño.

A continuación se presentan los principales conceptos de AOP con el objetivo de facilitar la comprensión de los conceptos posteriores. Puede encontrarse una descripción más detallada en [Per06, cap.3, secc. 1].

2.2.1 Código Base

AOP introduce una diferenciación explícita entre el código base y el código aspectual. El código base se compone de las unidades de software de un sistema módulos, objetos, componentes) que han sido obtenidos como resultado de una descomposición funcional. Sin embargo, el código aspectual se compone del conjunto de aspectos que han sido desarrollados para encapsular los distintos *crosscutting concerns* del sistema.

2.2.2 Join Point (puntos de enlace)

Los join points se sitúan en el código base de una aplicación. Un join point es un concepto semántico que define un punto específico de la ejecución del código base. Este punto puede ser extendido con el código aspectual, modificando de esta forma el flujo de ejecución original de la aplicación. Los join points son los elementos de coordinación ente el código base y el código aspectual.

Un join point no es una construcción explícita del lenguaje, sino que está asociado a un concepto semántico del lenguaje: por ejemplo, la invocación de un método, la declaración de una variable, una excepción, etc. En tiempo de ejecución, una misma construcción del lenguaje, dependiendo de las diferentes instanciaciones de dicha construcción, será un join point distinto. Un ejemplo claro de join point es el asociado a la invocación de un método. Las diferentes invocaciones del mismo método serán diferentes joinpoints con semántica diferente.

2.2.3 Pointcut

De la sección anterior se desprende que una aplicación podrá tener una gran cantidad de join points en su código base. Sin embargo, no todos los join points son interesantes o relevantes para inyectar el código de los aspectos en el código base. Como resultado, los join points relevantes para la inyección del código del aspecto deben ser seleccionados de algún modo. El mecanismo que permite su selección es el *pointcut*.

Un pointcut es un conjunto de join points cadidatos para inyectar el código del aspecto en el código base en tiempo de ejecución, y sus múltiples instanciaciones. Los pointcuts realizan el proceso de enlazado (*weaving*) entre el código base y el código del aspecto mediante la captura de los distintos join points.

2.2.4 Advice

Un *advice* define el código que debería ser ejecutado en el joinpoint de un pointcut específico. Los advices definen el código adicional para los join points que han sido seleccionados por los pointcuts. El proceso de insertar o reemplazar código base por el código del aspecto se denomina *weaving*.

Existen tres tipos principales de advice:

- **Before**: el advice *before* añade código al programa base justo antes del join point. Como resultado, el código del advice es ejecutado antes del código del join point.
- **Around**: el advice *around* sustituye el código del join point. Como resultado, el advice es ejecutado en lugar del código del join point.
- **After**: el advice *after* añade código al programa base justo después del join point. Como resultado, el código del advice es ejecutado después del código del join point.

Existen más tipos de advice, pero que son variaciones de los presentados anteriormente. Por ejemplo, *after returning* es un advice que se ejecuta cuando la ejecución del join point finaliza correctamente.

2.2.5 Aspecto

Un aspecto es una primitiva del lenguaje que encapsula un *crosscutting-concern*. Un aspecto está enlazado con uno o más métodos del código base a través de pointcuts. Por esta razón, un aspecto se compone de un conjunto de pointcuts y de uno o más advices. Como resultado, los aspectos especifican si su ejecuciónserá antes, después o en lugar de un método del código base a través del advice que está asociado a los pointcuts. Aunque los aspectos son normalmente pares de pointcuts y advices, pueden tener también su propio estado [Dou05]. La definición de aspecto propuesta por Kiczales es la siguiente:

> *"Aspectos son unidades modulares implementando un* crosscutting concern*, compuestos de pointcuts, advices y declaraciones ordinarias de Java"* [Kic97]

De esta forma, todo el código relativo a una funcionalidad concreta del sistema software (como conectarse a una base de datos) se encuentra unificado en una entidad independiente del resto del código, por lo que se facilita su mantenimiento.

Entre las distintas aproximaciones e implementaciones del modelo orientado a aspectos, se pueden observar dos tendencias diferenciadas entre sí por el proceso de generación de código: los modelos estáticos y los modelos

dinámicos. Los modelos estáticos (como AspectJ [AsJ03] o Loom.NET [Sch02]) generan como resultado del *weaving* un sólo componente en el que se encuentra mezclado el código funcional y el código de los aspectos. Por su parte, en los modelos dinámicos (como JAsCo [Suv03], Rapier-Loom [Sch03] o EOS [Raj03]) se generan distintas entidades para los aspectos y el componente. La ventaja de los modelos dinámicos es la facilidad que proporcionan para incluir o eliminar aspectos durante el proceso de ejecución. Este tipo de modelos facilitan la evolución y adaptación del software, puesto que permiten extender el código existente añadiendo o eliminando aspectos [Yan02].

2.3 PRISMA

PRISMA [Per06] es un *framework* para el desarrollo de sistemas software complejos, distribuidos y altamente reutilizables que incluye un modelo [Per05b], un lenguaje de descripción de arquitecturas orientado a aspectos [Per06a], una herramienta (PRISMACASE [Per06, cap. 9]) y una metodología. En esta sección se presentará tanto el modelo como el lenguaje de este *framework*.

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.

2.3.1 El Modelo PRISMA

PRISMA se caracteriza por la integración que realiza del Desarrollo de Software Basado en Componentes (CBSD) y del Desarrollo de Software Orientado a Aspectos (AOSD). Es por ello que los modelos arquitectónicos PRISMA se construyen en base a componentes y aspectos. Esta integración se consigue definiendo los tipos arquitectónicos (componentes, conectores y sistemas) como una 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 3) 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.

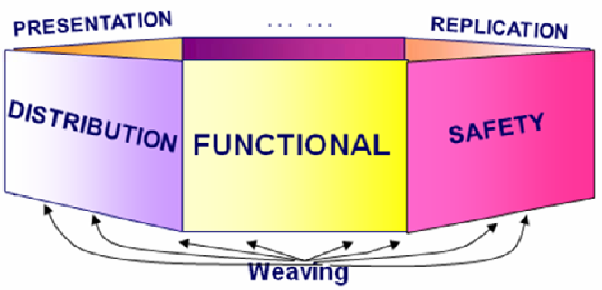

Figura 3 – Vista interna de un elemento arquitectónico PRISMA

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 4).

Figura 4 – 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.

2.3.2 Visión orientada a Aspectos

El desarrollo de software orientado a aspectos (AOSD) permite encapsular en un solo elemento (el aspecto) aquella funcionalidad que se encuentra diseminada a lo largo de todo el sistema (*crosscutting concern*). Un *crosscutting concern* en PRISMA se identifica con un aspecto que es importado por cada uno de los elementos arquitectónicos que se ven afectados por dicho

concern (ver Figura 5). Dicho de otro modo, un aspecto afecta (*crosscut*) aquellos elementos de la arquitectura que lo importan.

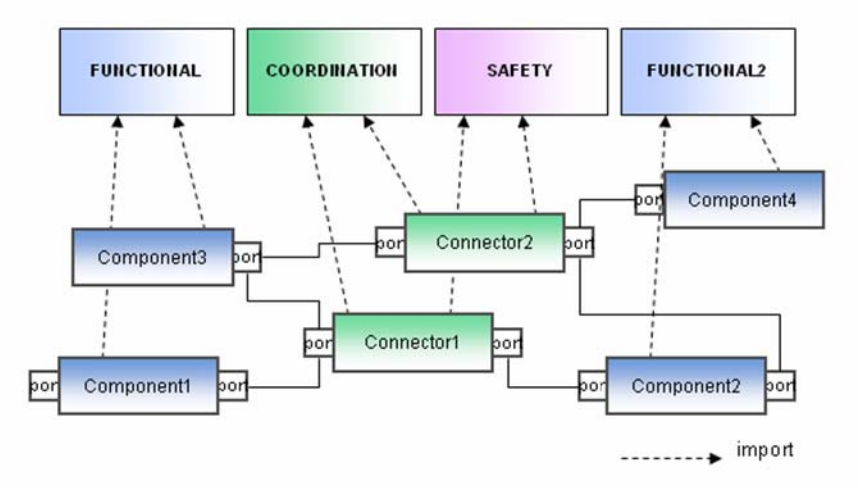

Figura 5 – *Crosscutting-concerns* en arquitecturas PRISMA

El modelo PRISMA va más allá y amplía el concepto de aspecto en el sentido de que prescinde del concepto de código base: la funcionalidad del sistema no se encuentra localizada en un núcleo central, sino que dicha funcionalidad se define también como un aspecto. Es por ello que, desde el punto de vista de la orientación a aspectos, PRISMA es un modelo simétrico [Har02].

Todo elemento arquitectónico está compuesto íntegramente por aspectos (ver Figura 3), 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], [Ali05].

Existen diferentes tipos de aspectos, definiéndose cada uno de ellos de forma independiente, y siendo el número de tipos de aspectos extensible. 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 modelados con 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. En términos de AOP (ver sección 2.1), los *join points* corresponden a los servicios definidos en los aspectos y los *pointcuts* corresponden a los *weavings*. A diferencia de otras aproximaciones orientadas a aspectos, los *weavings* (o *pointcuts*) se definen fuera de los aspectos, con lo que se consigue que los aspectos sean totalmente independientes del elemento arquitectónico que los importa.

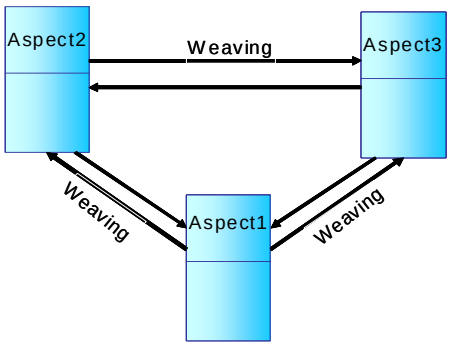

Figura 6 – Weavings entre aspectos

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. Existen varios tipos de *weavings*:

- *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.

Los aspectos PRISMA son altamente reutilizables e independientes, pues su especificación no depende de otros aspectos ni de los elementos arquitectónicos que los importan, facilitando su mantenimiento. Un cambio en una propiedad del sistema sólo requiere el cambio en el aspecto que lo importa, siendo entonces cada elemento arquitectónico que lo importa actualizado.

2.3.3 Visión orientada a Componentes

La visión externa de los elementos arquitectónicos está orientada hacia el concepto de componentes, a diferencia de la visión interna (orientada a aspectos). 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 software con precisión.

2.3.3.1 Componentes

Un componente PRISMA se define como un elemento arquitectónico que captura la funcionalidad del sistema software 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.

2.3.3.2 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.

2.3.3.3 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, componentes complejos que permiten encapsular un conjunto de elementos arquitectónicos (conectores, componentes y sistemas) conectados entre sí. De dicha encapsulación pueden surgir propiedades emergentes. De esta forma, pueden construirse arquitecturas jerárquicas, que permiten aumentar el nivel de abstracción de la arquitectura del sistema software.

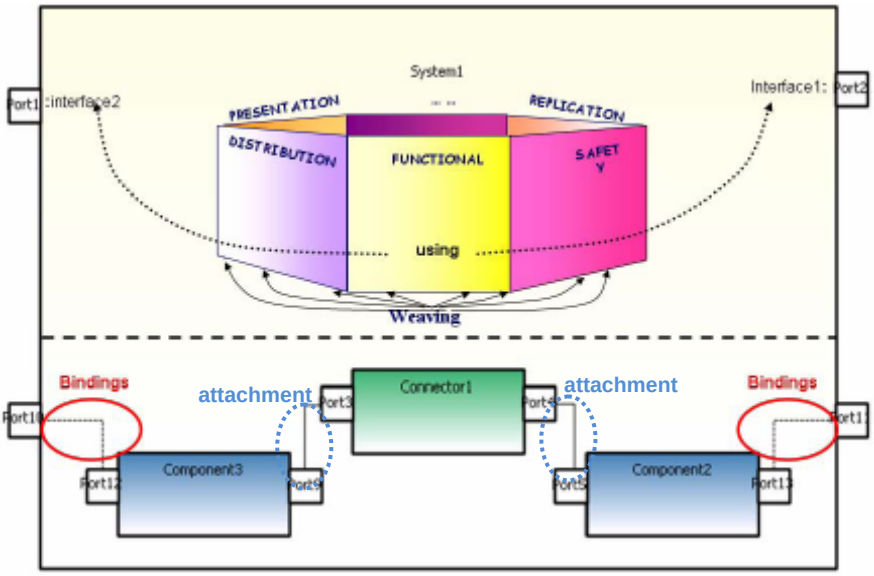

Figura 7 – Sistema PRISMA [Per06]

Un sistema está formado por los mismos elementos y restricciones que un componente: una función de identificación, un conjunto de aspectos, un conjunto de *weavings* que interconectan dichos aspectos, y un conjunto de puertos (ver Figura 7). No obstante, como es un tipo compuesto, está compuesto también por un conjunto de elementos arquitectónicos (componentes, conectores y sistemas).

Dentro de un sistema PRISMA existen dos tipos de conexiones entre los elementos arquitectónicos:

- Las conexiones entre los elementos arquitectónicos que componen el sistema se denominan **attachments** (ver attachments, Figura 7). Estas conexiones permiten que los distintos elementos arquitectónicos se comuniquen entre sí.
- Las conexiones entre el propio sistema y los elementos que contiene se denominan **bindings** (ver bindings, Figura 7). Estas conexiones permiten exportar los servicios de los elementos arquitectónicos del sistema al exterior.

Sin embargo, y como se verá en el siguiente apartado, un sistema puede estar descrito a distintos niveles de abstracción: a nivel de tipos, un sistema definirá un patrón arquitectónico altamente reutilizable; mientras que a nivel de configuración, un sistema describe un sistema concreto, para un contexto determinado.

2.3.4 Lenguaje de definición de Arquitecturas

El Lenguaje de Descripción de Arquitecturas de PRISMA (ADL) está basado en OASIS [Let98]. 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 enfoque CBSD y el enfoque AOSD. Este nivel permite definir distintos tipos arquitectónicos, que son almacenados en una librería para posteriormente reutilizarlos. Los ciudadanos de primer orden son: interfaces, aspectos, componentes y conectores. La selección y combinación de estos tipos permitirán especificar un sistema software determinado, mediante el uso de las primitivas del nivel de configuración.

El nivel de configuración permite definir las instancias y la topología que conformará un determinado sistema software. Para ello, en primer lugar se han de importar todos aquellos tipos (conectores, componentes y sistemas) definidos mediante el lenguaje de definición de tipos, 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 software, 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 reconfiguración (nivel de configuración), teniendo meta-eventos diferentes en cada lenguaje para hacer evolucionar a los tipos o a las conexiones existentes entre sus instancias.

2.4 Conclusiones

En este capítulo se han descrito las nociones preliminares de este trabajo. Han sido presentados los conceptos fundamentales del área de las arquitecturas software y del desarrollo de software orientado a aspectos.

Además, también se ha descrito la propuesta PRISMA, que permite desarrollar arquitecturas software orientadas a aspectos, basándose en el paradigma de MDD.

CAPÍTULO III

RECONFIGURACIÓN DINÁMICA DE ARQUITECTURAS SOFTWARE

3.1 Introducción

El concepto de Reconfiguración Dinámica procede del área de la configuración de sistemas distribuidos [Kra85]. Las propuestas más clásicas tan sólo contemplaban la configuración estática de sistemas distribuidos: ante una modificación de la topología del sistema, éste tenía que detenerse por completo y volverse a compilar tomando como entrada la nueva configuración. Sin embargo, poco a poco fue surgiendo el interés en soportar la configuración dinámica de los sistemas distribuidos, con el objetivo de tolerar una mayor disponibilidad.

Las propuestas dinámicas son aquellas que permiten modificar la configuración de un sistema distribuido en tiempo de ejecución, sin necesidad de detener por completo todo el sistema ni de recompilarlo. La propuesta de Kramer y Magee (CONIC) [Kra85] fue una de las primeras en soportar la configuración dinámica de sistemas distribuidos.

Posteriormente, a medida que el área de Arquitecturas Software se fue consolidando, los conceptos de configuración (estática y dinámica) de sistemas distribuidos fueron incorporándose progresivamente a los Lenguajes de Descripción de Arquitecturas (ADLs). Prueba de ello es que los mismos autores de CONIC también desarrollaron una propuesta de descripción de arquitecturas software (DARWIN, [Mag95]) que incorporaba los conceptos de configuración dinámica presentados con anterioridad.

En el área de Arquitecturas Software no existe una definición común y aceptada del concepto de Reconfiguración Dinámica, debido a que su significado es bastante intuitivo. Sin embargo, dado que en este trabajo se va a hacer un uso extensivo del término, para una mayor claridad definiremos el concepto:

> *La Reconfiguración Dinámica de Arquitecturas Software es un tipo de evolución del software que permite modificar la topología de un sistema en ejecución sin interrumpir su ejecución.*

Debido a que dicha definición se enmarca dentro del área de las arquitecturas software, la *topología* de un sistema hace referencia al conjunto de elementos arquitectónicos que lo consituyen (componentes y conectores) y a las relaciones entre ellos. Además, al referirse a la topología de un *sistema en ejecución*, se está haciendo referencia a las *instancias* de los elementos arquitectónicos que forman el sistema. No debe confundirse con el *tipo* de los elementos arquitectónicos. La Reconfiguración Dinámica únicamente modifica el número de instancias en ejecución de los tipos definidos en el sistema. La modificación de los tipos arquitectónicos en ejecución corresponde al área de la Evolución de Tipos, que es abordada en el siguiente capítulo.

