Soporte a la Evolución Dinámica de Tipos Arquitectónicos

Cristóbal Costa-Soria[1], Jennifer Pérez[2], Jose A. Carsí[1]

[1] ISSI, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022 Valencia
{ccosta, pcarsi}@dsic.upv.es

[2] E.U. Informática, Universidad Politécnica de Madrid,
Ctra. Valencia km 7, 28051 Madrid
jenifer.perez@eui.upm.es

Resumen. Los sistemas software con una fuerte naturaleza dinámica suponen un reto para la ingeniería del software. Este tipo de sistemas requieren de mecanismos que les permitan modificar tanto estructura como comportamiento en tiempo de ejecución, para adaptarse a las distintas situaciones que puedan presentarse. El área de arquitecturas software, que permite describir la estructura de los sistemas complejos a un alto nivel de abstracción, proporciona dos grados de dinamismo para la construcción de sistemas dinámicos, dependiendo de si lo que evoluciona es la configuración de la arquitectura o los tipos que componen dicha arquitectura. El primer tipo de evolución, denominado reconfiguración dinámica, permite a una arquitectura software cambiar su configuración en tiempo de ejecución, creando/destruyendo instancias de elementos arquitectónicos y/o las conexiones entre ellas. El segundo tipo de evolución, que denominamos evolución dinámica de tipos arquitectónicos, permite cambiar completamente la especificación arquitectónica de un sistema dinámicamente, bien introduciendo nuevos tipos arquitectónicos, modificando tipos e instancias en ejecución, o bien introduciendo nuevas conexiones. Este artículo presenta cómo soportar este último grado de dinamismo desde un punto de vista independiente de plataforma. Para ello, se han identificado los diferentes asuntos de interés implicados en el proceso y se han encapsulado en aspectos.

Palabras Clave: runtime evolution, types evolution at runtime, reflection, self-star systems, software architectures, CBSD, AOSD.

1. Introducción

En la actualidad, los sistemas software cada vez son más complejos, lo que conlleva que con frecuencia dichos sistemas deban someterse a revisiones posteriores para incorporar funcionalidades no previstas inicialmente o corregir errores. Sin embargo, la propia naturaleza de determinados sistemas, como aquellos que desempeñan misiones críticas y por tanto deben ejecutarse continuamente e ininterrumpidamente, hace inviable su detención para su adaptación o evolución. En este tipo de sistemas es donde surge la necesidad de incorporar capacidades que les permitan ser evolucionados en tiempo de ejecución, sin ser detenidos.

Las arquitecturas software [17] permiten describir sistemas complejos en términos de elementos arquitectónicos (componentes y conectores) y las interacciones entre sí (conexiones). Las propuestas para describir y especificar arquitecturas software, con el objetivo de dotar de mayor flexibilidad a los sistemas construidos, ofrecen algún tipo de soporte para la evolución dinámica. Ésta puede ser de dos tipos, dependiendo de si lo que se modifica es la configuración de la arquitectura o los tipos que componen dicha arquitectura. El primer tipo de evolución, la reconfiguración dinámica (también llamada dinamismo estructural [9]), permite a una arquitectura

software cambiar su configuración en tiempo de ejecución, creando o destruyendo dinámicamente instancias de elementos arquitectónicos y sus enlaces. El segundo tipo de evolución, la evolución dinámica de tipos arquitectónicos (o también llamado dinamismo arquitectónico [9]), permite a una arquitectura software cambiar completamente su tipo (i.e., su especificación) en tiempo de ejecución, introduciendo nuevos tipos de elementos arquitectónicos y conexiones, eliminando tipos existentes, o cambiando la forma en que los tipos interaccionan, cambiando de este modo tanto la composición como el comportamiento de la arquitectura software.

En este artículo se presenta una propuesta para soportar este último grado de dinamismo, la evolución dinámica de tipos arquitectónicos, refiriéndonos al proceso de evolucionar un tipo que ya se encuentra instanciado en un determinado sistema software, y la migración o evolución, en tiempo de ejecución, de todas sus antiguas instancias a la estructura y comportamiento definida por el nuevo tipo. Esta propuesta se presenta desde un punto de vista independiente de la plataforma. Para ello, se han identificado los diferentes asuntos de interés implicados en el proceso de evolución dinámica y se han encapsulado en aspectos. Con el objetivo de ilustrar nuestra propuesta, se describirá cómo se ha aplicado para un ADL concreto, aunque es extrapolable a cualquier otro. En particular, se mostrará cómo proporcionar evolución dinámica a tipos arquitectónicos PRISMA.

En este trabajo sólo se ha considerado la evolución interna del tipo, es decir, no se ha tenido en cuenta cómo dicha evolución puede afectar a las interacciones del tipo evolucionado con el resto de tipos del sistema. Así ocurre cuando las interfaces públicas del tipo son modificadas, con lo que para mantener la consistencia del sistema, deberán adaptarse las conexiones existentes para que el resto de elementos arquitectónicos puedan comunicarse con las instancias del nuevo tipo. Esta problemática ya ha sido abordada por otros autores, como en el trabajo de Cámara et al. [5], mediante la introducción dinámica de adaptadores que actúan de intermediarios entre las instancias existentes y las nuevas instancias, por lo que no ha sido considerada en este trabajo.

