

**Proceedings of the
Third International Workshop
on Autonomic and Self-Adaptive Systems
WASELF'10**

September 7, 2010
Valencia, Spain

Held in conjunction with CEDI 2010

Workshop Organizers:

Javier Cámara (INRIA Rhône Alpes, France)
Carlos E. Cuesta (Universidad Rey Juan Carlos, Spain)
Miguel Á. Pérez-Toledano (Universidad de Extremadura, Spain)

PREFACE

Traditionally, handling changing requirements, faults, or upgrades on different kinds of software-based systems have been tasks performed as a maintenance activity conducted by human operators at design/development time. However, factors such as uncertainty in the operational environment, resource variability, or the critical nature of some systems which cannot be halted in order to be changed, have lead to the development of systems able to replan and reconfigure their structure and behaviour at run time in order to improve their operation without any human intervention.

This kind of systems which typically operate using an explicit representation of their structure and goals has been studied within different research areas of software engineering (e.g., component-based development, requirements engineering, software architectures, etc.) and described with different names (self-adaptive, self-healing, self-managed, autonomic systems, etc.).

Another promising approach to systems able to dynamically adapt themselves is self-organizing systems. These are typically composed by a large number of constituent components which operate according to a set of local rules, rather than with an explicit representation of its structure and goals. In this case, the emergent behaviour derived from component interaction stabilizes the system in the event of faults or changes in the environment which need to be handled.

Although most research efforts in both approaches have been isolated and lacked specific forums for discussion until recently, there is a thriving international community currently involved in the study of self-* systems, laying out the foundations that will enable their systematic development. Likewise, the goal of this workshop is gathering software engineering researchers from fields related to the development of self-* systems in order to identify critical research challenges, as well as discussing models, techniques, tools, industrial cases, and methodologies for the development of complex systems able to dynamically adapt their behaviour. Moreover, the aim of the workshop is addressing all these topics stressing the importance of integrating different achievements and devising generic approaches to self-* systems engineering.

These are the proceedings of the Third Workshop on Autonomic and Self-Adaptive Systems, affiliated with the CEDI 2010, held in Valencia, Spain, from September 7th to 10th, 2010. These proceedings contain the 4 papers selected for participating in the workshop.

Javier Cámara
Carlos E. Cuesta
Miguel Á. Pérez-Toledano
Workshop Organizers

Organizing Committee

Javier Cámara (INRIA Rhône-Alpes, France)

Carlos Cuesta (Universidad Rey Juan Carlos, Spain)

Miguel Ángel Pérez-Toledano (Universidad de Extremadura, Spain)

Program Committee

Nelly Bencomo (Lancaster University, UK)

Javier Cámara (INRIA Rhône-Alpes, France)

Carlos Canal (Universidad de Málaga, Spain)

Pedro José Clemente Martín (Universidad de Extremadura, Spain)

Sorana Cîmpan (University of Savoie, France)

Carlos Cuesta (Universidad Rey Juan Carlos, Spain)

Juan Manuel Murillo (Universidad de Extremadura, Spain)

Jennifer Pérez (Universidad Politécnica de Madrid, Spain)

Miguel Ángel Pérez-Toledano (Universidad de Extremadura, Spain)

Pascal Poizat (University of Evry and LRI, France)

Pilar Romay (Universidad San Pablo-CEU, Spain)

Antonio Ruíz-Cortés (Universidad de Sevilla, Spain)

Gwen Salaün (Grenoble INP-INRIA-LIG, France)

Tabla de Contenidos

- *Reconfiguración Dinámica de Arquitecturas Software Aplicada a la Tolerancia a Fallos*

Páginas 1-10

Autores: Cristobal Costa-Soria, Jennifer Perez, José A. Carsí, Diego Alonso, Francisco Ortiz, Juan Ángel Pastor

- *Modelos Arquitectónicos Orientados a Servicios: Definición y Desarrollo Formal sobre un Caso de Estudio*

Páginas 11-20

Autores: Marcos López-Sanz, Carlos E. Cuesta, Esperanza Marcos

- *Marco de trabajo orientado a aspectos para el diseño de arquitecturas software adaptables dinámicamente*

Páginas 21-30

Autores: Amparo Navasa, Miguel Angel Perez-Toledano, Juan Manuel Murillo

- *Auto-Adaptación en Organizaciones de Agentes: de la Iniciativa al Acuerdo Emergente*

Páginas 31-39

Autores: J. Santiago Pérez-Sotelo, Carlos E. Cuesta, Sascha Ossowski

Reconfiguración Dinámica de Arquitecturas Software Aplicada a la Tolerancia a Fallos