Dado que la descripción más simple de cualquier arquitectura de software puede modelarse como un grafo, existen cuatro operaciones básicas para realizar reconfiguraciones [Kra85]:

- Create: permite instanciar elementos arquitectónicos
- Destroy: destruye instancias de elementos arquitectónicos
- Link: permite crear enlaces entre elementos arquitectónicos
- Unlink: destruye enlaces entre elementos arquitectónicos

Toda operación compleja se reduce a estas cuatro. En otros trabajos posteriores, como [Ore99] o [Bra04] se citan otras operaciones no básicas, como la sustitución de un componente por otro del mismo tipo, pero son básicamente combinaciones de las operaciones básicas.

Por otro lado, en nuestra propuesta se considera muy importante el hecho de que las propias arquitecturas sean capaces de reconfigurarse a sí mismas, siguiendo un enfoque autónomo. En la mayoría de las propuestas, la Reconfiguración Dinámica se lleva a cabo mediante entidades centralizadas. El hecho de permitir que cada arquitectura pueda reconfigurar su topología permite la construcción de arquitecturas descentralizadas y facilita la construcción de sistemas distribuidos.

Uno de los numerosos problemas que se plantean en el área es cómo especificar qué configuraciones son correctas o válidas. Sin embargo, la dificultad disminuye en este trabajo al partir de patrones arquitectónicos definidos a nivel de tipos. Las reconfiguraciones válidas sólo son aquellas realizadas de acuerdo al patrón arquitectónico, que especifica las conexiones válidas entre elementos arquitectónicos, su cardinalidad, y el número de instancias permitidas para cada elemento arquitectónico. Además, no puede instanciarse cualquier elemento arquitectónico, sino sólo aquellos que han

sido definidos en el patrón. De esta forma, gran parte de los problemas son resueltos de una forma natural, sin necesidad de introducir numerosas restricciones adicionales, ya que el modelo gráfico de PRISMA permite especificarlos.

En este capítulo se presentará una propuesta para soportar la reconfiguración dinámica de arquitecturas PRISMA. Para ello, en primer lugar se presentará el estado del arte en el campo de la reconfiguración dinámica. En segundo lugar se presentará un caso de estudio con el cual se ilustrarán los conceptos de la propuesta. En tercer lugar se presentarán los conceptos de la propuesta y la infraestructura necesaria para soportarla.

3.2 Estado del arte

En la última década se han realizado numerosos trabajos para abordar la reconfiguración dinámica de arquitecturas software [Bra04], [Cue02] y [Buc05]. La mayor parte se han centrado en cómo incorporar la reconfiguración dinámica al lenguaje de descripción de arquitecturas y en su descripción formal. A continuación se describen algunas de las propuestas más conocidas.

Darwin [Mag95] y LEDA [Can01] están basados en el lenguaje formal π-cálculo [Mil91], al igual que PRISMA. Ambas aproximaciones crean y destruyen instancias de componentes de una forma implícita: éstas son creadas cuando cualquiera de sus servicios es invocado, y son destruidas cuando no son usadas. Sin embargo, estas aproximaciones no proporcionan operaciones específicas para cambiar la arquitectura del software en tiempo de ejecución, como la creación/destrucción de enlaces entre elementos arquitectónicos. Dynamic Wright [All98] se basa en una variante del algebra de procesos CSP. Define un componente global denominado *Configuror* que proporciona los servicios de reconfiguración dinámica y describe cómo la arquitectura debe ser reconfigurada. Este componente es el único capaz de reconfigurar la arquitectura. Sin embargo, debido a que los componentes complejos dependen de este componente global, no es posible construir componentes complejos que requieran configurarse a sí mismos de manera autónoma.

Además, ninguna de las aproximaciones anteriores proporciona mecanismos para permitir a una arquitectura software conocer su configuración actual, y en base a esta información, poder tomar decisiones sobre qué operaciones de reconfiguración deben ser aplicadas. PiLar [Cue01] [Cue02] combina el álgebra de procesos CCS con los conceptos de Reflexión Computacional [Mae87] para introducir dinamismo en arquitecturas software. Esta aproximación soporta tanto la creación/destrucción explícita de instancias de elementos arquitectónicos como la creación/destrucción de sus enlaces en

tiempo de ejecución. La reflexión es utilizada para para obtener la configuración del sistema en tiempo de ejecución. La propuesta presentada en este capítulo es similar a la presentada en PiLar, salvo que la reflexión es utilizada para obtener información de los tipos de la arquitectura (y con ello poder decidir si los reconfiguraciones son permitidas o no), y que la información de la configuración de la arquitectura se mantiene como una parte del estado del sistema, en el mismo nivel de abstracción.

Por otro lado, también han emergido una serie de propuestas para proporcionar a las arquitecturas software de capacidades de auto-adaptación [Ore99]. La auto-adaptación es similar a la reconfiguración programada y, además, proporciona mecanismos de análisis y decisión para determinar en tiempo de ejecución cuándo se necesita una reconfiguración o adaptación de la arquitectura. Dashofy et al. [Das02] describe una infraestructura para la construcción de arquitecturas software capaces de auto-repararse. Su enfoque consiste en la generación dinámica de un plan de reparación para el sistema basado en las diferencias existentes entre el estado actual del sistema y la configuración que debería tener. Dicho plan de reparación es ejecutado por un Architecture Evolution Manager (AEM) global. El AEM invoca los servicios necesarios de bajo nivel que son proporcionados por la infraestructura de ejecución para adaptar el sistema a la configuración deseada. Es la infraestructura de ejecución la que finalmente ejecuta los servicios de adaptación en toda la arquitectura. Rainbow [Che05] proporciona un enfoque para soportar la auto-adaptación de los sistemas en ejecución. Dicho enfoque se divide en varios niveles, siendo el nivel de Arquitectura el responsable del proceso de reconfiguración, desde que un requerimiento de cambio es detectado hasta que el cambio es ejecutado. Tanto Rainbow como la herramienta de Dashofy utilizan mecanismos de reconfiguración externos al sistema y centralizados. Sin embargo, los grandes sistemas software requieren que la adaptación se gestione de una forma descentralizada, es decir, cada subsistema debe tener sus propios mecanismos de reconfiguración. En este sentido, el trabajo de Georgiadis et al. [Geo02] describe una infraestructura descentralizada para soportar la auto-organización de la arquitectura. Sin embargo, la infraestructura presentada no es escalable, dado que cada instancia de componente almacena una copia de la arquitectura global.

Aunque existen numerosas propuestas que combinan el uso de aspectos con las arquitecturas software, como JAsCo [Suv03] o CAM/DAOP [Pin05], prácticamente ninguna ha abordado el problema de la reconfiguración dinámica a través de aspectos, objetivo de este capítulo. La mayor parte de los esfuerzos se han centrado en soportar el weaving dinámico de aspectos en las arquitecturas software. Con esto, dichas propuestas logran extender dinámicamente la funcionalidad de las arquitecturas software, pero no tratan la modificación dinámica de la topología de la arquitectura software.

Una propuesta similar a la de este trabajo es la de Rasche y Polze [Ras03]. En su artículo presentan cómo puede realizarse la reconfiguración dinámica de sistemas basados en componentes mediante el uso de técnicas de la orientación a aspectos. Para ello, todo el código relativo a la configuración de los componentes se encapsula en un aspecto. Sin embargo, su aproximación está centrada en el nivel de implementación y es muy dependiente de la plataforma.

3.3 Caso de estudio

Con el objetivo de ilustrar cómo se soporta la reconfiguración dinámica en este trabajo, en este apartado introducimos un caso de estudio que requiere capacidades para reconfigurar su topología en tiempo de ejecución.

Este caso de estudio modela un sistema de banca virtual, esto es, una entidad bancaria que ofrece sus servicios a través de Internet. Este sistema interactúa con dos tipos de actores: los clientes y otras entidades bancarias. Por un lado, los clientes realizan ingresos, consultas de movimientos u ordenan disposiciones de efectivo. Por otro lado, el sistema bancario necesita comunicarse con otras entidades bancarias con el objetivo de realizar o recibir transferencias bancarias (ver Figura 8). Sin embargo, para una mayor simplicidad, tan sólo se tendrán en cuenta las principales operaciones relacionadas con los clientes.

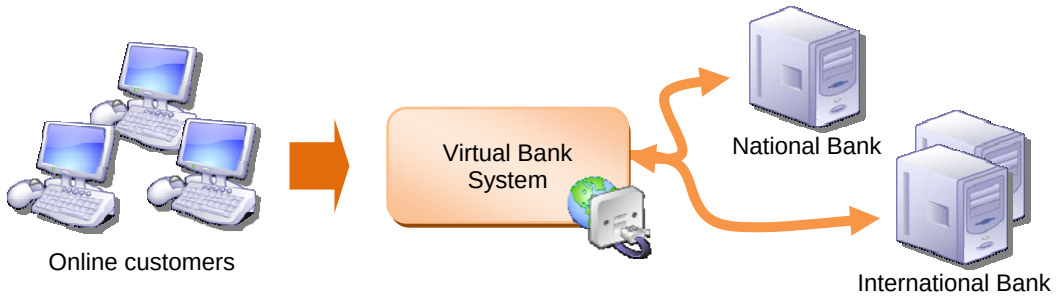

Figura 8 – Sistema de banca virtual

Como el contexto en el que se ejecuta el sistema de banca virtual es altamente inseguro, el principal requisito que debe tenerse en cuenta es la seguridad y la privacidad de las operaciones. Por esta razón, todas las operaciones con el banco virtual se realizan a través de canales cifrados: para cada cliente que se conecte al banco virtual, se crea un canal cifrado distinto (sesión). Para ello, el banco virtual ofrece el servicio *Login*, que tras comprobar si las credenciales del cliente son válidas, devuelve una clave única de sesión, *SessionID*, indicando que se ha creado una sesión segura. Es a partir de este momento cuando el cliente puede, utilizando esta clave,

realizar las operaciones bancarias deseadas de una forma segura: *withdraw*, *makeDeposit*, *makeCreditTransfer*, *getBankStatements*, etc.

El sistema bancario se ha modelado como un sistema PRISMA llamado *VirtualBankSystem* (ver Figura 9). Se compone de los aspectos *SecAsp*, *VirtualBankEvolver* y *Configuration*; de los componentes *Account* y *CustSession*; y de los conectores *ExtBank* y *AccountCnct*.

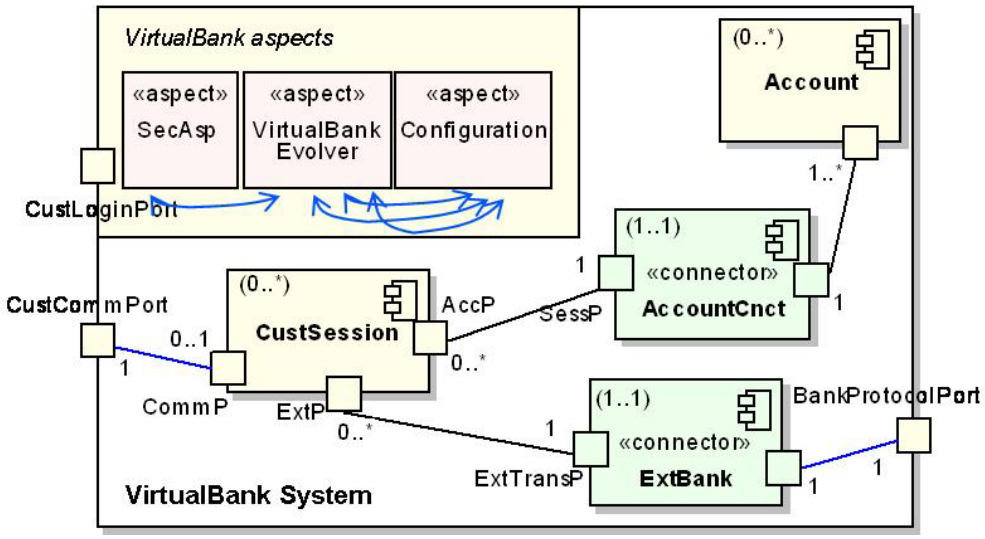

Figura 9 – Caso de estudio VirtualBank System

El componente *Account* define el estado y el comportamiento de una cuenta bancaria: contiene el saldo de una cuenta, el histórico de movimientos, los datos del titular, etc.

El componente *CustSession* proporciona los diferentes servicios bancarios que el cliente puede solicitar mientras la sesión está activa: *withdraw, makeDeposit*, etc. Dichos servicios se proporcionan al cliente a través del puerto *CustCommPort* del sistema: esto es realizado mediante un binding entre el puerto *CommP* del componente *CustSession* y el puerto *CustCommPort* del sistema *VirtualBank*. *CustSession* asegura que las operaciones bancarias son realizadas de una forma fiable, segura y confidencial. Una instancia de este componente es creada **dinámicamente** por cada sesión bancaria iniciada por un cliente. De esta forma, al mantener una instancia por sesión (en lugar de que una misma instancia gestione varias sesiones), se evita que el fallo en una sesión afecte al resto de sesiones de los otros clientes.

El conector *AccountCnct* es responsable de que las conexiones entre las instancias del componente *CustSession* y las instancias del componente *Account* sean fiables y seguras. El conector *ExtBank* gestiona los protocolos necesarios para realizar transacciones bancarias con otros sistemas bancarios, como adaptar los mensajes a los protocolos específicos de cada sistema bancario, procesos de encriptación, etc. Tan sólo es necesaria una

instancia de cada conector para mantener las distintas conexiones, debido a que su complejidad es menor y por tanto hay menor riesgo de fallo en tiempo de ejecución. Como las instancias del componente *CustSession* son creadas dinámicamente, las conexiones (attachments) entre dichas instancias y los conectores *AccountCnct* y *ExtBank* también deben ser creadas **dinámicamente**: el sistema debe configurarse dinámicamente cada vez que una nueva sesión bancaria es iniciada.

Por tanto, iniciar una sesión bancaria conlleva la siguiente reconfiguración interna del sistema:

1. Crear una nueva instancia del componente *CustSession*
2. Crear un attachment entre la nueva instancia de *CustSession* y el conector *AccountCnct*
3. Crear un attachment entre la nueva instancia de *CustSession* y el conector *ExtBank*
4. Crear un binding para exportar los servicios ofrecidos por la nueva instancia de *CustSession*.

Las mismas operaciones, pero a la inversa, deben realizarse para finalizar una sesión bancaria: deben eliminarse todas las instancias de componentes, attachments y bindings que ya no son necesarios. Por esta razón, el sistema *VirtualBank* está *continuamente reconfigurando su topología en tiempo de ejecución*, agregando o eliminando instancias de componentes, conectores y attachments. Es importante tener en cuenta que el resto de entidades que interactúen con el sistema *VirtualBank* (clientes y otros bancos) no deben verse afectados por el proceso de reconfiguración que se lleva a cabo internamente.

Por último, faltan por introducir los aspectos del sistema *VirtualBank*. Cada uno de los aspectos es de un *concern* diferente, dependiendo de cuál sea su funcionalidad en el sistema bancario. *SecAsp* es un aspecto del *concern* Security: realiza el proceso de autenticación para cada entidad que opera con el sistema bancario. Para ello, proporciona el servicio *Login* a través del puerto *CustLoginPort* del sistema *VirtualBank*. Este servicio comprueba si las credenciales facilitadas (usuario y contraseña) son válidas, y en caso afirmativo solicita la creación de una nueva sesión bancaria. Esto desencadena el proceso de reconfiguración del sistema que se ha descrito anteriormente. Cuando el proceso de creación de sesión finaliza, se le devuelve al servicio *Login* el identificador de la instancia del componente *CustSession* creado, *SessionID*, que a su vez devuelve como resultado del proceso de autentificación al cliente. Con este identificador, el cliente puede establecer una conexión segura a través del puerto del sistema *CustCommPort*.

Hasta ahora tan sólo se han descrito las entidades que componen el sistema *VirtualBank*, y cómo el sistema debe reconfigurarse, pero no se ha descrito

dónde se describe dicho proceso de reconfiguración y *quién* lo lleva a cabo. El proceso de reconfiguración se define en el aspecto *VirtualBankEvolver*, (del *concern* EvolutionManager). Es este aspecto el que gestiona cómo deben crearse las instancias y cómo deben conectarse al resto de elementos. Por su parte, el aspecto *Configuration* (del *concern* Configuration) proporciona los servicios de reconfiguración dinámica de bajo nivel y es el que realmente ejecuta la reconfiguración. No deben confundirse ambos aspectos: el primero únicamente *define* cómo debe reconfigurarse el sistema, mientras que el segundo *ejecuta* las distintas operaciones de reconfiguración y es el responsable de que se ejecuten de forma adecuada. Estos dos aspectos son los que proporciona nuestra propuesta para ofrecer capacidades de reconfiguración dinámica, los cuales serán descritos con mayor detalle en los siguientes apartados.

3.4 Servicios de Reconfiguración

El enfoque tradicional, y que siguen la mayor parte de los trabajos de la literatura, consiste en proporcionar reconfiguración a través de una entidad global, centralizada –a la que llamaremos *Configurador*–, que se encarga de reconfigurar la topología de toda la arquitectura del sistema. En arquitecturas software sencillas, de poca envergadura, este enfoque es el óptimo. Desde una misma entidad (el *Configurador*) se controla cómo debe reconfigurarse todo el sistema.