La estructura del artículo es la siguiente: en el apartado 2 se introduce el modelo PRISMA, modelo sobre el cual se ha aplicado esta propuesta. En el apartado 3 se describe nuestra propuesta, describiendo tanto los mecanismos utilizados a nivel de tipos como los utilizados a nivel de instancias. En el apartado 4 se discute la propuesta, comparándola con otros trabajos relacionados y, finalmente, en el apartado 5 se presentan las conclusiones y los trabajos futuros.

2. PRISMA

PRISMA es un enfoque para el desarrollo de arquitecturas software orientadas a aspectos independientes de tecnología [18, 19], concebido en el grupo de investigación ISSI. El modelo PRISMA tiene tres tipos de elementos arquitectónicos: componentes, conectores y sistemas. Externamente, los tres tipos son idénticos: encapsulan su funcionalidad como cajas negras y publican un conjunto de servicios que ofrecen a otros elementos arquitectónicos (ver Figura 1-A). Estos servicios, agrupados en interfaces, son publicados a través de los puertos, que son los puntos de interacción entre elementos arquitectónicos. Sin embargo, internamente se diferencian en que los componentes y conectores son elementos arquitectónicos simples, mientras que los sistemas son elementos arquitectónicos complejos.

Un elemento arquitectónico simple puede verse como un prisma, donde cada lado del prisma es un aspecto importado por el elemento arquitectónico (ver Figura 1-B). Un aspecto representa un comportamiento específico de un asunto (*concern*) -seguridad, coordinación, distribución, etc.- que se encuentra disperso por la arquitectura software. PRISMA se caracteriza por seguir un enfoque simétrico [12], en el cual los aspectos no están restringidos a especificar únicamente requerimientos no funcionales, sino que también pueden describir la propia funcionalidad del sistema. De esta forma, los elementos arquitectónicos simples son representados como un conjunto de aspectos. Los aspectos son sincronizados entre sí a través de relaciones de *weaving*. Un

weaving indica que la ejecución del servicio de un aspecto puede activar la ejecución de servicios en otros aspectos.

Por su parte, la vista interna de un elemento arquitectónico complejo (sistema) incluye un conjunto de elementos arquitectónicos (componentes, conectores y otros sistemas) y las conexiones entre ellos (ver Figura 1-C). Estas conexiones pueden ser de dos tipos: *attachments*, que modelan las comunicaciones realizadas dentro del sistema (entre elementos arquitectónicos); y *bindings*, que modelan las comunicaciones desde o hacia el exterior del sistema.

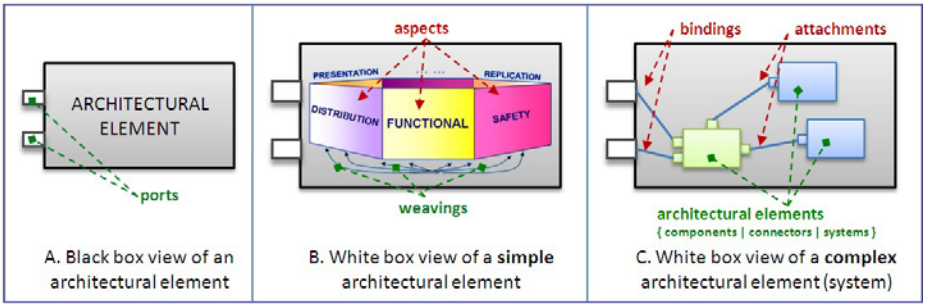

Fig. 1. Vistas de un elemento arquitectónico PRISMA

3. Evolución Dinámica de Tipos Arquitectónicos

De este modo, en PRISMA tendremos dos clases de tipos arquitectónicos que pueden ser evolucionados: los elementos arquitectónicos simples y los elementos arquitectónicos complejos (sistemas). Por motivos de espacio, sólo describiremos aquí cómo se evolucionan los elementos arquitectónicos complejos, aunque puede fácilmente extrapolarse a la evolución de elementos arquitectónicos simples. Puede encontrarse más información en un trabajo previo [8].