Cristóbal Costa-Soria¹, Jennifer Pérez², Jose A. Carsí¹,
Diego Alonso³, Francisco Ortiz³, Juan Ángel Pastor³

¹ Dpto. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Cno. de Vera s/n, 46022 Valencia

ccosta@dsic.upv.es, pcarsi@dsic.upv.es

² E.U. Informática, Universidad Politécnica de Madrid, Ctra. Valencia km 7, 28051 Madrid

jenifer.perez@eui.upm.es

³ Depto. Tecnología Electrónica, Universidad Politécnica de Cartagena, Campus Muralla del Mar s/n, 30202 Cartagena

diego.alonso@upct.es, francisco.ortiz@upct.es, juanangel.pastor@upct.es

Resumen

En la actualidad, el desarrollo de sistemas software tolerantes a fallos se realiza a un nivel dependiente de la tecnología, con lo que aumenta su complejidad y disminuye la reutilización. La mayoría de estrategias de tolerancia a fallos son estáticas: se basan en replicar elementos críticos para que, ante cualquier fallo, sus réplicas tomen el relevo. En este trabajo se describe cómo la reconfiguración dinámica de arquitecturas software puede aplicarse para desarrollar sistemas tolerantes a fallos. Las técnicas de reconfiguración dinámica permiten cambiar la configuración de sistemas software complejos en tiempo de ejecución, sin necesidad de detener el sistema. Este artículo describe cómo la reconfiguración dinámica es soportada a nivel de arquitecturas software y mediante aspectos, separando la funcionalidad de reconfiguración –y las políticas de recuperación frente a fallos– del resto de funcionalidades del sistema. Esto se ilustra mediante la definición de las políticas de recuperación del sistema de visión del Agrobot, un sistema robótico del ámbito agrícola.

1. Introducción

Hoy en día, la mayoría de sistemas software son complejos, distribuidos y dinámicos. Esto ha hecho que la flexibilidad, seguridad y tolerancia a fallos sea un factor fundamental en su proceso de desarrollo. La tolerancia a fallos es un aspecto ampliamente cubierto en el área de los sistemas

robóticos, pero esto se realiza a nivel de implementación, con lo que las soluciones son dependientes de tecnología y poco reutilizables.

Existen numerosas estrategias de diseño y patrones arquitectónicos para la detección y tolerancia a fallos [20]. Estas estrategias se basan en el uso de diferentes tipos de redundancia, que se caracterizan por tener varias réplicas de los elementos críticos ejecutándose concurrentemente en el sistema. En el caso de fallar uno de estos elementos, su funcionalidad continúa siendo ofrecida gracias al resto de réplicas en ejecución, que permiten “ocultar” el fallo. Este enfoque clásico de redundancia presenta la ventaja de que el tiempo de recuperación frente al fallo es inmediato, puesto que las réplicas ya se encuentran en ejecución y tienen el mismo estado que el elemento que ha fallado, y que la sustitución del elemento defectuoso es sencilla.

Sin embargo, este enfoque presenta una serie de inconvenientes: (i) los puntos de sincronización donde se comparan los resultados de las réplicas son difíciles de definir y de implementar; (ii) las réplicas consumen recursos del sistema; (iii) la complejidad del sistema aumenta, debido a que el número de mensajes entre las diferentes partes del sistema crece rápidamente con el número de réplicas; y (iv) este enfoque es estático, no puede adaptarse ante cambios del sistema durante su ejecución.

Una alternativa al uso de la redundancia para soportar tolerancia a fallos es el uso de técnicas de reconfiguración dinámica [32]. La reconfiguración dinámica permite dotar de flexibilidad a un

sistema software, permitiendo tanto cambiar su configuración como sustituir unos elementos por otros en tiempo de ejecución. El uso de la reconfiguración dinámica en el diseño de sistemas tolerantes a fallos permitiría evitar tanto la complejidad de mantener operativos de forma simultánea a varios componentes redundantes, como el consumo de recursos de computación adicionales.

Sin embargo, el desarrollo y mantenimiento de sistemas reconfigurables es complejo y costoso. Por un lado, su desarrollo es complejo ya que los mecanismos de reconfiguración son dependientes de la tecnología [24]: dependiendo de la tecnología empleada y de cómo se desarrolle el sistema, la forma de detener de un modo seguro partes del sistema variará, así como el modo de cambiar su configuración. Por otro lado, el mantenimiento es costoso, ya que las especificaciones de reconfiguración (i.e. las políticas de reconfiguración) suelen estar entremezcladas con la funcionalidad del sistema. Esto es debido a que la ejecución de la funcionalidad del sistema es la que genera la necesidad de reconfiguración, y por tanto, a través de la funcionalidad se detecta cuándo debe reconfigurarse el sistema.