Sin embargo, en arquitecturas software complejas, de gran envergadura, en las que la arquitectura del sistema está compuesta por numerosos subsistemas (arquitecturas jerárquicas), este enfoque no es apropiado. Los sistemas software así construidos no son autónomos: los subsistemas que lo constituyen son altamente dependientes del *Configurador* de la arquitectura. Los principales inconvenientes son:

- **Pérdida de autonomía**. Cada uno de los subsistemas es el único responsable de la reconfiguración de su topología. Cada subsistema debe poder reconfigurarse de forma independiente, sin necesidad de que una entidad global, ajena al subsistema, intervenga en el proceso.
- **Dificultad de mantenimiento**. El mantenimiento de cada subsistema es más complejo, dado que las reglas de reconfiguración que describen cómo debe reconfigurarse se encuentran junto al conjunto de reglas de reconfiguración del resto de subsistemas (el *Configurador*). Esto provoca que un error afecte al proceso de reconfiguración del resto de subsistemas.
- **Falta de encapsulación**. Para reconfigurar una arquitectura, debe conocerse cuál es su topología. En un enfoque centralizado, el *Configurador* conoce la topología de todos los subsistemas que lo

forman. Sin embargo, esto no es aceptable, pues la topología de cada subsistema tan sólo debería conocerla el propio subsistema. En caso de un fallo de seguridad en el *Configurador*, la arquitectura de todo el sistema software sería accesible al atacante, y por tanto, el sistema se vería comprometido.

- **Sistemas muy centralizados.** El enfoque de un *configurador* global tan sólo favorece el desarrollo de sistemas muy centralizados. Sin embargo, es conocido que las arquitecturas complejas son altamente distribuidas, y para mayor escalabilidad, el *configurador* debería ubicarse en cada nodo. Pero esto ocasiona problemas de mantenimiento, pues ante cualquier cambio, debe actualizarse en cada uno de los nodos, como ocurre con la propuesta de [Geo02].

El otro enfoque es descentralizar el proceso de reconfiguración. Este enfoque es preferible para la construcción de arquitecturas software complejas, en la que se delega el proceso de reconfiguración a cada subsistema. El principal inconveniente es que, dependiendo de cómo se realice, puede dificultar el mantenimiento de las reglas de reconfiguración, al estar éstas dispersas a lo largo de toda la arquitectura del sistema.

En nuestra propuesta, la reconfiguración es soportada de forma descentralizada, autónoma. Cada sistema PRISMA tiene acceso a un conjunto de servicios de reconfiguración dinámica independientes del resto de la arquitectura. Por una parte, la arquitectura que contiene a un sistema PRISMA no conoce la topología de dicho sistema ni puede reconfigurarlo. Por otra parte, reconfiguraciones de un sistema PRISMA no van a afectar al resto de sistemas en ejecución en la arquitectura.

Por esta razón, las instancias del sistema pueden necesitar evolucionar su configuración inicial en tiempo de ejecución por sí mismas, es decir, soportar reconfiguración dinámica autónoma. Por ejemplo, el sistema *VirtualBank* (ver sección 0) necesita reconfigurarse a sí mismo en tiempo de ejecución para gestionar las solicitudes de los clientes mediante la creación de nuevas instancias de componentes (instancias del componente *CustSession*) y mediante su conexión con el resto de elementos arquitectónicos (e.g.: *AccountCnct* y *ExtBank*). Sin embargo, reconfiguraciones de instancias del sistema autónomas están sujetas a las restricciones del patrón arquitectónico del sistema para satisfacer las propiedades del sistema y para evitar la degradación del sistema. Por ejemplo, una instancia del componente *CustSession* sólo puede conectarse con un conector *AccountCnct* (ver cardinalidades en Figura 9). Por esta razón, es importante introducir una forma de permitir a cada instancia de sistema reconfigurarse a sí misma de forma dinámica, pero teniendo en cuenta las restricciones definidas en el tipo del sistema (patrón arquitectónico).

La configuración del sistema debe considerarse como una parte del estado del sistema, porque diferentes configuraciones de sus elementos pueden cambiar el comportamiento del sistema. La configuración consiste en referencias a los elementos de los cuales el sistema está compuesto y a las conexiones entre los elementos. De esta forma, un sistema debe proporcionar servicios para tener conocimiento de su configuración interna y para cambiar su configuración en tiempo de ejecución. Los servicios que cambian la configuración del sistema son los *servicios de reconfiguración.* Estos servicios modifican la configuración del sistema de una forma segura asegurando que: (i) los elementos arquitectónicos en ejecución y las conexiones no sufren los cambios, y (ii) los elementos que padecen los cambios pueden finalizar sus transacciones de una forma consistente.

Antes de cambiar la configuración del sistema, los servicios de reconfiguración tienen que asegurar que los cambios requeridos no violan las restricciones descritas en el patrón arquitectónico del sistema. Esta verificación es realizada a través de mecanismos de refelexión. La Reflexión Computacional [Mae87] es la capacidad de un sistema software de razonar sobre sí mismo y de actuar sobre sí mismo. En nuestra propuesta, cada servicio de reconfiguración es ejecutado a nivel de instancias. Cada servicio de reconfiguración utiliza capacidades de reflexión para obtener la información del tipo del sistema en tiempo de ejecución; por ejemplo, solicitar qué tipo de conexiones entre los elementos arquitectónicos son correctos.

Los servicios de reconfiguración deben estar disponibles para cada sistema que necesite reconfigurarse a sí mismo. Como el Lenguaje de Descripción de Arquitecturas (ADL) de PRISMA no proporciona estas capacidades, un nuevo aspecto ha sido desarrollado para proporcionar acceso a la configuración del sistema y sus servicios de reconfiguración. El encapsular los servicios de reconfiguración en un aspecto tiene dos ventajas. Por una parte, el ADL de PRISMA no necesita ser extendido con nuevas primitivas. Por otra parte se evita que el concern de reconfiguración esté entrecruzado con el resto de concerns del sistema.

3.5 Arquitectura de la propuesta

3.5.1 Aspecto *Configuration*

La Reconfiguración Dinámica es un concern que se encuentra disperso (*crosscut*) entre la arquitectura de aquellos sistemas con necesidades dinámicas. Para evitar que el concern relativo a la reconfiguración dinámica se entrecruce con el resto de la funcionalidad de la arquitectura, dicho concern se ha encapsulado en un aspecto llamado *Configuration*. El aspecto

Configuration encapsula cada propiedad y comportamiento relativo a la reconfiguración dinámica. Éste es un aspecto genérico que se ha añadido al modelo PRISMA para dotar a los elementos arquitectónicos de capacidades dinámicas (ver Figura 10). Cualquier arquitectura o componente complejo que necesite capacidades de reconfiguración para evolucionar su composición interna importará este aspecto *Configuration*.

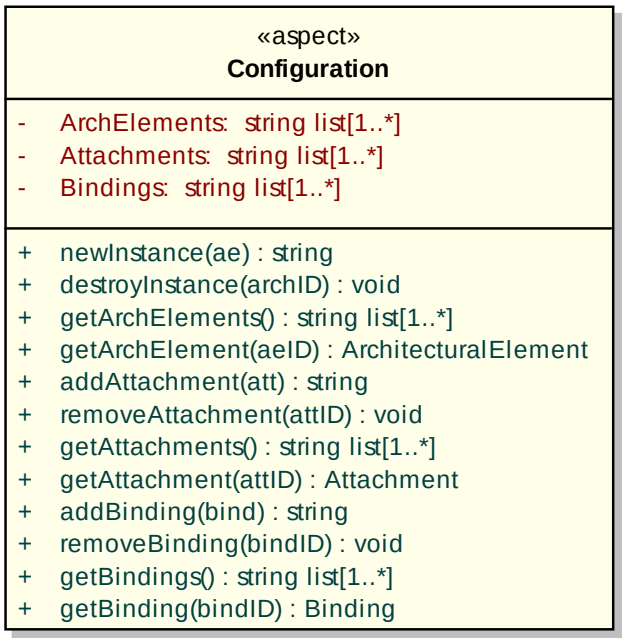

Figura 10 – El aspecto *Configuration*

El aspecto *Configuration* define un conjunto de atributos que almacenan la configuración actual del sistema y proporciona un conjunto de servicios para mantener y evolucionar esta configuración. Los atributos que el aspecto proporciona son listas dinámicas que almacenan referencias a: (i) instancias de los elementos arquitectónicos en ejecución del sistema (instancias de componentes, conectores y sistemas), (ii) instancias de attachments del sistema en ejecución, y (iii) instancias de bindings del sistema en ejecución.

Los servicios que el aspecto *Configuration* proporciona (ver Figura 10) permiten a los sistemas consultar y modificar la información contenida en los atributos descritos anteriormente. Existen dos tipos de servicios: los servicios de *introspección* y los servicios de *reconfiguración*. Los servicios de *introspección* permiten a una instancia de sistema conocer cuál es su topología en tiempo de ejecución. A su vez, los servicios de introspección pueden dividirse en dos tipos: aquellos que devuelven el conjunto de identificadores de los elementos que están en ejecución, y aquellos que devuelven información relativa al estado de un elemento concreto. Un ejemplo del primer tipo de servicios sería el servicio `getAttachments(...)`, que devuelve los IDs (i.e.: referencias) de todos los attachments que están instanciados en el sistema. Un ejemplo del segundo tipo de servicios sería el servicio `getAttachment(...)`, que dado el identificador de un attachment

concreto, devuelve la información relativa a los elementos arquitectónicos que está conectando y a través de qué puertos.

Los servicios de reconfiguración permiten modificar las referencias que están almacenadas en los atributos, es decir, modificar la configuración. Por ejemplo, el servicio `addAttachment` permite establecer una conexión entre dos instancias de elementos arquitectónicos en ejecución, indicando su ID y los puertos a conectar.

Además, los servicios de reconfiguración también tienen en cuenta si las reconfiguraciones solicitadas son permitidas por el tipo del sistema. Por ejemplo, el servicio `addAttachment` sólo permitirá crear el attachment requerido si los elementos arquitectónicos a conectar pueden efectivamente ser conectados y si la restricción de cardinalidad máxima sobre el número de dicho tipo de attachments se satisface. Esta verificación de las restricciones es realizada mediante mecanismos de reflexión. Cuando el aspecto *Configuration* es instanciado por primera vez en el sistema que lo importa, obtiene por reflexión la información del tipo del sistema, es decir, el patrón arquitectónico válido. Dicho patrón arquitectónico define qué tipos de conexiones son válidas, la cardinalidad máxima y mínima de dichas conexiones, los tipos de elementos arquitectónicos que pueden ser instanciados y el número de dichas instancias (ver el patrón arquitectónico presentado en el caso de estudio, Figura 9).

Los servicios de evolución ofrecidos por el aspecto son ejecutados directamente por la plataforma de ejecución. En nuestro caso, el middleware PrismaNET [Cos05], [Per05a]

3.5.2 Aspecto *EvolutionManager*

El aspecto *Configuration* proporciona los servicios de reconfiguración y la configuración actual del sistema. Sin embargo, este aspecto no se encarga de disparar el proceso de reconfiguración. Dicho proceso puede ser solicitado por cualquiera de los otros aspectos que constituyen el sistema, como resultado de una solicitud externa o como resultado del funcionamiento normal del sistema.

Un sistema de naturaleza dinámica necesita definir un conjunto de transacciones de configuración para reconfigurar su composición como respuesta a solicitudes externas. Una transacción de configuración es un conjunto ordenado de solicitudes de servicios de reconfiguración que debe ser realizado para obtener una nueva configuración adecuada, es decir, una nueva configuración que satisfaga el patrón arquitectónico del sistema.

Con el objetivo de prevenir que las transacciones de configuración se entremezclen con el resto de la funcionalidad del sistema, un nuevo aspecto

denominado *EvolutionManager* ha sido definido en PRISMA. Este aspecto describe el proceso de reconfiguración que debe ser realizado y el orden en que las solicitudes de servicio deben dispararse cuando se requiere un cambio; por ejemplo, cómo las nuevas instancias de componentes deben ser instanciadas y configuradas en la arquitectura del sistema. El aspecto *EvolutionManager* define el proceso de reconfiguración de cada elemento de un sistema.

Dicho aspecto consta de un conjunto de servicios de reconfiguración y de un conjunto de transacciones de configuración. Los servicios de reconfiguración definidos en el aspecto *EvolutionManager* son servicios de alto nivel, dependientes del dominio. Definen las posibles operaciones de reconfiguración permitidas de acuerdo con el patrón arquitectónico. Por ejemplo, si no es posible crear instancias de un elemento arquitectónico, el servicio de instanciación de elementos de dicho tipo no aparecerá en el aspecto. Lo mismo ocurre con las conexiones. Sólo las conexiones permitidas tendrán servicios para crear dichas conexiones. Por su parte, una transacción de configuración se define como un conjunto de invocaciones de los servicios de reconfiguración definidos en un orden establecido.

De esta forma, el proceso de reconfiguración es definido por el arquitecto software mediante la definición de transacciones de reconfiguración y la orquestación de los distintos servicios de reconfiguración para configurar la topología del sistema. De esta forma, se separan los servicios de reconfiguración de bajo nivel ofrecidos por el aspecto *Configuration* de los servicios dependientes del modelo ofrecidos por el aspecto *EvolutionManager* y se limitan las operaciones de reconfiguración a las definidas por el patrón arquitectónico.

Para ilustrar esta propuesta se utilizará el caso de estudio descrito anteriormente. En dicho caso de estudio, una nueva instancia del componente *CustSession* debe crearse dinámicamente cuando un cliente invoque el servicio *login* a través del puerto *CustomLoginPort* de *VirtualBank*, que publica los servicios del aspecto de seguridad *SecAsp* (ver Figura 9). Esta nueva instancia debe ser conectada adecuadamente con el resto de elementos arquitectónicos en tiempo de ejecución.

Todas las transacciones de configuración que el sistema necesita han sido definidas en el aspecto *VirtualBankEvolver*, del concern *EvolutionManager*. Por ejemplo, la transacción `createCustomerSession` de este aspecto (ver Figura 11) define cómo las nuevas sesiones del cliente deben ser creadas y configuradas.

La creación de una nueva sesión de cliente se especifica en el ADL de PRISMA como un proceso transaccional, para asegurar que todo el proceso de configuración se realiza de forma atómica. Como resultado, si la ejecución de uno de los servicios de configuración falla, los servicios que habían sido

ya ejecutados son deshechos, volviendo al estado previo. Las transacciones en el ADL de PRISMA se definen como procesos cuya sintaxis está basada en el álgebra de pi calculus. Por esta razón, el proceso de configuración ha sido dividido en dos subprocesos. En el primer subproceso (ver nº1, Figura 11) se invoca la creación de una nueva instancia del componente *CustSession*, proporcionando como dato de inicialización el ID del usuario asociado a la cuenta (UserID). Cuando la instancia es creada, el servicio devuelve el identificador de dicha instancia, que identifica a la sesión creada (SessionID).

```
EvolutionManager aspect VirtualBankEvolver
 ...
 Services
  createCustSession(input UserID:string,output SessionID:string);
  createAccount(user:string,balance:decimal,address:string,
     output compID:string);
  getExtBank!(output ExtBankID:string);
  getAccountCnct!(output AccountCnctID:string);
  attachCustSession2ExtBank(CompID:string, ExtBankID:string,
     output attID1:string);
  attachCustSession2AccountCnct!(CompID:string, AccountCnctID:string,
     output attID2:string);
  bindCustSessionCommP!(CompID:string,output BindID:string);
 ...
 Transactions
// Transaction of CustomerSession instance creation
   createCustomerSession(input UserID; output SessionID):
(1)  CREATE_CUSTOMER_SESSION =
       createCustSession!(UserID,output SessionID)→
       Create_Session_Connections;

(2)   CREATE_SESSION_CONNECTIONS =
// Connection between the new CustSession instance and the ExtBank
// and AccountCnct instances
(2.1)   getExtBank!(output ExtBankID) →
(2.2)   attachCustSession2ExtBank(SessionID,ExtBankID, output attID1)→
       getAccountCnct!(output AccountCnctID) →
       attachCustSession2AccountCnct!(SessionID,AccountCnctID,output attID2)→
// Binding of CustSession instance to the CommP port
(2.3)   bindCustSessionCommP!(SessionID,output BindID);

// Transaction of CustomerSession instance destr.
 destroyCustomerSession(input SessionID):
    ...
End_Aspect VirtualBankEvolver;
```

Figura 11 – Fragmento de la especificación del aspecto EvolutionManager

El segundo subproceso (ver nº2, Figura 11) realiza la conexión de la nueva instancia al resto de elementos arquitectónicos: de acuerdo con el patrón arquitectónico, los conectores *ExtBank* y *AccountCnct*. Debido a que las referencias a dichos conectores no son conocidas en tiempo de diseño, deben ser obtenidas en tiempo de ejecución. Esto es realizado mediante la invocación de los servicios `getExtBank()` y `getAccountCnct()` (ver nº2.1, Figura 11). Dichos servicios han sido definidos por el usuario para consultar la configuración en ejecución y obtener las instancias que se requieren.

Como en el patrón arquitectónico se indica que sólo puede haber en ejecución una instancia de dichos conectores, la especificación de dichos servicios se limita a obtener la lista de instancias del tipo *ExtBank* y devolver la única instancia que forma parte de dicha lista. Sin embargo, pueden realizarse operaciones más complejas, como recorrer cada una de las instancias hasta encontrar la que satisface la propiedad deseada.

Una vez obtenidas las referencias de las instancias de los conectores con los que se ha de conectar la nueva instancia de *CustSession*, se procede a conectarlos entre sí, mediante la creación de un attachment. Esto se lleva a cabo invocando los servicios `attach2ExtBank()` y `attach2AccountCnct()` (ver nº2.2, Figura 11). Como resultado, se obtiene la referencia del attachment creado, por si se necesita para tratamientos posteriores.

Por último, debe crearse el binding que permita exportar los servicios ofrecidos por la nueva instancia de *CustSession* fuera del sistema. Esto se realiza mediante la invocación del servicio `bindCustSessionCommP()` (ver nº2.3, Figura 11). Dicho servicio establece un binding entre el puerto del sistema *CommP* y el puerto *CustCommPort* de la nueva instancia.