El objetivo perseguido es el de dotar, a cada tipo arquitectónico del sistema software, de las capacidades necesarias para que sea capaz de cambiar dinámicamente tanto su especificación (el tipo), como sus instancias de forma autónoma. De este modo, cada tipo arquitectónico podrá evolucionar independientemente del resto de tipos, favoreciendo así la construcción de sistemas distribuidos, heterogéneos y autónomos. Para ello, cada tipo arquitectónico debe estar dotado de una infraestructura que le permita evolucionar en tiempo de ejecución. En nuestra propuesta, esta infraestructura está integrada como parte del propio tipo y como parte de cada una de las instancias. Existen dos razones por las cuales se ha distribuido la infraestructura de evolución entre el tipo arquitectónico y sus instancias, y están relacionadas con que, al igual que el tipo, cada instancia debe ser capaz de evolucionar de forma autónoma. La primera razón es que cada instancia es la única que puede determinar en qué momento está lista para evolucionar. En tiempo de ejecución, cada instancia tiene un estado y un conjunto de transacciones distintas al resto de instancias del mismo tipo, por lo que el instante en que cada instancia estará lista para su evolución será distinto para cada una de ellas. La segunda razón es que se ha seguido un enfoque de evolución incremental, en el cual cada instancia se descompone en las partes estructurales que la forman y mediante una serie de operaciones de evolución atómicas, son modificadas. Estas partes estructurales sólo son accesibles por cada instancia en ejecución, por lo que sólo podrán ser modificadas por mecanismos de evolución que sean ejecutados por dichas instancias.

La infraestructura de evolución está distribuida del siguiente modo. El tipo dispone de los mecanismos necesarios para: (1) devolver o generar su especificación y proporcionar los servicios para modificar dicha especificación, (2) actualizar su especificación con los cambios introducidos, de forma que nuevas instancias se creen de acuerdo al nuevo tipo, y (3) controlar el proceso de migración (o evolución) de cada una de sus instancias. Por su parte, cada instancia dispone de una serie de mecanismos para: (4) alcanzar un estado quiescente [13], en el cual las transacciones en

ejecución finalizan de forma consistente, (5) modificar dinámicamente su estructura (en memoria) de acuerdo a los cambios indicados por el tipo, (6) si es posible, migrar el estado antiguo a las nuevas estructuras introducidas por el nuevo tipo.

3.1 Evolución del tipo

Nuestra propuesta se caracteriza por dotar a cada tipo arquitectónico de presencia real en la aplicación software, a través de una entidad que representa al tipo arquitectónico y que está en ejecución junto al resto de las instancias de dicho tipo. Esta entidad, a la que nos referiremos como *M*, puede ser vista desde dos puntos de vista diferentes. Por una parte, se comporta como clase, en el sentido de que proporciona servicios para la creación y destrucción de instancias del tipo arquitectónico que representa, así como también se encarga de mantener la población de instancias que han sido creadas. Por otra parte, esta entidad se comporta como objeto, pues tiene un estado y unos servicios que lo modifican. Sin embargo, dicho estado es una descripción editable del tipo que representa y, por tanto, los servicios que modifican dicho estado son en realidad servicios de evolución, pues al cambiar la descripción editable del tipo se está realmente modificando al tipo y sus instancias. Usando los conceptos del campo de la reflexión computacional [14], esta entidad M es en realidad una meta-instancia (o meta-componente), pues contiene la reificación del tipo (esto es, una descripción editable), que está causalmente conectada al tipo: todos los cambios que se hagan en dicha reificación serán reflejados en el tipo y en sus respectivas instancias.

La estructura interna de una meta-instancia se compone de cuatro módulos o áreas funcionales: (1) Builder, responsable de crear y destruir instancias del tipo arquitectónico; (2) TypeDescription, que encapsula la reificación del tipo arquitectónico y la población de instancias; (3) EvolutionPlanning, que proporciona los servicios de evolución; y (4) EvolutionMonitoring, que supervisa el proceso de migración de instancias desde el tipo antiguo al nuevo tipo.

Las meta-instancias han sido incorporadas en PRISMA utilizando los conceptos del propio modelo PRISMA: una meta-instancia es un componente simple PRISMA, formado por un conjunto de aspectos. Dichos aspectos son cada una de las áreas funcionales identificadas arriba, pues identifican un asunto de intéres distinto del proceso de evolución, y que es compartido por cada meta-instancia que incorpore mecanismos para la evolución de tipos (a excepción del módulo Builder, que es diferente para cada tipo). Por compartido nos referimos a que el tipo de dichos aspectos es común a todas las meta-instancias, y tan sólo se diferencian en el estado que adoptan cuando son instanciados en una meta-instancia particular. Las relaciones entre cada uno de estos aspectos se han definido mediante weavings, aunque por razones de espacio no se detallarán aquí.

El aspecto Builder proporciona los servicios para la creación y destrucción de instancias del tipo que representa la meta-instancia. Sus servicios son publicados a través de un puerto de la meta-instancia, que es bloqueado mientras dura el proceso de evolución, ya que dichos servicios también deben modificarse para que futuras instanciaciones sean acordes al nuevo tipo.