3.5.3 Activación del proceso de reconfiguración

Además de especificar la transacción de configuración (i.e. *createCustomerSession*) también es necesario definir *cuándo* deben activarse los pocesos de configuración, es decir, cuándo deben ejecutarse las transacciones de configuración definidas.

El disparo del proceso de reconfiguración se realiza en PRISMA a través de la sincronización de cualquier servicio del resto de aspectos del sistema con la transacción de reconfiguración deseada del aspecto *EvolutionManager*. Dicha sincronización entre aspectos se realiza mediante los weavings.

Por ejemplo, la transacción *createCustomerSession* sólo debe ejecutarse si el servicio *login* valida que las credenciales proporcionadas (*UserID* y *Pass*) son correctas. Este servicio devuelve una clave de sesión si las credenciales son correctas, o *null* en caso negativo. (ver nº1, Figura 12)

Un aspecto no puede solicitar un servicio de otro aspecto, debido a que esto entremezclaría el código de los diferentes concerns del sistema. Por esta razón, las operaciones de reconfiguración solicitadas por el aspecto *VirtualBankEvolver* son *weveadas* con el aspecto *Configuration*, que proporciona la ejecución de reconfiguración correspondiente. Por ejemplo, cuando la transacción *CreateCustomerSession* solicita ejecutar el servicio *createCustSession*, un weaving es ejecutado que reemplaza la ejecución de este servicio con la ejecución del servicio *newInstance* del aspecto

Configuration (ver nº2, Figura 12). De forma similar se realiza con el proceso de conexión de instancias (ver nº3, Figura 12).

```
System VirtualBank
   Security aspect import SecAsp;
   EvolutionManager aspect import VirtualBankEvolver;
   Configuration aspect import Configuration;
   ...
Weavings
  ⌈VirtualBankEvolver.CREATECUSTOMERSESSION(SessionID,UserID)
1 |   afterif(ok=true)
  ⌊SecAspect.login(UserID,Pass,Ok,SessionID);

   VirtualBankEvolver.DESTROYCUSTOMERSESSION(SessionID)
      after
   SecAspect.logout(UserID,SessionID);

  ⌈Configuration.newInstance("CustSession",[UserID],CompID)
2 |   instead
  ⌊VirtualBankEvolver.createCustSession(UserID,CompID);

  ⌈Configuration.addAttachment("AccP",SessionID,"SessP",AccountCnctID,AttID)
3 |   instead
  ⌊VirtualBankEvolver.attachCustSession2AccountCnct!(SessionID,AccountCnctID,
       AttID);
   ...
End_Weavings;
...
End_System VirtualBank
```

Figura 12 – Especificación de los weavings en el sistema *VirtualBank*

Como resultado de este proceso de configuración, una nueva instancia de componente es creada y configurada adecuadamente en la arquitectura existente en tiempo de ejecución. La principal ventaja obtenida es que se evita tener código (especificaciones) entremezcladas: (i) las transacciones de configuración se describen externamente al proveedor de los servicios de reconfiguración, y (ii) las condiciones para disparar dichas transacciones de configuración también se definen independientemente.

3.6 Infraestructura necesaria

En los apartados anteriores se ha descrito cómo especificar y usar la Reconfiguración Dinámica a un alto nivel de abstracción: a través de dos aspectos (*EvolutionManager* y *Configuration*) los elementos arquitectónicos pueden reconfigurar su topología, sin necesidad de preocuparse de los detalles de bajo nivel dependientes de la tecnología.

Sin embargo, los detalles de bajo nivel no son menos importantes, dado que la ejecución de arquitecturas software y el soporte a su reconfiguración en

tiempo de ejecución debe poder ser implementable. Por esta razón, en esta sección se describirán las características que debe proporcionar la tecnología para soportar la Reconfiguración Dinámica, así como pautas de diseño para ofrecer dichas características dadas las limitaciones tecnológicas actuales. No se introducirán detalles de implementación, por estar fuera de los objetivos de este trabajo, aunque dichas pautas serán utilizadas para futuras implementaciones.

Para comprender los mecanismos que hacen posible la Reconfiguración Dinámica, primero es necesario comprender cómo se ejecutan las arquitecturas software. Tal como se ha descrito anteriormente, una arquitectura software es un conjunto de elementos arquitectónicos simples (componentes y conectores) o complejos (sistemas) que proporcionan y/o solicitan servicios a través de sus puertos. Las interacciones entre los distintos elementos arquitectónicos se establecen a través de los puertos. En PRISMA, las conexiones entre puertos de elementos arquitectónicos se denominan attachments.

El proceso de Reconfiguración Dinámica de una arquitectura puede conllevar la creación o destrucción de instancias de elementos arquitectónicos, así como la modificación de las conexiones entre las instancias existentes. Este proceso no es sencillo: cambiar las referencias en memoria de los elementos arquitectónicos y sus interacciones (i.e. las configuraciones del sistema) en tiempo de ejecución sin afectar al resto de la arquitectura no es fácil, debido al gran número de dependencias. Por ejemplo, la destrucción de una instancia de componente implica verificar lo siguiente: (i) que las comunicaciones entre otros elementos arquitectónicos son completadas de forma segura (i.e. no hay interrupción de las transacciones en ejecución) y (ii) que los puertos no están disponbiles para futuras comunicaciones. Si además dicha instancia estaba conectada a otros elementos arquitectónicos, además debe verificarse que: (i) las interacciones son detenidas de forma segura (no se interrumpe un servicio ya en ejecución); (ii) los elementos arquitectónicos son notificados adecuadamente, para evitar esperas indefinidas y que puedan conectarse a otros elementos arquitectónicos alternativos.

3.6.1 Operaciones de Reconfiguración

El proceso de reconfiguración de una arquitectura consiste en la ejecución transaccional de varias operaciones de reconfiguración (por ejemplo, ver el proceso de reconfiguración *CreateAccount* del caso de estudio, cuya especificación se muestra en la Figura 11). La complejidad de este proceso dependerá del número y tipo de operaciones a realizar. Como se describió en la sección 3.1, existen cuatro tipos de operaciones básicas de reconfiguración: (i) *creación de instancias de elementos arquitectónicos*, (ii) *destrucción de instancias de elementos arquitectónicos*, (iii) *creación de*

conexiones entre elementos arquitectónicos, y (iv) *destrucción de conexiones entre elementos arquitectónicos*. A su vez, la complejidad de cada una de estas operaciones depende del número de dependencias implicadas en el proceso.

3.6.1.1 Creación de instancias de elementos arquitectónicos

La instanciación de un elemento arquitectónico tiene un coste computacional bajo, puesto que las subtareas a realizar por la plataforma de ejecución son comunes a la instanciación de cualquier entidad ejecutable:

- Búsqueda del tipo del elemento arquitectónico.
- Creación e inicialización de una instancia de dicho tipo.

Esta operación no tiene dependencias con otros elementos arquitectónicos, por lo que tampoco afectará al resto de elementos en ejecución.

3.6.1.2 Destrucción de instancias de elementos arquitectónicos

La complejidad de esta operación dependerá de si la instancia a destruir se encuentra conectada con otros elementos arquitectónicos o no. Si se encuentra conectada, la operación tendrá un coste temporal mayor, pues tendrá que esperar hasta que los servicios en ejecución (servidos y/o solicitados por otros elementos arquitectónicos) finalicen de forma ordenada. Las subtareas que debe realizar la plataforma de ejecución son:

- Notificar a la instancia en ejecución que va a ser destruida, con el objetivo de que ésta finalice sus operaciones en curso y/o realice operaciones previas a su destrucción, como almacenar el estado actual para futuras ejecuciones.
- A su vez, la instancia bloqueará sus puertos para evitar que lleguen nuevas peticiones de servicio. En caso de recibir nuevas peticiones, los puertos serán los encargados de rechazarlas y notificar del hecho de que la instancia va a ser destruida.
- Si hay servicios en ejecución, la instancia deberá finalizarlos primero de una forma segura.

  Una vez la instancia notifique que está lista para ser destruida (o se supere un determinado tiempo límite, para prevenir bloqueos), podrá destruirse la instancia en ejecución y todas las conexiones que estuvieran conectadas a sus puertos.

3.6.1.3 Creación de una conexión (attachment):

No conlleva ningún problema, dado que al detectar los puertos de la existencia de una conexión, estos procederán al envío de las peticiones que tenían pendientes, restableciéndose la comunicación en caso de haber sido

interrumpida previamente o iniciándose una nueva transacción en caso de ser la primera vez.

3.6.1.4 Destrucción de una conexión

Este es el caso más complejo. Requiere que previamente se notifique a los puertos implicados de la necesidad de detener la comunicación. Entonces los nuevos servicios serán pospuestos, mientras que los que están en ejecución continuarán en ejecución. El problema en este caso es cuando el servicio en ejecución requiera de otros servicios o no finalice. En este caso lo más sencillo es definir un temporizador que le dé un tiempo de vida para la finalización de los servicios en ejecución. En caso de finalizar este temporizador, deberá finalizarse forzosamente la conexión.

3.6.2 Características de la infraestructura

Para que la Reconfiguración Dinámica de una arquitectura pueda llevarse a cabo sin perjuicio de la corrección del sistema en ejecución y sin que se vea degradado, deben tenerse en cuenta una serie de consideraciones respecto al diseño de la infraestructura:

- **Los elementos arquitectónicos deben ser independientes**.

  Cada uno de los elementos arquitectónicos: (i) es independiente del resto de elementos arquitectónicos, (ii) se ejecuta en paralelo, y (iii) tiene suficiente autonomía para no verse afectado en caso de caída de las conexiones o de las entidades con las que se comunica.

- **Los puertos deben ser tolerantes a fallos**.

  El punto anterior se apoya sobre esta característica. Los puertos de comunicación de los elementos arquitectónicos deben ser tolerantes a fallos. En primer lugar, debe tolerarse la interrupción del canal de comunicación con carácter temporal (por ejemplo, durante un proceso de reconfiguración). En segundo lugar, debe tolerarse la interrupción definitiva del canal de comunicación (por ejemplo, se ha destruido la conexión o la instancia con la que se comunicaba) para que se tomen las medidas adecuadas (como esperar hasta que un nuevo canal de comunicación sea creado).

- **Las conexiones deben ser independientes entre sí**.

  Si una conexión es eliminada, el resto de conexiones no debe verse afectada por este hecho: deben continuar transmitiendo las peticiones de servicio.

- **Existen mecanismos para notificar la interrupción de las comunicaciones**.

Debe existir un mecanismo para notificar a los elementos implicados (los puertos) que la conexión va a ser detenida temporalmente, con el objetivo de que finalicen las transacciones en ejecución y no se envíen nuevas peticiones. Esto puede realizarse mediante mecanismos de gestión de excepciones distribuidas.

- **Debe soportarse la ejecución transaccional de servicios**.

  El proceso de reconfiguración consta de varias operaciones consecutivas. Si alguna de ellas falla, debe deshacerse todo el proceso. Por tanto, debe implementarse un mecanismo para deshacer las operaciones de reconfiguración aplicadas en caso de error.

La mayoría de las plataformas más extendidas ofrecen escaso soporte a la modificación dinámica de elementos en ejecución. Es por esto que debe desarrollarse una infraestructura que realice la reconfiguración y que la plataforma no ofrece por sí sola. En esta propuesta, esta infraestructura es proporcionada por el middleware PRISMANET [Cos05], [Per05a]. PRISMANET es un middleware implementado en la plataforma .NET que permite la ejecución de arquitecturas PRISMA. Este middleware ofrece los mecanismos necesarios para conocer y modificar la configuración en ejecución de cada instancia de sistemas PRISMA. Como resultado, PRISMANET no ejecuta solamente arquitecturas software PRISMA, sino que también proporciona los principales servicios para ejecutar reconfiguraciones dinámicas. PRISMANET garantiza que las dependencias de configuración entre los diferentes elementos son preservados y que también se genere el código necesario cuando se requiera.

3.7 Reconfiguración Ad-Hoc

Existen dos formas en las que un sistema puede ser reconfigurado, como ya fue introducido por Endler [End92]: reconfiguración programada y reconfiguración ad-hoc. Las reconfiguraciones programadas son cambios preplaneados descritos en el tipo del sistema en tiempo de diseño y que son activados por el propio sistema. Un ejemplo de dicha reconfiguración es destruir una conexión entre dos instancias cuando una de ellas ha dejado de responder, e instanciar una nueva para mantener el funcionamiento del sistema. Los cambios definidos en el caso de estudio son ejemplos claros de reconfiguraciones programadas, dado que todas las reglas de reconfiguración son definidas en el diseño del sistema, previamente a su puesta en ejecución.

Por su parte, las reconfiguraciones ad-hoc son aquellos cambios que son diseñados e introducidos en el sistema mientras éste se encuentra en funcionamiento. Normalmente, dichos cambios se introducen en el sistema manualmente, a través de una herramienta externa, que tiene acceso a

todas las instancias de los elementos arquitectónicos en ejecución y puede reconfigurarlas. Sin embargo, no es apropiado ofrecer reconfiguración ad-hoc a través de una herramienta externa por tres razones.

Primero, un estado inconsistente puede ser alcanzado si la herramienta externa y los mecanismos de reconfiguración programada intentan cambiar el estado del sistema (su configuración) simultáneamente. Segundo, la herramienta externa no respeta el principio de encapsulación, puesto que requiere tener acceso a la topología interna de cada uno de los sistemas de la arquitectura. Tercero, en arquitecturas software distribuidas, no es apropiado que los arquitectos software puedan reconfigurar sistemas remotos si no es a través de las conexiones existentes entre los elementos arquitectónicos y definidos en la arquitectura.

En nuestra propuesta, los dos tipos de reconfiguración son soportados. La reconfiguración programada se ofrece como se ha descrito anteriormente, a través de los aspectos *Configuration* y *EvolutionManager*. Mientras que la reconfiguración Ad-Hoc es ofrecida de forma similar a la anterior, sin necesidad de proporcionar una herramienta externa ni de extender el ADL de PRISMA. Para ello, cuando se desee reconfigurar una arquitectura en tiempo de ejecución manualmente (ad-hoc), bastará con especificar un nuevo componente e introducirlo en la arquitectura en ejecución. Este nuevo componente se conectará al sistema a reconfigurar, concretamente a los puertos de reconfiguración. E invocando a los servicios de reconfiguración del sistema, podrá ir reconfigurándolo. A su vez, dicho componente podrá tener especificado el protocolo de reconfiguración en un aspecto (por lo que será reconfiguración programada), o implementará una interfaz interactiva que permitirá al arquitecto configurar dinámicamente el sistema. Sin embargo, el problema que surge es la seguridad: cualquier elemento arquitectónico podría configurar sistemas que exporten servicios de reconfiguración sin más que conectarse al puerto de reconfiguración.

Por esta razón, puede optarse por exportar o no todos los servicios de reconfiguración, o tan sólo un subconjunto. El arquitecto es libre de elegir el modo, ya que puede hacerlo a través del ADL de PRISMA. Y respecto a la autentificación, bastará con que incorporar, de forma similar a como se ha realizado con el caso de estudio, una clave de sesión por servicio de reconfiguración, y que el componente reconfigurador se autentifique primero.

De esta forma, es imposible que las reconfiguraciones ad-hoc y programada cambien la topología del sistema simultáneamente, puesto que ambas son proporcionadas por la misma entidad: el aspecto *Configuration*. Este aspecto procesa las peticiones de reconfiguración sin importar si los servicios son solicitados por el arquitecto software o disparados por el sistema. Los cambios en la reconfiguración programada son disparados por weavings entre servicios del aspecto *EvolutionManager* y del aspecto *Configuration*.

Los cambios en la reconfiguración ad-hoc podrán ser disparados por el arquitecto software sólo si los servicios de reconfiguración del sistema son exportados a través de un puerto del sistema.

3.8 Conclusiones

Este capítulo ha presentado una nueva propuesta para soportar la reconfiguración dinámica de arquitecturas software, mediante el uso de técnicas orientadas a aspectos.

En esta propuesta se han identificado dos concerns relacionados con la reconfiguración dinámica, que se han encapsulado en dos aspectos denominados *Configuration* y *EvolutionManager*. De esta forma, el código relacionado con la reconfiguración dinámica no se entremezcla con la funcionalidad de la arquitectura, facilitando su mantenimiento y reutilización.

Cada uno de estos aspectos es importado por aquellos componentes complejos que requieran modificar su configuración en tiempo de ejecución. El aspecto *EvolutionManager* permite especificar cómo debe realizarse el proceso de reconfiguración de la arquitectura a un alto nivel de abstracción. Por su parte, el aspecto *Configuration* proporciona al elemento arquitectónico los servicios de reconfiguración dinámica y los servicios que le permiten conocer su configuración en tiempo de ejecución. Mientras que el EvolutionManager describe la transacción de reconfiguración a realizar, el Configuration es el que proporciona la ejecución de cada uno de los servicios que componen la transacción de reconfiguración. Este aspecto es el que proporciona los servicios de reconfiguración a bajo nivel, y que está ligado a la infraestructura de ejecución. Por último, mediante los weavings se definen las reglas que provocarán que se dispare el proceso de reconfiguración.

Las ideas presentadas en este capítulo han sido publicadas en varios congresos. Uno de ellos específico del área de arquitecturas software:

- **Costa C.**, Ali N., Pérez J., Carsí J.A., Ramos I. *Dynamic Reconfiguration of Software Architectures Through Aspects*. Oquendo, F. (ed.): First European Conference on Software Architecture (ECSA'07). Lecture Notes on Computer Science, vol. 4758, pp. 279-283. Springer, Heidelberg, September 2007. ISSN 0302-9743, ISBN: 978-3-540-75131-1

- **Costa C.**, Pérez J., Carsí J.A. *Hacia la construcción de arquitecturas software dinámicas*. En actas de V jornadas de trabajo de DYNAMICA. Valencia, 23-24 noviembre, 2006. ISBN: 84-690-2623-2.

- **Costa C.**, Pérez J., Carsí J.A. *Hacia la reconfiguración dinámica de arquitecturas software orientadas a aspectos*. En actas del IV Taller de Desarrollo de Software Orientado a Aspectos (DSOA'06), junto a XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06). Servicio de publicaciones de la Universidad de Extremadura, Informe técnico TR 24/06, pp. 35-40, septiembre 2006. ISBN: 84-7723-670-4

- **Costa C.**, Pérez J., Ali N., Carsí J.A., Ramos I. *PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos*. En actas de X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05), pp. 27-34. I Congreso Español de Informática (CEDI'05). Granada, septiembre 2005. ISBN 84-9732-434-X

CAPÍTULO IV

EVOLUCIÓN DINÁMICA DE TIPOS ARQUITECTÓNICOS

En el capítulo anterior se ha contemplado el dinamismo de las arquitecturas software desde el punto de vista de la reconfiguración: cómo las arquitecturas software pueden modificar su topología en tiempo de ejecución. Sin embargo, como ya se mencionó en el capítulo anterior, la reconfiguración de una arquitectura está limitada al patrón arquitectónico: no pueden añadirse instancias de componentes que no estuviesen definidos en el patrón arquitectónico ni nuevos tipos de conexiones. Esto es importante para prevenir que el sistema se reconfigure de forma errónea y llegue a perderse la consistencia del sistema.

Sin embargo, en los sistemas abiertos, la reconfiguración dinámica no es suficiente. Se requiere un grado de dinamismo mayor para poder incorporar cambios más drásticos, como incorporar nuevos tipos de componentes o modificar los tipos de conexiones existentes. Este tipo de cambios modifican la semántica del sistema, es la llamada *Evolución de tipos*.

La evolución de tipos puede ser estática o dinámica, dependiendo de si los tipos son evolucionados (cambian su semántica) en tiempo de diseño o mientras éstos se encuentran en ejecución. La mayoría de los sistemas que requieren ser modificados no necesitan el soporte a la evolución dinámica de tipos. En el campo de las arquitecturas software, la actualización o modificación del tipo de un componente se lleva a cabo mediante el reemplazo de la vieja versión por la nueva. Si la infraestructura de ejecución soporta reconfiguración dinámica, sustituir el viejo componente por el nuevo podrá hacerse en tiempo de ejecución, sin más que detener las conexiones existentes. Si no soporta reconfiguración dinámica, deberá detenerse por completo todo el sistema para poder instanciar el nuevo tipo de componente.

Sin embargo, en muchos casos, el reemplazo de un componente en tiempo de ejecución no es suficiente si uno de los requisitos del sistema es preservar una alta disponibilidad. Al eliminar una instancia en ejecución, se pierde el

estado y muchas de las transacciones que estaban en ejecución deben ser reiniciadas. En arquitecturas orientadas a aspectos, como PRISMA, en las que los diferentes concerns se encuentran claramente diferenciados, la actualización de un solo concern (o añadir un nuevo concern) implicaría detener todos los concerns en ejecución y perder su estado, aunque dichos concerns no se viesen afectados por los cambios.

Por otro lado, introducir un *nuevo tipo* de componente en la arquitectura normalmente requiere detener por completo el sistema en ejecución, puesto que dicho tipo no existía en el patrón arquitectónico y es necesario actualizarlo antes de poder crear instancias del nuevo componente. En este caso, también todos los elementos en ejecución de la arquitectura se ven afectados.

Es en estos casos en los que es necesario el soporte a la *Evolución Dinámica de Tipo*s. La Evolución Dinámica de Tipos Arquitectónicos persigue los siguientes objetivos:

- Actualizar los tipos arquitectónicos (componentes, conectores y sistemas) en ejecución (i.e.: tienen instancias en ejecución) y propagar el cambio a toda la población de instancias, para que sean consistentes con el nuevo tipo.
- La actualización de las instancias en ejecución debe hacerse preservando en la medida de lo posible su estado, es decir, sólo debe afectar a aquellas partes realmente modificadas. Por ejemplo, en el caso de evolucionar la topología de una arquitectura para añadir un nuevo tipo de componente, no debe provocar que todos los componentes en ejecución se vean afectados por el cambio.
- La actualización de las instancias debe minimizar el impacto del cambio al resto de elementos arquitectónicos conectados a la instancia que está siendo modificada.

En este capítulo se presentará una propuesta para soportar la evolución dinámica de tipos arquitectónicos en arquitecturas PRISMA. Para ello, en primer lugar se presentará el estado del arte en el campo de la evolución dinámica de tipos, concretamente del área de las arquitecturas software. En segundo lugar se presentará un caso de estudio con el cual se ilustrarán los conceptos de la propuesta. En tercer lugar se presentarán los conceptos de la propuesta y la infraestructura necesaria para soportarla.

4.1 Estado del arte

En las últimas décadas ha habido un interés creciente en la investigación de los procesos de evolución del software. Por un lado, las líneas de trabajo tradicionales han seguido estudiando la evolución (estática) del software con la finalidad de reducir el tiempo y el coste del proceso de mantenimiento. Por otro lado, otros grupos de trabajo se han centrado en el estudio de la evolución dinámica, con la finalidad de proporcionar mayor flexibilidad a los sistemas software. Por esta razón, existen en la literatura distintos enfoques, dependiendo del área en la que se hayan aplicado. En esta sección se describen algunos de los trabajos más relevantes dentro del contexto que nos ocupa, los relacionados con el AOSD y las arquitecturas software.

Los trabajos de adaptabilidad propuestos por la comunidad de Desarrollo de Software Orientado a Aspectos aportan en su mayoría mecanismos para enlazar aspectos dinámicamente al código base en ejecución. La mayor parte de los trabajos han sido enfocados para soportar AOP en la Programación Orientada a Objetos. Dichas aproximaciones son normalmente desarrolladas en Java y .NET y son muy dependientes de la plataforma. SetPoint [Bra07] permite la agregación y eliminación dinámica de aspectos. El proceso de weaving se basa en la evaluación de los predicados lógicos sobre la meta-información añadida al código base. Rapier-Loom.NET [Sch03] y EOS [Raj03] permiten la agregación/eliminación dinámica de aspectos, pero las definiciones de weaving se definen dentro de los aspectos, perdiendo entonces su reusabilidad. El trabajo de Yang et al. [Yan02] permite la definición de reglas de adaptación (i.e. weavings) fuera del código base, y la agregación/eliminación de nuevo código (i.e. aspectos) a través de la evaluación de dichas reglas. Sin embargo, todas estas aproximaciones tienen en común que (i) enlazan los aspectos a nivel de instancias en lugar de a nivel de tipos, y (ii) no pueden cambiar el código base, tan sólo pueden extenderlo con nuevo comportamiento.

En el área de las arquitecturas software hay numerosos trabajos que abordan la adaptabilidad en tiempo de ejecución, pero la mayor parte de ellos se limitan a la reconfiguración dinámica [Bra04], [Cue02]. Además, en el área de las arquitecturas software, la evolución de tipos no ha sido tratada a nivel arquitectónico. La práctica común cuando un elemento arquitectónico en ejecución necesita ser modificado es reemplazarlo por otro nuevo, perdiéndose por tanto su estado previo. MARMOL [Cue01], [Cue02] es un modelo formal que define una meta-arquitectura genérica para proporcionar a los ADLs conceptos de Reflexión Computacional [Mae87]. La idea principal es proporcionar al sistema con una representación editable de sí mismo. De esta forma, los cambios realizados a esta representación son reflejados en el sistema en ejecución. Sin embargo, este trabajo solamente formaliza los conceptos de Reflexión requeridos, y no describe la infraestructura necesaria para soportar dichos conceptos. El proyecto

ArchWare [Mor04], [Oqu04b] proporciona un prototipo basado en un lenguaje formal y ofrece un soporte completo para la evolución dinámica de las arquitecturas software. La adaptabilidad en tiempo de ejecucióln es realizada mediante el Evolution Meta-Process Model [Bal05]. Cada componente ArchWare se compone de un proceso productivo (que proporciona el comportamiento del componente), y un proceso de evolución. Este proceso de evolución evoluciona y controla el proceso productivo del componente. Sin embargo, no hay ninguna referencia en los documentos del proyecto acerca de cómo el proceso de evolución es capaz de evolucionar los tipos de los componentes. Todos los ejemplos proporcionados siempre se basan en el nivel de instancias.

También existen en la literatura aproximaciones que proporcionan evolución dinámica y combinan AOP y arquitecturas software. JAsCo [Suv03] introduce el concepto de conectores para el weaving entre los aspectos y el código base, que permite un alto nivel de reusabilidad para los aspectos. Además, JAsCo proporciona un lenguaje expresivo que permite la definición de relaciones entre los aspectos. Sin embargo, debido al hecho de que los aspectos se enlazan de una forma referencial, esta propuesta requiere una plataforma de ejecución intermedia para interceptar la aplicación destino e insertar los aspectos en tiempo de ejecución. De una forma similar, CAM/DAOP [Pin05] es un enfoque de arquitectura software que introduce aspectos como conectores especiales entre componentes. Sin embargo, aunque soporta el enlace dinámico de aspectos, no soporta la agregación de nuevos *tipos* de aspectos en tiempo de ejecución (no planeados en tiempo de diseño).

El trabajo de [Gus04] presenta un enfoque similar a nuestra propuesta, ya que propone encapsular en un aspecto la funcionalidad que proporciona evolución de tipos. La unidad del cambio son las clases Java, y el aspecto es el que, actuando como un proxy, redirige las peticiones a la versión más reciente del código modificado. El trabajo de [Wan06] describe de forma bastante completa cómo lograr la evolución de tipos en tiempo de ejecución. Su propuesta es muy similar al trabajo de Gustavsson, en el sentido de que también emplea proxies para mantener el versionado de las distintas clases en ejecución. Dichas propuestas tienen en común que son muy dependientes de la implementación, concretamente de Java.

4.2 Caso de estudio

Con el objetivo de ilustrar esta propuesta se utilizará un caso de estudio sencillo. La especificación completa del caso de estudio se encuentra en [Cab05]. Este caso de estudio consiste en un brazo robot cuyos movimientos son controlados por diferentes articulaciones: Base, Shoulder (hombro), Elbow (codo) y Wrist (muñeca). El funcionamiento de cada una de las articulaciones es prácticamente idéntico, tan sólo varían los valores máximo y mínimo que cada articulación admite (en grados) y en cómo se envía la orden de movimiento al controlador hardware.

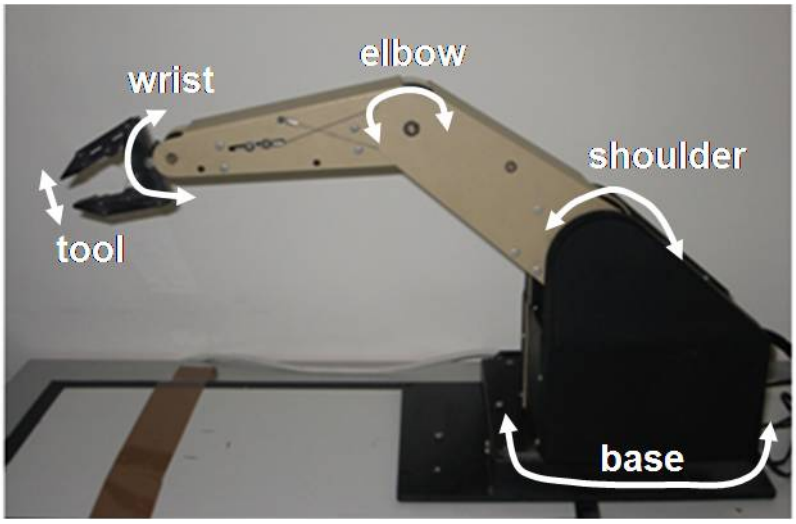

Figura 13 – Robot TeachMover

Por esta razón, todas las articulaciones se han modelado como instancias de un componente de tipo Joint (articulación). En el momento de creación de cada instancia, se proporcionan los valores de inicialización (parámetros de configuración) de cada articulación. El comportamiento del componente Joint es definido por un conjunto de aspectos, de entre los cuales tan sólo se tendrán en cuenta dos de ellos: un aspecto funcional, Fun, que define cómo se envían los movimientos al controlador hardware, y un aspecto de seguridad, Saf, que comprueba que los movimientos del componente Joint se encuentran entre los rangos máximo y mínimo admitidos por la articulación. Los servicios que este componente ofrece se exportan al resto de componentes de la arquitectura a través del puerto OperPort (ver Figura 14).

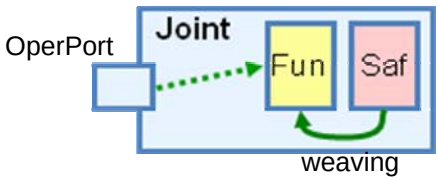

Figura 14 – Componente Joint

```
Component Joint
     Functional Aspect Import Fun;
     Safety Aspect Import Saf;

     Weavings
       Fun.movejoint(NewHalfsteps, Speed)
         afterIf (Secure = true)
       Saf.Check(NewHalfsteps, Secure);
     End_Weavings;

     Ports
       OperPort: IMotionJoint, Played_Role Fun.ACT;
     End_Ports;

     new(input Argmin: integer, input Argmax: integer)
       {Fun.begin();
        Saf.begin(input InitMinimum: integer,
                  input InitMaximum : integer); }
     destroy(){
       Fun.end();
       Saf.end();}
End_Component Joint;
```

Figura 15 – Especificación PRISMA del componente Joint

Uno de los requisitos no funcionales más importantes del componente Joint es que va a ser ensamblado dentro de un sistema software de alta disponibilidad. Este sistema software desempeña tareas críticas, por lo que su funcionamiento debe ser continuo y sin interrupciones. Todos los elementos que se integren en dicho sistema deberán ser capaces de actualizarse en tiempo de ejecución, sin perturbar al resto del sistema en ejecución.

Para cumplir con este requisito, el componente Joint ha sido desarrollado con una infraestructura capaz de hacerle evolucionar en tiempo de ejecución sin forzar al resto del sistema en el que se ejecuta a ser reiniciado.

Como era inevitable, al tiempo de estar el componente Joint implantado y en ejecución en el sistema destino, nuevos requerimientos emergen: se requiere la inclusión de un servicio de emergencia para parar instantáneamente todos los movimientos de las articulaciones. Este nuevo requerimiento afecta al aspecto de seguridad, Saf. Para ello, una nueva versión del aspecto de seguridad, Saf2, ha sido desarrollada y debe implantarse en el sistema en ejecución. Dado que el proveedor del software no tiene acceso al sistema en ejecución, un nuevo tipo de componente, Updater, ha sido desarrollado para reemplazar en tiempo de ejecución el viejo aspecto (Saf) por el nuevo (Saf2), sin intervención alguna.

En los siguientes apartados se irá describiendo el proceso mediante el cual este componente solicita la adaptación del component Joint y cómo este cambio es propagado a todo el conjunto de instancias en ejecución de dicho componente.

4.3 Modelado de la Evolución de Tipos

La infraestructura para la evolución dinámica de tipos propuesta en este trabajo se basa en una serie de conceptos, los cuales son descritos a continuación.

4.3.1 Niveles MOF

En primer lugar, para poder evolucionar componentes debe distinguirse claramente entre tipos e instancias de componentes, ya que se ubican en distintos niveles de abstracción. La especificación Meta-Object Facility (MOF) de OMG [OMG02] permite distinguir claramente entre tipos e instancias de una forma sencilla y elegante. Su principal objetivo es la gestión de las descripciones de modelos a distintos niveles de abstracción y su modificación estática. MOF define una "arquitectura" de tres niveles, centrada en el enfoque del Desarrollo Dirigido por Modelos [OMG03].

Básicamente, cada uno de los niveles define el lenguaje en el que se construirán los términos del nivel inmediatamente inferior, y así sucesivamente. En MOF, el nivel superior (M3) es también el más abstracto (ver nivel M3, Figura 16). Este nivel define el lenguaje abstracto utilizado para describir las entidades del nivel inmediatamente inferior (los metamodelos). La especificación MOF propone el modelo MOF como el lenguaje abstracto para definir todos los tipos de metamodelos, como UML o PRISMA.