El aspecto TypeDescription contiene el estado de la meta-instancia, formado por la población de instancias y la descripción del tipo. Por un lado, la población de instancias es actualizada cada vez que se crea una instancia nueva, añadiendo una referencia a ella, o cada vez que se destruye una instancia, borrando la referencia correspondiente. Por otro lado, la descripción del tipo indica qué elementos constituyen el tipo arquitectónico y las relaciones entre dichos elementos. Por ejemplo, en el caso de especificaciones de sistemas PRISMA, esta estructura de datos almacena los tipos de componentes que componen el sistema, los tipos de conexiones que pueden establecerse entre los componentes, y las cardinalidades correspondientes. Este aspecto codifica las relaciones entre conceptos independientes de plataforma (i.e. el metamodelo PRISMA) y los conceptos dependientes de tecnología (i.e. la implementación del modelo de componentes orientado a aspectos en .NET). Por tanto, contiene también los patrones de generación automática de código que deben utilizarse para regenerar el tipo.

El aspecto EvolutionPlanning proporciona los servicios de evolución dinámica de tipos. Tan sólo ofrece dos servicios al exterior: el servicio Reify, que devuelve un objeto de tipo

<*Type*>*Spec*, y el servicio Reflect, que requiere como parámetro de entrada un objeto de tipo <*Type*>*Spec*. <Type>Spec es una forma genérica de nombrar al tipo del objeto, ya que su tipo dependerá de cuál sea el meta-tipo del tipo representado por la meta-instancia. Por ejemplo, sea el tipo S un sistema PRISMA (esto es, un componente complejo), y cuyo meta-tipo es System, que describe la estructura y características de los elementos arquitectónicos complejos en PRISMA. Siendo M_S la meta-instancia que representa al tipo S en tiempo de ejecución, el objeto devuelto por M_S.Reify será del tipo *SystemSpec.*

Un objeto de tipo <*Type*>*Spec* tiene como estado (privado) la descripción de un tipo y proporciona un conjunto de servicios que permiten consultar y cambiar dicho estado (los servicios de evolución). Los servicios de evolución sólo permiten modificar la especificación del tipo de acuerdo a su meta-tipo respectivo, que es <*Type*>. Siguiendo con el ejemplo PRISMA, el objeto de tipo *SystemSpec* devuelto por M_S.Reify contendrá como estado la descripción del tipo S, y como servicios de evolución ofrecerá aquellos definidos en el metamodelo de PRISMA [21]: *addComponent*, *addConnector*, *addAttachment*, *addPort*, *removeComponent*, etc. De este modo, el actor del proceso de evolución -que puede ser tanto externo al sistema (un actor humano) como interno al sistema (otro elemento arquitectónico)-: (1) mediante el servicio Reify() obtendrá la reificación editable del tipo, (2) modificará la especificación del tipo a través de los servicios de evolución proporcionados por el objeto <Type>Spec devuelto, y (3) devolverá dicho objeto a través del servicio Reflect(), con lo que se iniciará el proceso de evolución del tipo.

El aspecto EvolutionPlanning coordina el proceso de evolución. Cuando el servicio Reify es invocado, a partir de la descripción del tipo almacenada en el módulo TypeDescription, construye el objeto <Type>Spec (que dependerá del meta-tipo del tipo almacenado en TypeDescription), y lo devuelve. Sin embargo, el proceso de evolución no empieza hasta que el servicio Reflect no es invocado. El proceso de evolución consta de varias etapas, que son llevadas a cabo de forma distribuida: es iniciado por EvolutionPlanning, después es realizado localmente por cada instancia, y es supervisado por el aspecto EvolutionMonitoring.

Las tareas de evolución llevadas a cabo por EvolutionPlanning son las siguientes. En primer lugar se bloquea el aspecto Builder, para evitar la creación/destrucción de instancias mientras se está actualizando el tipo. Esto se realiza deteniendo el puerto de la meta-instancia que exporta los servicios del aspecto Builder. En segundo lugar, se actualiza la descripción del tipo almacenada en el aspecto TypeDescription, utilizando la información del objeto <Type>Spec proporcionado al invocar el servicio Reflect. En tercer lugar, el aspecto Builder es regenerado completamente de acuerdo con la descripción del tipo del aspecto TypeDescription, mediante técnicas de generación de código. Finalizado este paso, el aspecto Builder ya está capacitado para crear instancias del nuevo tipo, por lo que es desbloqueado y se permite la creación de nuevas instancias. Por último, se recorre la población de instancias (almacenadas en TypeDescription) y para cada una de ellas se invoca el servicio ReflectToInstance. Dicho servicio inicia el proceso de evolución de cada instancia a nivel local, adaptando su estructura interna a la del nuevo tipo. Finalmente, se pasa el control al aspecto EvolutionMonitoring, con lo que el aspecto EvolutionPlanner podrá admitir nuevas evoluciones de tipos.

El aspecto EvolutionMonitoring define las políticas de migración de instancias que deben llevarse a cabo en el proceso de evolución de tipos: si sólo las nuevas instancias deben crearse de acuerdo al nuevo tipo, o por el contrario, deben evolucionarse todas las instancias al nuevo tipo. En este último caso hay que definir el tiempo que tienen las instancias para evolucionar y las medidas correctoras para el caso en que dichas instancias no hayan evolucionado (como forzar la evolución obligando a perder el estado actual, posponer la evolución otro intervalo adicional, o cancelar la evolución de dicha instancia).