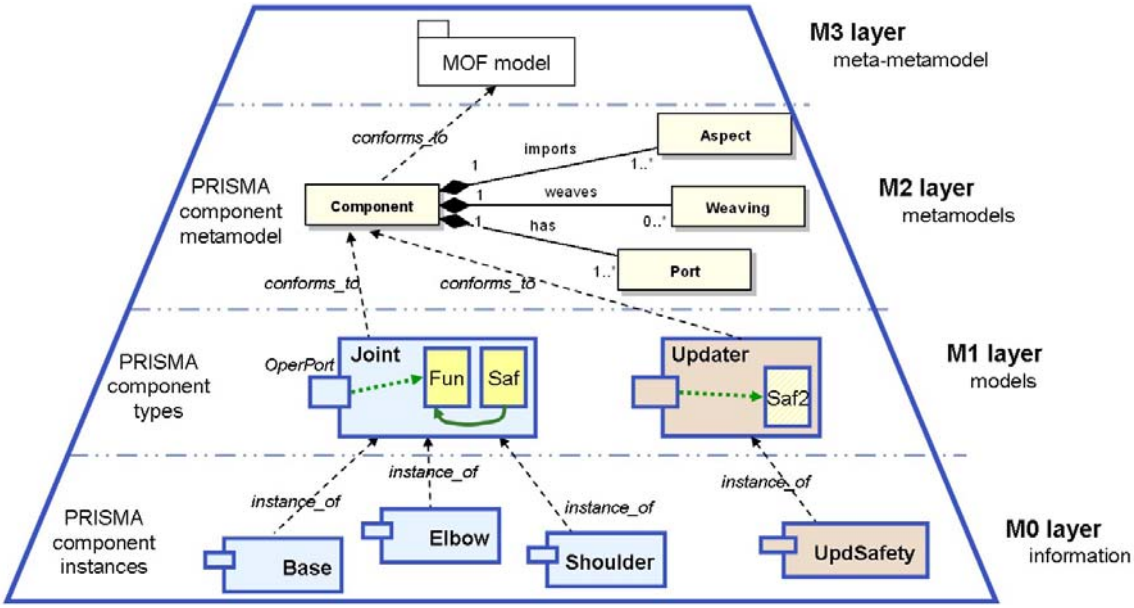

Figura 16 – Niveles de la Meta-Object Facility (MOF) y componentes PRISMA

El nivel M2 (definición de metamodelos) define la estructura y semántica de los modelos situados en el siguiente nivel inferior. El metamodelo PRISMA es

definido a este nivel: los componentes PRISMA son una agregación de Aspectos, Puertos y Weavings (ver nivel M2, Figura 16). El comportamiento de un componente se define mediante la importación de aspectos y su sincronización a través del uso de weavings. Los servicios publicados son definidos a través del uso de puertos.

El nivel M1 comprende los modelos que describen los datos. Estos modelos son descritos utilizando las primitivas y relaciones descritas en el nivel de metamodelado (M2). Los tipos de componentes PRISMA (como Joint o Updater) se ubican en el nivel M1 (ver nivel M1, Figura 16). Por ejemplo, el componente Joint importa dos aspectos: un aspecto Funcional y uno de Seguridad (ver los aspectos Fun y Saf, nivel M1, Figura 16). Dichos aspectos se sincronizan a través de un conjunto de weavings.

Por último, debido a las particularidades del metamodelo PRISMA (descrito en el nivel M2) que define que un tipo de componente puede ser instanciado, existe un nivel más que en MOF no se contempla como tal. Es el nivel M0 (ver Figura 16) y contiene las distintas instancias en ejecución de los tipos de componente (e.g. Base, Elbow, UpdSafety, etc.). Dichas instancias se comportan como se describe en sus tipos respectivos.

4.3.2 Reflexión Computacional

Sin embargo, la especificación MOF fue diseñada para especificar y gestionar metamodelos desde un punto de vista estático. MOF no describe cómo tratar la adaptación dinámica de sus elementos en tiempo de ejecución. Por esta razón, el concepto de Reflexión Computacional [Smi82], [Mae87] es empleado para proporcionar adaptación dinámica a los modelos (en este caso, tipos de componentes). La Reflexión Computacional se define como la capacidad de un sistema software de razonar sobre sí mismo y actuar sobre sí mismo. Para hacer esto, un sistema debe tener una representación de sí mismo que sea editable y que esté causalmente conectada a sí mismo. De este modo, los cambios realizados en dicha representación (que es gestionada como un dato) serán *reflejados* en el sistema y viceversa.

De este modo, un sistema tiene dos vistas de sí mismo: la *base-view* y la *meta-view* (ver Figura 17). La *base-view* "ejecuta" el comportamiento de la lógica de negocio del sistema y modifica el conjunto de valores que definen el estado del proceso. La *meta-view* define cómo se comporta el sistema, es una "descripción" del sistema. Esta vista permite al sistema cambiar su comportamiento mediante la modificación de su representación. El proceso de obtener una representación editable del sistema (la *meta-view*) se denomina *reificación*, y el proceso contrario se denomina *reflexión* (ver Figura 17). La principal ventaja de la reflexión computacional es el hecho de

que permite describir la auto-adaptación de un sistema de una forma simple y natural.

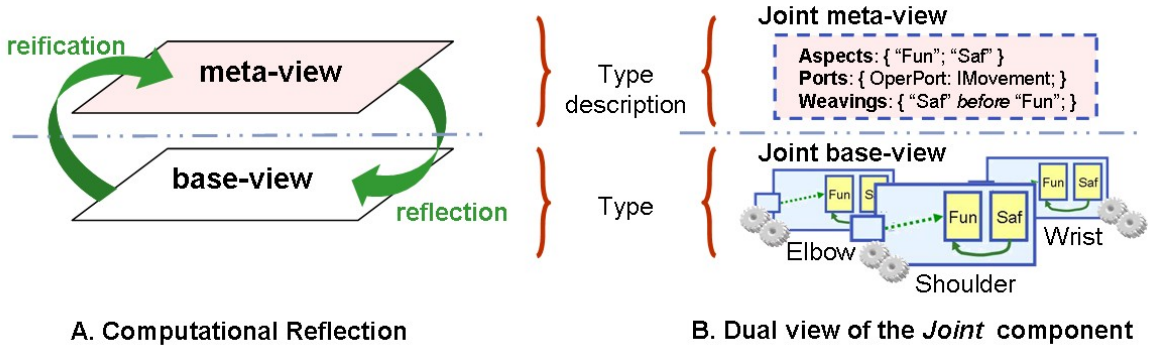

Figura 17 – Vista dual de un sistema reflexivo

El metamodelo PRISMA describe la estructura y el comportamiento de un componente mediante aspectos, weavings y puertos (ver nivel M2, Figura 16). Por esta razón, la *meta-view* asociada al tipo de un componente PRISMA es una estructura editable cuyos términos son aspectos, puertos y weavings. Por ejemplo, la *meta-view* del tipo del componente Joint describe dicho componente como compuesto de: (i) los aspectos Fun y Saf, (ii) un puerto OperPort, y (iii) un weaving entre los aspectos Fun y Saf (ver descripción del tipo, Figura 17). La *base-view* asociada al tipo de un componente PRISMA es la "ejecución" de los aspectos, puertos y weavings "descritos" en la *meta-view*. Por ejemplo, la *base-view* del tipo del componente Joint es el conjunto de todas sus instancias: Base, Elbow, Shoulder, etc. (ver Tipos, Figura 17).

4.3.3 MOF + Reflexión Computacional

Los niveles de abstracción proporcionados por MOF y la capacidad para describir la auto-adaptación proporcionada por la Reflexión Computacional nos permite definir la infraestructura necesaria para adaptar dinámicamente la estructura interna de un componente. En este trabajo, únicamente nos centraremos en tipos de componentes (nivel M1) e instancias de componentes (nivel M0). Cada tipo de componente tiene una vista dual: la *base-view* y la *meta-view*. Sin embargo, cada vista se encuentra en un nivel diferente MOF (ver Figura 18), como se describe a continuación.

Cada instancia de componente (e.g.: Elbow, Wrist, etc.) es un proceso en ejecución que tiene su propio estado y se comporta como el tipo del componente (e.g.: Joint) especifica. De este modo, el comportamiento de la instancia es "proporcionado" por la *base-view* de su tipo (i.e. el tipo en ejecución, ver $base\text{-}view_1$ en M0, Figura 18). Este comportamiento es "descrito" por la *meta-view* del tipo en el nivel inmediatamente superior (ver $meta\text{-}view_1$ en M1, Figura 18).

A su vez, el tipo de un componente puede verse como un proceso en ejecución que tiene también su estado y comportamiento: (i) el estado es la *meta-view* del tipo del componente (una representación editable de sí mismo), y (ii) el comportamiento es la *base-view* de un componente PRISMA (ver *base-view*$_2$ en el nivel M1, Figura 18), que se compone de un conjunto de servicios para gestionar el estado del tipo. Este conjunto de servicios son los denominados servicios de evolución, porque son los que cambian la descripción del tipo del componente. En PRISMA, dichos servicios de evolución son: *addAspect*, *removeAspect*, *addPort*, etc. y son "descritos" a su vez en la *meta-view* del Componente PRISMA en el nivel M2 (ver *meta-view*$_2$ en M2, Figura 18).

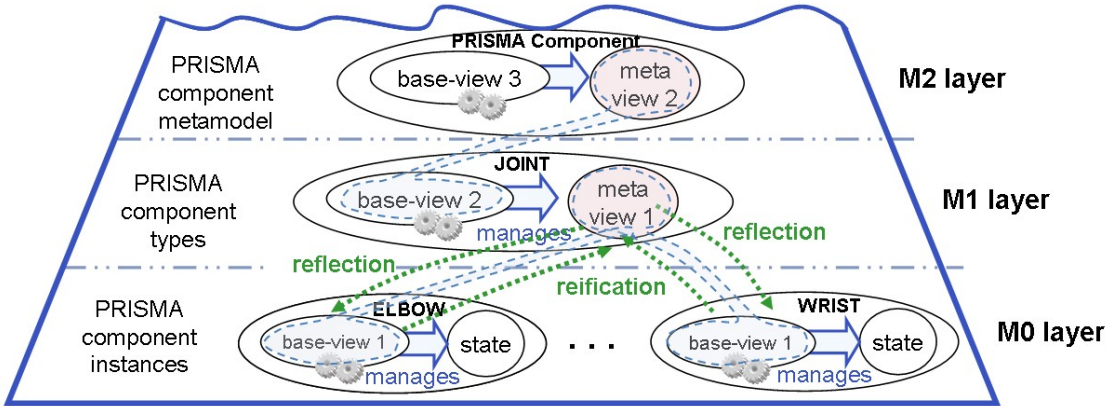

Figura 18 – Vista dual de los tipos de componentes reflexivos

Los servicios de evolución son proporcionados y ejecutados por cada tipo de componente (en nuestro caso de estudio, el componente Joint). Esto es debido a que la *base-view* de un Componente PRISMA es parte de cada tipo de componente (ver la *base-view*$_2$ de Joint en el nivel M1, Figura 18). A medida que un servicio de evolución cmabia la representación interna del tipo de un componente (la *meta-view*), dichos cambios son también reflejados en la *base-view* del tipo del componente. Esto significa que cada instancia de componente tendrá su estructura y comportamiento actualizado de acuerdo con los cambios realizados en la *meta-view* del tipo del componente. Por ejemplo, como consecuencia de la relación de *reflexión* (ver reflection, Figura 18), la ejecución del servicio de evolución `addAspect("Saf2")` en la *meta-view* del tipo Joint disparará la adición del aspecto Saf2 en cada instancia del tipo Joint.

El modelo descrito aquí permite la descripción del proceso de adaptación dinámica desde un alto nivel de abstracción. Sin embargo, hay algunas cuestiones en el proceso de reflexión y reificación que deben ser tratados en cada implementación específica. El proceso de reificación debe tener en cuenta cómo obtener el tipo y la estructura interna de una instancia en ejecución de un componente. El proceso de reflexión debe tener en cuenta: (i) cómo difundir los cambios realizados a la *meta-view* del tipo a todas sus instancias en ejecución, y (ii) cómo cambiar la estructura interna de cada instancia de componente en ejecución sin afectar aquellas partes de la

estructura que no han sido modificadas. Por esta razón, en la siguiente sección se describen los mecanismos propuestos que debe proporcionar la infraestructura para tratar con estas cuestiones.

4.4 Evolución Dinámica de Tipos Arquitectónicos

Una vez los conceptos de Reflexión Computacional y MOF han sido introducidos, a continuación se presenta en detalle la propuesta para la adptación dinámica de componentes orientados a aspectos. La adaptación dinámica de la estructura interna de los componentes es disparada cuando cualquier servicio de evolución proporcionado por un tipo es invocado. Entonces, el servicio de evolución modifica la descripción del tipo de componente (la *meta-view* del tipo) y la relación de reflexión realiza la adaptación interna de sus instancias.

4.4.1 Evolución del tipo de un componente

Los sistemas autónomos y heterogéneos requieren que cada uno de sus componentes implemente sus propios mecanismos de adaptación de una forma descentralizada. Por esta razón, el principal objetivo de esta propuesta es proporcionar la adaptación interna de los componentes en tiempo de ejecución de una forma descentralizada y autónoma. La adaptación descentralizada se consigue al no depender de una entidad centralizada que gestione el proceso de evolución de *todos* los tipos arquitectónicos de la arquitectura software. Cada tipo arquitectónico es la única entidad responsable de sus instancias y es la única capaz de evolucionarlas. La adaptación autónoma se consigue en el sentido de que las *instancias* proporcionan por sí mismas la infraestructura para ser evolucionadas dinámicamente de una forma segura. Cada instancia mantiene su propio estado y es la única capaz de decidir el mejor momento para aplicar las adaptaciones.

4.4.1.1 Servicios de evolución proporcionados por los tipos

Los servicios de evolución que el tipo de un componente ofrece depende de la tecnología interna en la que ha sido implementado (en un estilo imperativo, o en cualquier lenguaje declarativo o formal). Sin embargo, es posible identificar aquellas partes del componente que son independientes de la tecnología. Las principales partes de un componente que son independientes de la tecnología son: (i) comportamiento y estado, (ii) puertos, y (iii) interacciones internas entre los distintos procesos de un componente. En PRISMA, el comportamiento y el estado son proporcionados mediante la

composición de aspectos, los puertos mediante los puertos del componente, y las interacciones internas mediante los weavings (que sincronizan la ejecución de los aspectos). De este modo, los servicios que el tipo de un componente debería proporcionar son aquellos que permiten modificar las principales partes de un componente. Podemos distinguir dos tipos de servicios de evolución: servicios de Evolución de Tipos y servicios de Introspección.

Los servicios de Evolución de Tipos son aquellos relativos a la modificación del tipo, como adiciones o eliminaciones de partes del componente. En el caso de PRISMA, los servicios de Evolución de Tipos principales son:

`AddAspect(), RemoveAspect(), AddPort(), RemovePort(), AddWeaving() y RemoveWeaving().`

Los servicios de Introspección son aquellos servicios de evolución que permiten conocer la estructura interna de un componente y proporcionan información sobre las partes que lo componen. En el caso de PRISMA, algunos de los servicios de Introspección proporcionados son:

`GetAspects(), GetWeavings() y GetPorts().`

4.4.1.2 Estructura reflexiva de los tipos

El proceso de evolución puede ser disparado por la lógica de negocio o por un usuario del sistema. Ambos pueden ser representados en la arquitectura mediante componentes. De este modo, la necesidad de evolucionar un componente específico emerge de otro componente que invoca en tiempo de ejecución los servicios de evolución del componente a ser modificado. En nuestro caso de estudio, la instancia UpdSafety invocará los servicios de evolución del tipo Joint para introducir los nuevos requisitos de seguridad en el sistema.

Sin embargo, para poder invocar los servicios de evolución, debe existir un enlace desde el nivel de instancias (M0) al nivel de los modelos (M1), es decir, el enlace entre la *base-view* de UpdSafety y la *meta-view* del Joint. Este enlace se ha denominado *reification link* (ver *reification_link* en Figura 19) debido a que (i) es un enlace hacia arriba entre los niveles y (ii) permite a las instancias invocar los servicios de modificación de tipos, que solamente están disponibles en un nivel superior. El reification link debe ser proporcionado por cualquier ADL en el que deba soportarse la evolución dinámica de tipos arquitectónicos. Existen distintas formas en un reification link puede ser expresado sintácticamente. En PRISMA, éste es descrito especificando el nombre del tipo del componente que tiene que ser evolucionado, seguido por el operador punto "." y el servicio de evolución a ser invocado. Por ejemplo, la instancia UpdSafety añade el aspecto "Saf2" al componente Joint de esta forma:

```
Joint.AddAspect("Saf2")
```

"Joint" es el tipo a ser evolucionado, y "AddAspect" es el servicio de evolución a ser ejecutado. El reification link es expresado mediante el operador "." Se ha elegido esta sintaxis porque es auto-descriptiva: los tipos proporcionan sus propios servicios de evolución.

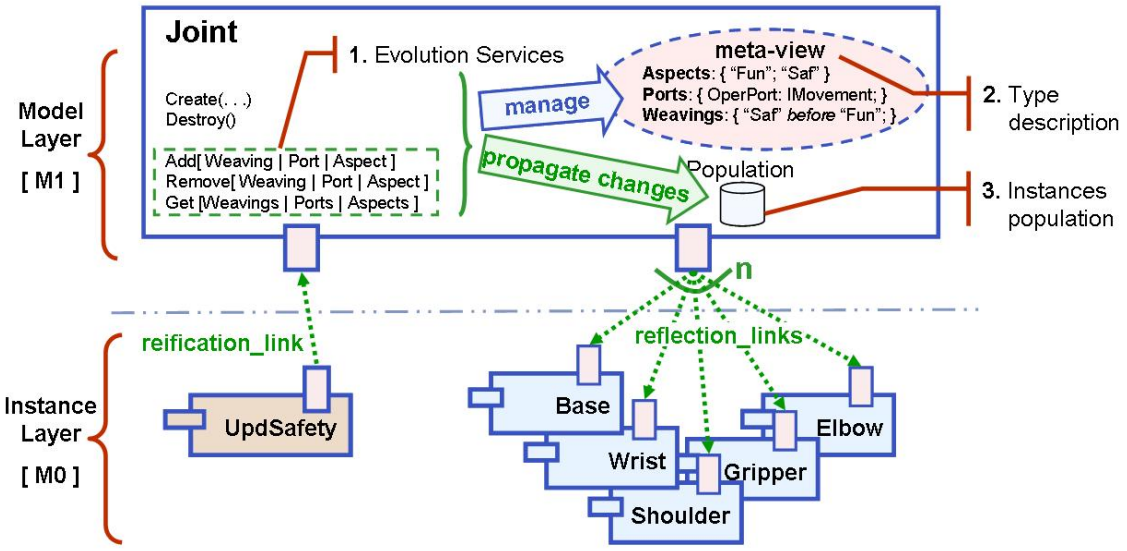

Figura 19 – Infraestructura reflexiva para la adaptación de componentes

El tipo de un componente es una factoría de instancias: es responsable de la creación y destrucción de sus instancias. Por esta razón, el tipo de un componente gestiona la población de sus instancias, esto es, una referencia a cada instancia en ejecución (ver nota 3, Figura 19). La población de instancias es mantenida normalmente por la plataforma de ejecución (e.g. el recolector de basura en Java o .NET). Sin embargo, el acceso a dicha información es necesaria para poder propagar los cambios realizados al tipo a cada una de sus instancias posteriormente. En nuestro caso de estudio, la población del tipo Joint se compone de las instancias que han sido creadas en la configuración incial: Base, Elbow, Shoulder, etc.

El tipo de un componente proporciona sus propios servicios de evolución para cambiarse a sí mismo (ver nota 1, Figura 19). Dichos servicios modifican (o obtienen información de) la *meta-view* del tipo del componente (ver nota 2, Figura 19). Cada tipo es responsable de mantener la reificación de su *meta-view* actualizada mientras se encuentra en ejecución. Dependiendo de la implementación, la *meta-view* puede ser representada en términos del ADL o directamente en código dependiente de la plataforma (como C#). En el prototipo desarrollado, la estructura reificada se ha representado en C# para facilitar su manipulación fácilmente por el código.

De este modo, cuando la instancia UpdSafety invoca el servicio de evolución `AddAspect("Saf2")` (a través del enlace de reificación, ver Figura 19), la *meta-view* del Joint es actualizada. Como consecuencia de modificar la *meta-view*, los cambios son reflejados en las instacias, es decir, son propagados a

toda la población del tipo. Esto es realizado en dos etapas. La primera etapa consiste en almacenar los cambios realizados a la *meta-view* (i.e. la representación del tipo Joint) para poder hacerlos persistentes, mediante la generación del ADL resultante o del código dependiente de la plataforma. El efecto inmediato es que la lista de aspectos `{"Fun"; "Saf"}` de la *meta-view* es actualizada por `{"Fun"; "Saf2"}`. De este modo, las nuevas instancias del componente Joint serán creadas (mediante la ejecución del servicio `Joint.Create(…)`) utilizando el tipo actualizado.

El siguiente paso es propagar los cambios a las instancias del tipo. Del mismo modo que con los reification links, debe existir un enlace desde el nivel de modelos (M1) al nivel de instancias (M0) para *reflejar* los cambios en cada instancia en ejecución. Dichos enlaces se han denominado *reflection links* (ver *reflection_link* en Figura 19). Un reflection link es creado por cada referencia a una instancia del componente (la población).

4.4.2 Evolución de instancias de componentes

Los tipos de los componentes no realizan directamente los cambios evolutivos sobre sus instancias. Dichos cambios deben ser ejecutados internamente por cada instancia por dos motivos principales.

Por un lado, cada instancia tiene que decidir cuándo y cómo ejecutar sus cambios para realizar las modificaciones de una forma segura. Esta modificación segura es necesaria para: (i) asegurar que aquellas partes que no van a ser modificadas no se vean afectadas por el proceso de modificación; y (ii) garantizar que la modificación es ejecutada una vez las transacciones en ejecución han finalizado y todos los procesos internos alcanzan un estado seguro. Por esta razón, cada instancia de componente (e.g. Base, Elbow, Shoulder, etc.) es provista de un aspecto *Evolution Planner*, cuya misión es supervisar proceso de actualización de la instancia a la cual pertenece.

Por otro lado, el proceso de adaptación sólo tiene sentido en el nivel de instancias, porque las adaptaciones en tiempo de ejecución que preservan el estado son muy dependientes de la tecnología (e.g. detener hilos de ejecución, modificar áreas de memoria, cargar/descargar tipos en memoria, etc.) Por esta razón, cada instancia es compuesta por dos aspectos dependientes de la tecnología: el *Actuator*, que es el aspecto que realmente realiza los cambios en la estructura interna del componente, y el *Sensor*, que proporciona información sobre qué está sucediendo en la instancia del componente.

4.4.2.1 El aspecto Evolution Planner

La clave para alcanzar la adaptación dinámica de la estructura interna de las instancias de componentes es maximizar la independencia entre las partes internas de un componente que puede sufrir cambios. De esta forma, el reemplazar la parte interna de un componente tiene un impacto mínimo en las otras partes en ejecución. Aquellas partes que son dependientes en algún grado con la que parte que está siendo modificada tan sólo necesitarán ser detenidas temporalmente mientras los cambios son realizados.

Una dependencia de evolución se define como una relación binaria sobre el conjunto de las partes internas de un componente (P). Una parte interna $x \in P$ tiene una dependencia de evolución con otra parte interna $y \in P$, definida como $EV_{DEP}(x,y)$, si cualquier cambio en x causa un cambio en y. La cantidad total de dependencias de evolución posibles sobre el conjunto de partes internas P se define como la permutación matemática con repetición de P: $|P|^2$. Como un componente PRISMA está compuesto de tres tipos de partes (P_{PRISMA} = *{ports, weavings, aspects}*), el conjunto de dependencias de evolución posibles que un componente PRISMA puede tener es nueve (i.e.: Ev_{DEP}={ (port,aspect), (port,weaving), (port,port), …}). Sin embargo, debido al alto nivel de reutilización proporcionado por PRISMA, realmente tan sólo hay dos dependencias de evolución entre las partes internas: Ev_{DEP}(*aspect,port*) y Ev_{DEP}(*aspect*, *weaving*). Dichas dependencias de evolución son tenidas en cuenta por el aspecto Evolution Planner.

El aspecto Evolution Planner (ver nota 3, Figura 20) tiene el conocimiento de cómo adaptar la estructura interna de una instancia de componente de una forma segura. Por esta razón, es consciente de qué tipo de dependencias de evolución entre las partes internas de un componente pueden ocurrir al aplicar un cambio en tiempo de ejecución. La acción a realizar es distinta dependiendo del tipo de cambio a realizar. borrados, modificaciones o inserciones. En el borrado de apectos, los puertos y weavings relacionados necesitan también ser eliminados. En el reemplazo o modificación de aspectos, los puertos y weavings relacionados sólo necesitarán ser borrados si los puntos de dependencia entre el aspecto y los puertos y weavings son modificados. En otras palabras: (i) un puerto será borrado si la interfaz que publica es eliminada del aspecto modificado, y (ii) un weaving será borrado si el método que intercepta o dispara es modificado por el aspecto que está siendo evolucionado. Finalmente, la inserción de aspectos, weavings y puertos puede realizarse sin comprometer el resto de partes del componente en ejecución o sus comunicaciones, debido a que aún no hay relaciones entre ellos.

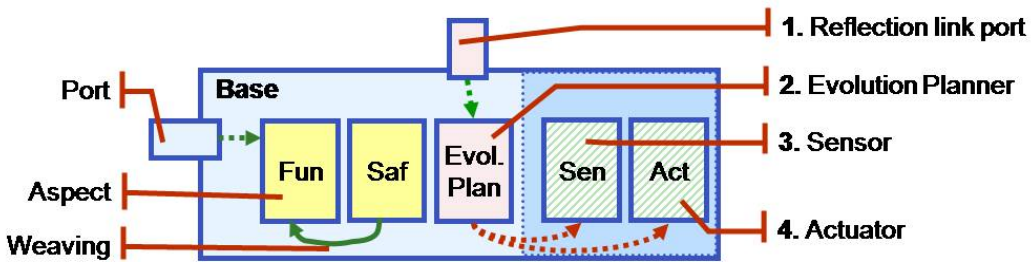

Figura 20 – Estructura interna de una instancia de componente

El Evolution Planner *refleja* los cambios que son realizados en la *meta-view* del tipo del componente a las instancia a la cual el Evolution Planner pertenece. Por esta razón, el aspecto Evolution Planner proporciona los mismos servicios de evolución que el tipo del componente, aunque dichos servicios son únicamente aplicados a nivel de instancias. Los cambios a ser aplicados son recibidos a través de los *reflection links* (ver nota 1, Figura 20). Estos cambios corresponden a los servicios de evolución que la instancia UpdSafety ha aplicado en la *meta-view* del componente Joint y que han sido propagados al Evolution Planner de la instancia Base a través de un *reflection link*:

```
Remove Aspect("Saf"); AddAspect("Saf2"); AddWeaving(…)
```

El Evolution Planner aplicará dichos cambios mediante la coordinación de las acciones a ser realizadas por los aspectos Actuator y Sensor. Esta coordinación es realizada de acuerdo con los protocolos de adaptación correctos que conoce.

4.4.2.2 Los aspectos Actuator y Sensor

El aspecto Actuator (ver nota 4, Figura 20) es el responsable de aplicar los cambios en la instancia en ejecución: mediante la generación y el enlace de código, la creación de elementos dinámicamente, la invocación de mecanismos de adaptación a bajo nivel, etc. Este aspecto proporciona además un conjunto de servicios adicionales para indicar a las partes del componente que va a realizarse un cambio y que por tanto han de alcanzar un estado seguro. Dichos servicios son:

```
StopAspect(), StopPort(), StopWeaving(),
StartAspect(), StartPort() y StartWeaving().
```

Dichos servicios son necesarios en caso de que esté presente una dependencia de evolución, para evitar interacciones entre la parte del componente que está siendo cambiada y las partes dependientes.

El Sensor (ver nota 2, Figura 20) es un aspecto que proporciona servicios para supervisar qué está ocurriendo en la instancia en ejecución: cuándo un aspecto ha sido realmente agregado a la instancia del componente en ejecución, cuándo una parte del componente (i.e.: puerto, aspecto, weaving)

ha sido iniciada/detenida, etc. Para estos caso, el Sensor proporciona servicios adicionales para obtener el estado en ejecución de cada elemento: `GetAspectState(), GetWeavingState(), y GetPortState()`.

Ambos aspectos, Sensor y Actuator, son dependientes de la tecnología, porque realizan tareas que dependen en gran medida de cómo las instancias de los componentes son implementadas. La principal ventaja es que permiten abstraernos de los detalles específicos de la plataforma.

Los principales servicios proporcionados por el Actuator y el Sensor han sido desarrollados en un trabajo previo sobre la tecnología .NET [Per05a]. Un componente PRISMA ha sido desarrollado como una colección de distintos objetos (aspectos, puertos y weavings) que: (i) pueden ser agregados o eliminados dinámicamente; (ii) son altamente independientes entre sí; (iii) interaccionan entre sí mediante el uso de mecanismos asíncronos. Los mecanismos asíncronos son muy importantes porque permiten que las partes internas del componente puedan ser detenidas y reiniciadas después de que los cambios hayan sido realizados, sin que el código en ejecución de las partes llegue a detectar que se ha realizado un cambio.

4.5 Conclusiones

Este capítulo ha descrito un enfoque novedoso para soportar la adaptabilidad en tiempo de ejecución de componentes orientados a aspectos. Sin embargo, la propuesta es suficientemente genérica para ser aplicada en otro tipo de componentes. La propuesta descrita adopta las ventajas derivadas del uso de aspectos en arquitecturas software para beneficiarse de su reutilización y su mantenimiento, que son propiedades fundamentales para el desarrollo de sistemas software complejos.

La evolución dinámica es proporcionada mediante el uso de conceptos de reflexión computacional, ya que permiten describir sistemas auto-adaptables de una forma sencilla y natural. Además, en este trabajo se han abordado también los mecanismos necesarios para modificar tanto los tipos de componentes en ejecución como las instancias en ejecución de dichos tipos. Gracias a la infraestructura de evolución proporcionada, las instancias en ejecución pueden disparar la modificación de tipos de componentes, con lo que sus instancias en ejecución son auto-adaptadas dinámicamente de acuerdo con las modificaciones requeridas. Además, el proceso de auto-adaptación de las instancias de los componentes es posible ya que dicho proceso sólo afecta a aquellas partes del componente que están sufriendo los cambios, gracias a la independencia entre los elementos PRISMA.

Esta infraestructura proporciona los mecanismos necesarios para *planificar* y *ejecutar* adaptaciones. De este modo, tan pronto como al tipo de un componente se le solicita hacer una adaptación, tanto el tipo como cada una de sus instancias *planifican* cuándo podrán ser aplicados los cambios y entonces *ejecutarlos*. Sin embargo, para proporcionar la funcionalidad completa de componentes auto-adaptables, es necesario soportar también la capacidad de dichos componentes para *monitorizar* su estructura y *analizar* cuándo es necesario introducir un cambio [Kep03]. Estos son temas que se abordarán en trabajos futuros.

Otro aspecto que ha quedado por desarrollar es cómo definir restricciones para limitar la evolución de los componentes, de tal forma que no todos los servicios de evolución estén permitidos y que existan ciertas limitaciones, como determinar el número máximo de conexiones, los tipos de aspectos soportados, etc. Por otro lado, la capacidad de evolucionar tipos en tiempo de ejecución también introduce problemas importantes de seguridad, ya que modificaciones no autorizadas del tipo pueden comprometer la seguridad del sistema en ejecución. Por esta razón, otro trabajo futuro es la incorporación de mecanismos de seguridad para asegurar que el tipo de un componente sólo puede ser evolucionado por otros componentes autorizados.

Las ideas presentadas en este capítulo han sido publicadas en un congreso internacional específico del área:

- **Costa C.**, Pérez J., Carsí J.A. *Dynamic Adaptation of Aspect-Oriented Components*. Schmidt, H.W., Crnkovic, I., Heineman, G.T., Stafford, J.A. (eds): 10th International Symposium on Component-Based Software Engineering (CBSE'07). Lecture Notes on Computer Science, vol. 4608, pp. 49-67. Springer, Heidelberg, July 2007. ISSN 0302-9743, ISBN: 978-3-540-73550-2. *Ratio de aceptación: 22% (19 de 86)*

CAPÍTULO V

CONCLUSIONES

5.1 Conclusiones

La relevancia de este trabajo radica en la visión integradora que proporciona del campo. Hasta la fecha, la mayor parte de los trabajos existentes relacionados con arquitecturas de software dinámicas tan sólo se han centrado en proporcionar un grado de dinamismo, bien reconfiguración dinámica en los enfoques procedentes de arquitecturas software, bien evolución de tipos en los enfoques procedentes del desarrollo basado en componentes o del desarrollo orientado a aspectos.

Las principales aportaciones de este trabajo son las siguientes:

- *Integra los dos grados de dinamismo relevantes en el área de las Arquitecturas Software: la Reconfiguración Dinámica y la Evolución Dinámica de Tipos.*

  Esta propuesta permite el modelado y especificación de ambos grados de dinamismo. De esta forma, se sientan las bases para poder construir sistemas autónomos [Kep03] capaces de adaptarse al contexto. Ningún trabajo previo ha combinado ambos grados de dinamismo, sino que se han centrado en sólo uno de ellos.

- *Contempla el dinamismo de las arquitecturas jerárquicas.*

  La mayor parte de trabajos de la literatura se han centrado solamente en soportar el dinamismo de arquitecturas planas, es decir, de un solo nivel de granularidad. En esta propuesta se ha contemplado el dinamismo tanto a nivel arquitectónico (global) como a nivel de componentes (local). De esta forma, los distintos subsistemas de una arquitectura software tienen la capacidad de reconfigurar y/o evolucionar su estructura interna independientemente de la arquitectura.

- *Separación del concern de Evolución del resto de la funcionalidad*

  Las descripciones de cómo debe reconfigurarse el sistema y cómo debe evolucionar se encapsulan en aspectos, por lo que se evita que dicho código se entremezcle con el resto de la funcionalidad del sistema, facilitando por tanto la reutilización y el mantenimiento.

- *Los elementos arquitectónicos son capaces de evolucionar independientemente de la arquitectura que los contiene*

  El trabajo realizado ha preservado en todo momento la autonomía de cada uno de los elementos arquitectónicos, evitando la dependencia respecto a entidades centralizadas y facilitando así que los elementos arquitectónicos puedan reconfigurarse y/o ser evolucionados sin necesidad de la intervención de un elemento externo[1].

5.2 Trabajos futuros

Se pretende que este trabajo sea la base de una tesis que se enmarcará en el ámbito de los sistemas complejos altamente dinámicos y por lo tanto, en la definición y descripción de sistemas capaces de evolucionar dinámicamente.

Los trabajos futuros a realizar son varios:

- Incorporar las pautas de diseño y de especificación de arquitecturas dinámicas en la herramienta ya existente de modelado de arquitecturas software orientadas a aspectos PRISMACASE [Per07], permitiendo la construcción y generación automática de código de arquitecturas capaces de evolucionar dinámicamente.
- Integrar las propuestas de reconfiguración dinámica y de evolución de tipos en un solo modelo, con el objetivo de poder describir arquitecturas reconfigurables y adaptables en tiempo de ejecución.
- Incorporar mecanismos para aumentar el grado de autonomía de las arquitecturas software y poder construir con ello arquitecturas auto-organizadas y auto-adaptables.

[1] Cuando nos referimos a la autonomía en la evolución de los elementos arquitectónicos se hace referencia a que cada elemento evoluciona en paralelo y de forma independiente al resto, no son coordinados por un configurador global. No debe confundirse con la capacidad de que los elementos generen por sí mismos las nuevas configuraciones, lo que queda fuera del ámbito de este trabajo.

5.3 Publicaciones

El trabajo realizado durante los dos primeros años de investigación, y que se ha materializado en esta tesis de máster, ha generado varias publicaciones, tanto nacionales como internacionales.

Es de resaltar la publicación como artículo completo en el CBSE'07 de la propuesta sobre evolución dinámica de tipos presentada en el capítulo 4. Este congreso internacional, con un ratio de aceptación del 22% (19/86), valida la propuesta presentada.