De esta forma, cualquier elemento arquitectónico podrá evolucionar un tipo arquitectónico sin más que conectarse al puerto de evolución de la meta-instancia deseada, solicitar la reificación del tipo. Acto seguido, mediante los servicios de evolución proporcionados, podrá modificar dicha reificación adecuadamente y al reflejarla, desencadenar el proceso de evolución tanto del tipo como de sus respectivas instancias en ejecución.

3.2 Evolución de las instancias

La mayoría de los enfoques para la evolución dinámica de tipos realizan la evolución de instancias a través de la migración del estado [22]: se crea una instancia del nuevo tipo y se le transfiere el estado de la instancia antigua. Para ello, el nuevo tipo debe proporcionar funciones para transformar las estructuras de datos del tipo antiguo a las estructuras de datos del nuevo tipo. Esto requiere que la meta-instancia (o el tipo) tome parte activa en el proceso de evolución de sus instancias.

Sin embargo, esto puede optimizarse si se conoce cuál es la especificación del tipo que está siendo evolucionado: el tipo puede descomponerse en entidades más pequeñas y las relaciones entre ellas, lo que permite aislar las entidades que van a ser evolucionadas mediante el bloqueo temporal de sus relaciones con el resto de entidades. Esto se ve más claro con los tipos arquitectónicos: un sistema PRISMA está formado por varias entidades (elementos arquitectónicos y puertos) y relaciones entre ellas (attachments y bindings). El proceso de evolución consiste entonces en dotar a cada instancia de mecanismos para aislar las partes estructurales que la constituyen (es decir, las entidades y relaciones que constituyen el tipo) y que van a sufrir cambios, mecanismos para que puedan reemplazarlas y para que sean capaces de volver a ensamblarlas de nuevo con el resto de partes de la instancia. De esta forma, pueden reemplazarse solamente las entidades a modificar mientras el resto de entidades siguen en ejecución. La ventaja de descomponer así las instancias, frente a las propuestas en las que se migra el estado las instancias, es más notable cuando lo que se evolucionan son tipos constituidos por entidades que se ejecutan concurrentemente y son altamente independientes entre sí, como es el caso de las arquitecturas software y el caso de los componentes integrados por aspectos, como en el enfoque PRISMA.

Los mecanismos para evolucionar las instancias son proporcionados por tres módulos o áreas funcionales: InstanceEvolutionPlanning, EvolutionSensor y EvolutionEffector. Estos módulos identifican distintos asuntos de interés del proceso de evolución de instancias, por lo que para su implementación en PRISMA también han sido encapsulados en aspectos.

El aspecto EvolutionSensor proporciona servicios para obtener las referencias (en memoria) a cada una de las partes estructurales que componen la instancia (y poder así manipularlas), así como servicios para monitorizar el *status* de cada una de dichas partes estructurales. El *status* indica si una parte estructural está lista para ser evolucionada: no tiene transacciones pendientes en ejecución que puedan alterar su estado, es decir, es quiescente [13] o tranquila [23].

El aspecto EvolutionEffector proporciona los servicios que realmente efectúan la modificación de la estructura de la instancia, en términos del meta-tipo de la instancia. Por ejemplo, en sistemas PRISMA, el EvolutionEffector proporciona los servicios *addComponent*, *addConnector*, *addAttachment*, *addPort*, *removeComponent*, etc. Es decir, ofrece los mismos servicios que proporcionaba el objeto *SystemSpec*, sólo que mientras este último sólo actualizaba una especificación (datos), el EvolutionEffector está aplicando los cambios a estructuras en memoria (código en ejecución). Además, el aspecto EvolutionEffector también proporciona los servicios para detener o reiniciar a cada una de las partes estructurales, es decir, que alcancen el estado quiescente o lo abandonen, respectivamente. Estos servicios en PRISMA son: *StartComponent*, *StartAttachment*, *StopComponent*, *StopAttachment*, etc.

Sin embargo, el aspecto EvolutionEffector no tiene en cuenta las dependencias entre las partes estructurales al aplicar los cambios, ni si estaban listas para ser evolucionadas. Esto es realizado por el aspecto InstanceEvolutionPlanning, que coordina el proceso de evolución a nivel de instancias. El aspecto InstanceEvolutionPlanning recibe el conjunto de cambios a aplicar en la estructura de la instancia a través del objeto <Type>Spec. Internamente, el objeto <Type>Spec almacena los cambios realizados a la especificación del tipo como diferencias respecto a la especificación original. De esta forma, el proceso de evolución de tipos se realiza como un proceso de evolución incremental, a través de operaciones atómicas que van modificando la estructura original de la instancia en ejecución, bien introduciendo nuevos elementos, bien eliminándolos. El aspecto InstanceEvolutionPlanning extrae el conjunto de operaciones atómicas de evolución a