5.3.1 Artículos en revistas internacionales

- Ali N., Pérez J., **Costa C.**, Ramos I., Carsí J.A. *Replicación distribuida en arquitecturas software orientadas a aspectos utilizando ambientes*. IEEE Latin America Transactions, Special Edition JISBD'06 (Jornadas de Ingeniería del Software y Bases de Datos), vol. 5, issue 4, pp. 231-237. IEEE Region 9, July 2007. ISSN 1548-0992.

5.3.2 Artículos en congresos internacionales

- **Costa C.**, Ali N., Pérez J., Carsí J.A., Ramos I. *Dynamic Reconfiguration of Software Architectures Through Aspects*. Oquendo, F. (ed.): First European Conference on Software Architecture (ECSA'07). Lecture Notes on Computer Science, vol. 4758, pp. 279-283. Springer, Heidelberg, September 2007. ISSN 0302-9743, ISBN: 978-3-540-75131-1.

- **Costa C.**, Pérez J., Carsí J.A. *Dynamic Adaptation of Aspect-Oriented Components*. Schmidt, H.W., Crnkovic, I., Heineman, G.T., Stafford, J.A. (eds): 10th International Symposium on Component-Based Software Engineering (CBSE'07). Lecture Notes on Computer Science, vol. 4608, pp. 49-67. Springer, Heidelberg, July 2007. ISSN 0302-9743. *Ratio de aceptación: 22% (19 de 86)*

- Ali N., Pérez J., **Costa C.**, Ramos I., Carsí J.A., *Mobile Ambients in Aspect-Oriented Software Architectures*. K. Sacha (ed.): Software Engineering Techniques: Design for Quality. IFIP Series, vol. 227, pp. 37-48. Springer, Warsaw, Poland, October 2006. ISSN 1571-5736.

- **Costa C.**, Ali N., Millán C., Carsí J.A., *Transparent Mobility of Distributed Objects using .NET*. In proc. of 4th International Conference on .NET Technologies, pp. 11-18. Pilsen, Czech Republic, June 2006. ISBN 80-86943-10-0.

- Pérez J., Ali N., **Costa C.**, Carsí J.A., Ramos I., *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. In proc. of 3rd International conference on .NET Technologies, pp. 97-108. Pilsen, Czech Republic, June 2005. ISBN 80-86943-01-1.

5.3.3 Artículos en congresos nacionales

- Pérez J., **Costa C.**, Carsí J.A., Ramos I. *Verificación de Modelos Arquitectónicos Orientados a Aspectos*. En actas de XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07), junto a II Congreso Español de Informática (CEDI'07). Zaragoza, 11-14 septiembre 2007. (aceptado, pendiente de publicar)

- Pérez J., **Costa C.**, Carsí J.A., Ramos I. *PRISMA CASE.* En actas de XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07), junto a II Congreso Español de Informática (CEDI'07). Zaragoza, 11-14 septiembre 2007. (aceptado, pendiente de publicar) (Demostración)

- Cabello M.E., **Costa C.**, Ramos I. *Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico*. XIX Congreso Nacional y V Congreso Internacional de Informática y Computación. Chiapas, México, octubre 2006. ISBN: 970-31-0751-6.

- Ali N., Pérez J., **Costa C.**, Ramos I., Carsí J.A. *Replicación distribuida en arquitecturas software orientadas a aspectos utilizando ambientes.* En actas de XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06), pp. 379-388. Sitges, octubre 2006. ISBN 84-95999-99-4

- **Costa C.**, Pérez J., Ali N., Carsí J.A., Ramos I. *PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos.* En actas de X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05), pp. 27-34. I Congreso Español de Informática (CEDI'05). Granada, septiembre 2005. ISBN 84-9732-434-X

5.3.4 Artículos en workshops nacionales

- **Costa C.**, Pérez J., Carsí J.A. *Hacia la construcción de arquitecturas software dinámicas*. En actas de V jornadas de trabajo de DYNAMICA. Valencia, 23-24 noviembre, 2006. ISBN: 84-690-2623-2.

- **Costa C.**, Pérez J., Carsí J.A. *Hacia la reconfiguración dinámica de arquitecturas software orientadas a aspectos*. En actas del IV Taller de Desarrollo de Software Orientado a Aspectos (DSOA'06), junto a XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06). Servicio de

publicaciones de la Universidad de Extremadura, Informe técnico TR 24/06, pp. 35-40, septiembre 2006. ISBN: 84-7723-670-4

- Ali N., Pérez J., **Costa C.**, Carsí J.A., Ramos I. *Implementation of the PRISMA Model in the .Net Platform*. En Actas II jornadas de trabajo de DYNAMICA (DYNamic and Aspect-Oriented Modelling for Integrated Component-based Architectures), junto a JISBD'04. Malaga, 11 noviembre 2004.

5.3.5 Informes técnicos

- **C. Costa**, J. Pérez, J.A. Carsí. *Estudio e implementación de un modelo de arquitecturas orientado a aspectos y basado en componentes sobre tecnología .NET*. Informe técnico DSIC-II/11/05. Universidad Politécnica de Valencia, abril 2005.

BIBLIOGRAFÍA

[Ali03] Ali N.H., Silva J., Jaen J., Ramos I., Carsi J.A., Perez J. *Mobility and Replicability Patterns in Aspect-oriented Component-Based Software Architectures.* 15th IASTED, Parallel and Distributed Systems, Acta Press, pp. 820-826. Marina del Rey, C.A., USA, Noviembre 2003.

[Ali05] Ali N., Ramos I., Carsí J.A., *A Conceptual Model for Distributed Aspect-Oriented Software Architectures*. Int. Symp. on Information Technology: Coding and Computing (ITCC), IEEE Computer Society, Vol. 2, Las Vegas, Nevada, USA, April, 2005.

[All98] Allen, R., Douence, R., Garlan, D.: *Specifying and Analyzing Dynamic Software Architectures*. 1st Int. Conf. on Fundamental Approaches to Software Engineering (FASE'98), held as part of the Joint European Conference on Theory and Practice of Software (ETAPS'98), pp. 21-37. Lisbon, Portugal, 1998.

[Am04] Ambler, S., *Agile Model-driven Development with UML 2.0*, Cambridge University Press, 2004.

[And03] Andrade, L.F., Fiadeiro J. L., *Architecture Based Evolution of Software Systems*. Formal Methods for Software Architectures, Lecture Notes in Computer Science, Springer Verlang, Eds. Marco Bernardo and Paola Inverardi, LNCS 2804, September 2003.

[Aosd] *Aspect-oriented software development.* En: http://aosd.net

[AsJ03] The AspectJ Team. *AspectJ Programming Guide*, 2003.

[AsJ07] AspectJ Project, en http://eclipse.org/aspectj/

[Bal85] Balzer R., *A 15 Year Perspective on Automatic Programming*. IEEE Transactions on Software Engineering, vol.11, num.11, págs. 1257-1268, Noviembre 1985.

[Bal05] Balasubramaniam, D., Morrison, R., Kirby, G. et al.: *A Software Architecture Approach for Structuring Autonomic Systems*. Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005), pp. 1-7. St. Louis, Missouri, USA, 2005.

[Bey05] Beydeda, S., Book, M., Gruhn V., *Model-Driven Software Development*, Springer, 2005.

[Bra04] Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: *A Survey of Self-Management in Dynamic Software Architecture Specifications*. 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04). Newport Beach, California, 2004.

[Bra07] Braberman, V.: *The SetPoint! project*, http://setpoint.codehaus.org, 2007.

[Buc05] Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: *Towards a taxonomy of software change*. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 17, Issue 5, pp. 309-332. Wiley InterScience, 2005.

[Cab05] Cabedo R., Pérez J., Ramos I.: *The application of the PRISMA Architecture Description Language to an Industrial Robotic System*. Technical Report, DSIC II/11/05, Polytechnic University of Valencia, September, 2005. (In Spanish)

[Can01] Canal C, Pimentel E., Troya J.M., *Compatibility and Inheritance in Software Architectures*. Science of Computer Programming, vol. 41, no. 2. 2001.

[Che05] Cheng, S., Garlan, D., Schmerl, B.: *Making Self-Adaptation an Engineering Reality*. In: O. Babaoglu, M. Jelasity, A. Montresor et al. (eds.): Self-Star Properties in Complex Information Systems. LNCS, vol. 3460, pp. 158-173. Springer, Bertinoro, Italy, 2005.

[Cos05] Costa, C., Pérez, J., Ali, N., Carsí, J.A., Ramos, I.: *PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures*, X Conference on Software Engineering and Databases (JISBD), pp. 27-34. Granada, September, 2005. (In Spanish)

[Cue01] Cuesta, C.E., Fuente, P.d.l., Barrio-Solárzano, M.: *Dynamic Coordination Architecture through the use of Reflection*. 2001 ACM Symposium on Applied Computing, pp. 134-140. Las Vegas, Nevada, United States, 2001.

[Cue02] Cuesta, C.E. *Arquitectura Dinámica de Software Basada en Reflexión*. Tesis Doctoral, Departamento de Ciencias de la Computación, Universidad de Valladolid, 2002.

[Das02] Dashofy, E.M., van der Hoek, A., Taylor, R.N.: *Towards Architecture-Based Self-Healing Systems*. In proc. of First Workshop on Self-Healing Systems (WOSS'02), pp. 21-26. Charleston, South Carolina (november 18-19 2002.

[Dou05] Douence R, Le Botlan D., *Report Towards a Taxonomy of AOP Semantics*. AOSDEurope Deliverable D11, AOSD-Europe-INRIA-1, INRIA, CNRS, 7 July 2005, pp. 1-13.

[DSo98] D'Souza, D. F., & Wills, A. C.: *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[End92] Endler, M., & Wei, J.: *Programming Generic Dynamic Reconfigurations for Distributed Applications*. First International Workshop on Configurable Distributed Systems, pp. 68-79. London, UK, Mar 1992.

[Gar93] Garlan, D., Shaw M., *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering, Vol. I. Eds. V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jersey, 1993.

[Gar95] Garlan, D., Perry D., *Introduction to the Special Issue on Software Architecture*. IEEE Transactions on Software Engineering, vol. 21 no. 4, April 1995.

[Gar00] Garlan D., Monroe R. T., Wile D., *Acme: Architectural Description of Component-Based Systems*. Foundations of Component-Based Systems,

Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, pp. 47-68, 2000.

[Gar01] Garlan, D., *Software Architecture*, Wiley Encyclopedia of Software Engineering, J. Marciniak (Ed.), John Wiley & Sons, 2001.

[Geo02] Georgiadis, I., Magee, J., Kramer, J.: *Self-Organising Software Architectures for Distributed Systems*. First Workshop on Self-Healing Systems (WOSS'02), pp. 33-38. Charleston, South Carolina, November 18-19, 2002.

[Gus04] Gustavsson, J., Staijen, T., Assmann, U.: *Runtime Evolution as an Aspect*. Proceedings of First International Workshop on Foundations of Unanticipated Software Evolution. Barcelona, Spain, 2004.

[Har02] Harrison, W. H., Ossher, H. L., Tarr, P. L.: *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. Technical Report RC22685, Thomas J. Watson Research Center, IBM, 2002.

[IEE00] *IEEE Recommended Practices for Architectural Description of Software-Intensive Systems*. IEEE Std 1471-2000, Software Engineering Standards Committee of the IEEE Computer Society, 21 September 2000.

[Kep03] Kephart, J. O., & Chess, D. M.: *The Vision of Autonomic Computing*. In: Computer, Vol. 36, No. 1, pp. 41-50. IEEE Computer, 2003.

[Kic97] Kiczales, G., et al., *Aspect-Oriented Programming*. In European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland, 1997, Springer-Verlag, pp.220-242.

[Kic01] Kiczales G., Hilsdale E., Huguin J., Kersten M., Palm J., Griswold W.G., *An Overview of AspectJ*. The 15th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol.2072, Budapest, Hungary, June 18-22, 2001.

[Kra85] Kramer, J., Magee, J.: *Dynamic Configuration for Distributed Systems*. IEEE Transactions on Software Engineering, Vol. 11, Issue 4, pp. 424-436, 1985.

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O., *OASIS 3.0: "Un enfoque formal para el modelado conceptual orientado a objeto"*. Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721- 663-0, 1998.

[Mae87] Maes, P.: *Concepts and Experiments in Computational Reflection*. In: SIGPLAN Notices, Vol. 22, No. 12, pp. 147-155. ACM Press, New York, NY, USA, 1987.

[Mag95] Magee J.N., Dulay N., Eisenbach S., Kramer J., *Specifying Distributed Software Architectures*. Fifth European Software Engineering Conference (ESEC), Barcelona, Spain, September, 1995.

[Mey03] Meyer B., *The Grand Challenge of Trusted Components*. International Conference on Software Engineering (ICSE), IEEE Computer Press, Portland, Oregon, May 2003.

[Mil91] Milner, R., *π-Cálculo Poliádico: A tutorial*, 1991.

[Mor04] Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K. et al.: Support for Evolving Software Architectures in the ArchWare ADL. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04), pp. 69-78. Oslo, Norway, June 12-15, 2004.

[Nie95] Nierstrasz O., Meijler T.D., *Research Directions of Software Composition*. ACM Computing Surveys (CSUR), Vol. 27, no. 2, June, 1995.

[OMG03] Object Management Group (OMG): *Model Driven Architecture Guide*, http://www.omg.org/docs/omg/03-06-01.pdf, 2003.

[OMG02] Object Management Group (OMG): *Meta-Object Facility (MOF) 1.4 Specification*. Technical Report formal/2002-04-03. http://www.omg.org/technology/documents/formal/mof.htm, 2002.

[Oqu04a] Oquendo F., *π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures*. ACM Software Engineering Notes, Vol. 29, No. 3, May, 2004.

[Oqu04b] Oquendo F., Warboys B., Morrison R., Dindeleux R, Gallo F., Garavel H., Occhipinti C., *ArchWARE: Architecting Evolvable Software*. 1st European Workshop in Software Achitecture (EWSA'04), Lecture Notes in Computer Science LNCS 3047, St Andrews, UK, pp. 257-271. Springer, 2004.

[Ore99] Oreizy, P., Gorlick, M., Taylor, R.N., Heimbigner, D., Johnson, G. et al.: *An Architecture-Based Approach to Self-Adaptive Software*. IEEE Intelligent Systems, 14, pp. 54-62, 1999.

[Per92] Perry, D., Wolf A., *Foundations for the Study of Software Architecture*. ACM Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.

[Per05a] Pérez, J., Ali, N., Costa, C., Carsí, J.A, Ramos, I.: *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. 3rd Int. Conf. on .NET Technologies, pp. 97-108. Pilsen, Czech Republic, June 2005.

[Per05b] Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: *Dynamic Evolution in Aspect-Oriented Architectural Models*. In: R. Morrison, & F. Oquendo (eds.): 2nd European Workshop on Software Architecture (EWSA'05). LNCS, Vol. 3527, pp. 59-76. Springer, Pisa, Italy, 2005.

[Per06] Pérez, J. *PRISMA: Aspect-Oriented Software Architectures*. PhD Thesis, Department of Information Systems and Computation, Polytechnic University of Valencia, Spain, December 2006.

[Per06a] Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: *Designing Software Architectures with an Aspect-Oriented Architecture Description Language*. 9th Int. Symp. on Component-Based Software Engineering (CBSE06). Lecture Notes in Computer Science, Vol. 4063, pp. 123-138. Springer, 2006.

[Per07] Pérez J., Costa C., Carsí J.A., Ramos I. PRISMA CASE. XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). Zaragoza, 11-14 septiembre 2007. (Demostración)

[Pin05] Pinto, M., Fuentes, L., Troya, J. M.: *A Dynamic Component and Aspect-Oriented Platform*. In: The Computer Journal, Vol. 48, No. 4, pp. 401-420. Oxford University Press, 2005.

[Raj03] Rajan, H., & Sullivan, K.: *Eos: Instance-Level Aspects for Integrated System Design*. 9th European Software Engineering Conf. held jointly with 11th ACM SIGSOFT Intern. Symp. on Foundations of Software Engineering. Helsinki, Finland, pp. 297-306, September 2003.

[Ras03] Rasche, A., Polze, A.: *Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET*. Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03), Hakodate, Hokkaido, Japan, 14-16 May 2003.

[Sch06] Schmidt D.C., *Model-Driven Engineering*, IEEE computer Society, 2006.

[Sha94] Shaw M., *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status*. Workshop on Studies of Software Design, January, 1994.

[Sch02] Schult W., Polze A., *Aspect-Oriented Programming with C# and .NET*. 5th IEEE Intern. Symp. on Object-Oriented Real-time Distributed Computing, IEEE Computer Society Press, pp. 241-248, Washington, DC, 2002.

[Sch03] Schult, W., & Polze, A.: *Speed Vs. Memory Usage-an Approach to Deal with Contrary Aspects*. 2nd Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Int. Conf. on Aspect-Oriented Software Development (AOSD). Boston, Massachusetts, 2003.

[Smi82] Smith, B.C. *Reflections and Semantics in a Procedural Language*. PhD Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1982.

[Suv03] Suvée D., Vanderperren W., Jonckers V., *JAsCo: an Aspect-Oriented Approach Tailored for Component-Based Software Development*. 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 21-29, Boston, Massachusetts, March, 2003.

[Szy98] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[Yan02] Yang, Z., Cheng, B.H.C., Stirewalt, R.E.K. et al.: *An Aspect-Oriented Approach to Dynamic Adaptation*. In proc. of First Workshop on Self-Healing Systems (WOSS'02). Charleston, South Carolina, pp. 85-92, 2002.

[Wan06] Wang, Q., Shen, J., Wang, X., Mei, H.: *A component-based approach to online software evolution*. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 18, pp. 181-205. Wiley InterScience, 2006.