realizar. Cada operación de evolución implica que la parte estructural a ser modificada alcance previamente un estado quiescente (esto es, finalice primero sus transacciones de forma segura), a excepción de las conexiones, en cuyo caso son sus extremos quien tienen que alcanzar dicho estado. Además, puede que para alcanzar un estado quiescente, también deban ser detenidas las partes estructurales directamente conectadas, lo que también es tenido en cuenta. Sin embargo, no se describirá aquí cómo se alcanza la quiescencia, por quedar fuera del ámbito de este artículo. Para llevar a cabo todo este proceso, el aspecto InstanceEvolutionPlanning va coordinando los distintos servicios ofrecidos por los aspectos EvolutionSensor y EvolutionEffector, utilizando los servicios del primero para obtener referencias y el estado de las partes estructurales de la instancia y los servicios del segundo para ir aplicando los cambios.

Es importante señalar que mientras la meta-instancia sí que genera nuevo código y lo persiste en disco, a nivel de instancias sólo se alteran estructuras en memoria, para adaptarlas al nuevo tipo.

3.3 La Evolución Dinámica de Tipos como un Crosscutting Concern

De acuerdo con el enfoque de Desarrollo de Software Orientado a Aspectos (AOSD), un aspecto representa un comportamiento específico de un asunto (*concern*) –seguridad, coordinación, distribución, etc.- que se encuentra disperso (*crosscutting concern*) en un sistema software complejo. La evolución dinámica de tipos puede considerarse también un asunto de interés del sistema, pues está presente en todos aquellos tipos que sean evolucionables y en sus respectivas instancias.

Además, un *concern* puede ser representado por varios aspectos, y es efectivamente como se ha materializado en nuestra propuesta, desarrollando cada aspecto una parte de la funcionalidad de la evolución de tipos. Algunos aspectos proporcionan la funcionalidad independiente de plataforma, describiendo el proceso de evolución en términos del metamodelo utilizado (PRISMA) y coordinando las acciones a realizar a un alto nivel de abstracción, como es el caso de los aspectos EvolutionPlanning, EvolutionMonitoring y InstanceEvolutionPlanning. Por otro lado, otros aspectos, como Builder, TypeDescription, EvolutionSensor y EvolutionEffector son los que realizan el puente entre los conceptos independientes de plataforma (los de PRISMA) y los conceptos dependientes de tecnología (la implementación del modelo de componentes en una determinada plataforma). Los servicios ofrecidos por estos últimos aspectos son dependientes del metamodelo PRISMA, pero su implementación es realizada en la tecnología en la que las arquitecturas PRISMA se ejecuten (actualmente .NET [20]). De este modo, esta separación de aspectos facilita la portabilidad de la implementación PRISMA a otras plataformas, en las que los cambios a realizar en el modelo de evolución están localizados en estos cuatro últimos aspectos, mientras que el resto de aspectos sólo utilizan conceptos independientes de plataforma.

Por otro lado, otra ventaja derivada de los modelos orientados a aspectos es que un mismo aspecto puede ser importado por más de un elemento arquitectónico que necesite tener en cuenta el comportamiento del *concern* que dicho aspecto define. En el caso del concern de evolución dinámica, todos los tipos arquitectónicos PRISMA que incorporen soporte a la evolución dinámica de tipos importarán este conjunto de aspectos, favoreciendo la reutilización y el mantenimiento del código de evolución de toda la aplicación.

4. Discusión

Muchos trabajos han abordado el problema de la evolución dinámica de sistemas software [4, 15]. Wang [24] describe cómo soportar la evolución dinámica de componentes (simples) en Java. Para ello, modifica el classLoader de Java para que para nuevas instanciaciones usen el nuevo tipo modificado. Respecto a las instancias, sólo son migradas sino están ejecutando ningún servicio, y la transferencia de estado se realiza mediante los mecanismos de reflexión de Java. El bloqueo de

las peticiones entrantes se realiza a través de un gestor de instancias de componentes, que intercepta todas las conexiones dirigidas a las instancias. Es un enfoque más dirigido a arquitecturas orientadas a servicios, pues las instancias son creadas bajo petición y tienen una vida más corta, lo que garantiza que en algún momento dejen de procesar servicios y puedan ser migradas al nuevo tipo. Sin embargo, la migración del estado se realiza para estructuras muy sencillas. En [22], Vandewoude aborda cómo tratar la migración de estructuras de datos complejas entre distintas versiones de tipos, y en [23], cómo alcanzar un estado tranquilo, menos restrictivo que la quiescencia, para evolucionar las instancias en ejecución de forma segura.

Estos trabajos son bastante interesantes, aunque todos ellos llevan a cabo el proceso de evolución de forma centralizada: la infraestructura de ejecución (esto es, el middleware) proporciona los mecanismos para evolucionar todos los tipos presentes en el sistema. Sin embargo, este enfoque no es adecuado, pues no es escalable para sistemas distribuidos, y tan sólo es aceptable para sistemas homogéneos (todos los elementos del sistema son de la misma tecnología). En nuestra propuesta, la evolución de tipos es proporcionada de forma independientemente para cada tipo del sistema software en ejecución. La escalabilidad es proporcionada por los aspectos, ya que todos los tipos que utilicen la misma tecnología (nuestro enfoque) importarán los mismos aspectos, que están definidos una sola vez en todo el código.

En el área de las arquitecturas software hay numerosos trabajos que abordan la adaptabilidad en tiempo de ejecución [3], aunque en su mayoría abordan sólo la reconfiguración dinámica. Esto es debido a que numerosos autores no han establecido la distinción entre reconfiguración dinámica y evolución de tipos que se ha descrito en este trabajo: con frecuencia se refiere a la evolución de tipos cuando se evolucionan dinámicamente elementos arquitectónicos simples (componentes), y se refiere a la reconfiguración dinámica cuando lo que se modifican son arquitecturas software, que no están restringidas a la especificación de un tipo arquitectónico, como sí lo están en el enfoque PRISMA. La mayor parte de ADLs que incorporan soporte para el dinamismo arquitectónico, como LEDA [6], PiLaR [9], Plastik [1] o SOFA [2], no describen cómo darle soporte a dicho dinamismo, ya que se centran en cómo describir dicho dinamismo, dejando para la fase de implementación la decisión de cómo llevar a cabo dicho proceso evolutivo. Nuestra propuesta sigue un enfoque mixto: se describen a alto nivel los cambios a realizar (en términos del ADL elegido, PRISMA) y se han identificado y ubicado los mecanismos que lo hacen posible, con el objetivo de que las entidades definidas en el ADL puedan también interactuar con dichos mecanismos (i.e., un tipo arquitectónico puede invocar servicios de la meta-instancia de otro tipo para evolucionarlo). El uso de la reflexión de este trabajo es similar a la propuesta en el ADL PiLaR, en la que cada sistema tiene acceso a la representación editable de sí mismo.

Los trabajos de Dashofy [10] y Garlan [11] sí que describen la infraestructura necesaria para describir sistemas software auto-reparables y auto-adaptables, respectivamente, basándose en modelos que describen la arquitectura válida del sistema. El inconveniente de estos trabajos es precisamente la no descentralización de los mecanismos de evolución, bajo la suposición de que todos los subsistemas deben ser reconfigurables y accesibles. Morrison et al. [16] describe a un sistema evolucionable como aquel compuesto por dos procesos: Producer, que proporciona la funcionalidad del sistema, y Evolver, que es capaz de evolucionar al otro proceso. La evolución se realiza descomponiendo el proceso Producer en los elementos que lo constituyen y siendo detenidas todas las conexiones entre ellos, aunque hubiera elementos que no fueran afectados por la evolución del tipo. Su propuesta es bastante cercana a la desarrollada en este trabajo, ya que el proceso encargado de evolucionar a un sistema está embebido dentro de él (aunque separado para facilitar su mantenimiento), y sigue un proceso de descomposición del sistema para evolucionar el tipo en ejecución. La propuesta de Morrison sigue un enfoque completamente orientado a procesos, mientras que nuestro trabajo se centra en el área de arquitecturas software, utilizando aspectos para separar los mecanismos de evolución respecto al resto de funcionalidad del sistema.

5. Conclusiones y trabajos futuros

En este trabajo se ha presentado una propuesta para soportar la evolución dinámica de tipos, aplicada al campo de las arquitecturas software, y al enfoque PRISMA en particular. Esta propuesta describe una infraestructura para dotar a cada tipo arquitectónico (simple o complejo) de la capacidad de ser evolucionado en tiempo de ejecución de forma independiente, sin necesidad de definir una entidad centralizada para evolucionar todos los tipos de un sistema. De esta forma, los tipos así construidos pueden integrarse en sistemas heterogéneos y distribuidos. Además, también se preserva el principio de encapsulación: un tipo arquitectónico es una caja negra, y como tal, su evolución sólo puede realizarse por los mecanismos internos del tipo, que son los que conocen la estructura interna del tipo y cómo cambiarla. Desde un punto de vista externo, un tipo evolucionable proporciona capacidades reflexivas para obtener su reificación y un conjunto de servicios de evolución para modificar su especificación de forma consistente. Desde un punto de vista interno, el proceso de evolución de tipos está separado en distintos asuntos o áreas de interés, que se encuentran distribuidos entre la reificación del tipo y sus instancias en ejecución. Estos asuntos o concerns han sido encapsulados en aspectos, favoreciendo así su reutilización y mantenimiento. Los aspectos a nivel del tipo se encargan de evolucionar la especificación y los servicios de creación y destrucción de instancias, mientras que los aspectos a nivel de instancias se encargan de evolucionar la estructura interna de cada instancia. Otra contribución de este trabajo es que la evolución a nivel de instancias se realiza a través de la descomposición de la estructura interna de la instancia y mediante un proceso de desarrollo incremental, añadiendo/eliminando las entidades que se han añadido/eliminado del tipo.

Actualmente, estamos incorporando los conceptos descritos en este trabajo al middleware PRISMANET [20], que soporta la ejecución de arquitecturas PRISMA y su reconfiguración dinámica [7]. Una vez finalizada la implementación, se llevará a cabo un estudio para estimar los tiempos de respuesta del proceso de evolución y compararlos con otras propuestas. Otro de los trabajos que esperamos realizar a corto plazo es la definición de restricciones en el proceso de evolución de tipos: por ejemplo, para limitar qué partes del tipo pueden ser evolucionadas y cuáles no.

Agradecimientos

Este trabajo ha sido financiado por el CICYT (Comisión Interministerial de Ciencia y Tecnología), proyecto META TIN2006-15175-C05-01, y cofinanciado por la Comunidad de Madrid y la Universidad Rey Juan Carlos dentro del proyecto IASOMM URJC-CM-2007-CET-1555. También ha sido financiado parcialmente gracias a una beca FPI de la Conselleria d'Educació i Ciència (Generalitat Valenciana) concedida a C. Costa.

Referencias

1. Batista, T., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-Based Systems. In 2nd European Workshop on Software Architectures (EWSA'05). LNCS, vol. 3527. Springer (2005)
2. Bures, T., Hnetynka P., Plasil F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In: 4th Int. Conference on Software Engineering Research, Management and Applications (SERA'06). Seattle, Washington, USA (2006) 40-48
3. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. In proc. of 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04). Newport Beach, California (2004) 28-33
4. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. Journal on Software Maintenance and Evolution, vol. 17(5). Wiley (2005)
5. Cámara, J., Canal, C., Cubo, J., and Murillo, J. M. An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. *Electron. Notes Theor. Comput. Sci.* 189 (2007) 21-34

6. Canal, C., Pimentel, E., Troya, J.M.: Specification and Refinement of Dynamic Software Architectures. In: First Working IFIP Conf. on Software Architecture (WICSA'99). San Antonio, Texas, USA (1999)
7. Costa, C., Ali, N., Pérez, J., Carsí, J.A., Ramos, I.: Dynamic Reconfiguration of Software Architectures Through Aspects. In: 1st Europ. Conf. on Software Architecture (ECSA'07). LNCS, vol. 4758 (2007)
8. Costa, C., Pérez, J., Carsí, J.A: Dynamic Adaptation of Aspect-Oriented Components. In: 10th Int. ACM Symp. on Component-Based Software Engineering (CBSE'07). LNCS, vol. 4608. Springer (2007) 49-65
9. Cuesta, C.E., Fuente, P.d.l., Barrio-Solárzano, M.: Dynamic Coordination Architecture through the use of Reflection. In: ACM Symposium on Applied Computing. Las Vegas, Nevada, USA (2001) 134-140
10. 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). Charleston, South Carolina (2002) 21-26
11. Garlan, D., Cheng, S., Huang, S., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In: Computer, Vol. 37. IEEE Computer Society (2004) 46-54
12. Harrison, W. H., Ossher, H. L., Tarr, P. L.: Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Tech. Rep. RC22685 (W0212-147), T. J. Watson Research Centre, IBM (2002)
13. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. In: IEEE Transactions on Software Engineering, Vol. 16, No.11 (1990), 1293-1306
14. Maes, P.: Concepts and Experiments in Computational Reflection. In: SIGPLAN Not., Vol. 22, No. 12. ACM Press, New York, NY, USA (1987) 147-155
15. McKinley, P. K., Sadjadi, S. M., Kasten, E. P. and Cheng, B. H. C.: Composing Adaptive Software. In: Computer, vol. 37(7). IEEE Computer Society (2004) 56-64
16. Morrison, R., Balasubramaniam, D., Kirby,G., et al: A Framework for Supporting Dynamic Systems Co-Evolution. In: Autom. Software. Eng, Vol. 14(3). Springer (2007) 261-292
17. Perry, D. E., & Wolf, A. L.: Foundations for the Study of Software Architecture. In: ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4 (1992) 40-52
18. Pérez, J. PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Polytechnic University of Valencia (2006).
19. Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In proc. of the 9th Int. Symp. on Component-Based Software Engineering (CBSE06). Lecture Notes on Computer Science, Vol. 4063. Springer (2006) 123-138
20. 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. Pilsen, Czech Republic (2005) 97-108
21. Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: Dynamic Evolution in Aspect-Oriented Architectural Models. In 2nd European Workshop on Software Architecture (EWSA'05). LNCS, Vol. 3527. Springer (2005) 59-76
22. Vandewoude, Y., Berbers, Y.: Component state mapping for runtime evolution. In proc. of Int. Conf. on Programming Languages and Compilers. Las Vegas, Nevada, USA (2005)
23. Vandewoude, Y., Ebraert, P., et al.: Tranquillity: A low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. IEEE Transactions on Soft. Eng., Vol. 33, No. 12. (2007) 856-868
24. 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 (2006)