Es por esto que, para facilitar el desarrollo y mantenimiento de sistemas dinámicamente reconfigurables: (i) la reconfiguración debe describirse a un mayor nivel de abstracción, independientemente de los detalles específicos de tecnología (e.g. los mecanismos que soportan la reconfiguración del sistema en tiempo de ejecución) y, (ii) la funcionalidad del sistema debe estar claramente separada de la especificación de cómo debe reconfigurarse el sistema en tiempo de ejecución. Para ello, pueden seguirse dos enfoques complementarios: las arquitecturas software para describir sistemas software a un alto nivel de abstracción, y el desarrollo orientado a aspectos para separar la reconfiguración de la funcionalidad del sistema.

Las arquitecturas software [29] describen un sistema en términos de elementos arquitectónicos (componentes y conectores) y las interacciones entre sí (conexiones). La reconfiguración dinámica de arquitecturas permite cambiar la composición de un sistema en ejecución en términos de creación/destrucción de instancias de elementos arquitectónicos y de la modificación de las conexiones entre ellos [15],[23]. Por otro lado,

el Desarrollo de Software Orientado a Aspectos (AOSD) [22] propone la separación de los *crosscutting-concerns*¹ de un sistema software en entidades separadas denominadas aspectos. La separación de dichos *crosscutting-concerns* en entidades separadas permite su reutilización en distintas entidades del sistema software a la vez que facilita su mantenimiento. La reconfiguración dinámica es intrínsecamente un *crosscutting-concern*: las reglas de reconfiguración están mezcladas con el código funcional y se encuentran diseminadas por todo el sistema. Por esta razón, la encapsulación del código de reconfiguración en aspectos, ayudará al desarrollo de sistemas reconfigurables a la vez que facilitará su mantenimiento.

En este trabajo se describe cómo pueden definirse estrategias de recuperación frente a fallos: (i) a través de la reconfiguración dinámica, evitando el uso excesivo de elementos redundantes y optimizando el uso de recursos; (ii) a un alto nivel de abstracción, mediante arquitecturas software, contribuyendo a reducir la complejidad del sistema; y (iii) mediante aspectos, separando la funcionalidad de reconfiguración/recuperación del resto de la funcionalidad del sistema software, con lo que se aumenta la reutilización de dichos artefactos. Para ello, se ha elegido el modelo PRISMA, el cual permite especificar arquitecturas software orientadas a aspectos independientes de plataforma y da soporte al Desarrollo Dirigido por Modelos, generando código C# a partir de sus arquitecturas [26], [27].

Las ideas presentadas en este trabajo se ilustran con un caso de estudio del dominio robótico. El motivo de elegir este dominio es debido a que, aunque la robótica engloba múltiples disciplinas (como la mecánica, la electrónica y la informática), la capacidad de un robot para actuar de forma autónoma e *inteligente* reside cada vez más en su software de control. Por ello, para superar las limitaciones de los robots es imprescindible superar las limitaciones del software que gobierna su comportamiento. Para manejar esta complejidad es de gran ayuda el uso de las herramientas y metodologías proporcionadas por la Ingeniería del Software.

¹ Funcionalidad transversal de un sistema: auditoría, persistencia, seguridad, comunicación distribuida, etc.

La estructura del artículo es la siguiente: en la sección 2 se introduce PRISMA y los aspectos que ofrecen soporte a la reconfiguración dinámica de arquitecturas. En la sección 3 se introduce el Agrobot y el diseño de una estrategia de recuperación mediante nuestra propuesta. Finalmente, en las secciones 4 y 5 se describen los trabajos relacionados y las conclusiones.

2. Reconfiguración dinámica de Arquitecturas Software

El proceso de reconfiguración dinámica puede ser de dos tipos, según como se inicie dicho proceso [15]: *ad-hoc*, si el proceso de reconfiguración es iniciado y dirigido en tiempo de ejecución por un agente externo al propio sistema; o *programado*, si el proceso de reconfiguración es iniciado y dirigido en tiempo de ejecución por el propio sistema, en base a un conjunto de reglas de reconfiguración definidas en el sistema. La reconfiguración dinámica programada puede usarse para describir sistemas auto-organizados, i.e. sistemas capaces de reconfigurar su topología por sí mismos en tiempo de ejecución. En particular, la reconfiguración programada puede usarse para describir sistemas tolerantes a fallos, a través de la definición de las estrategias de recuperación a realizar tras detectar un fallo.

En un sistema software complejo, como es el caso de los sistemas robóticos, la reconfiguración dinámica programada puede ser necesaria, además de para el propio sistema, para cada uno de los subsistemas que lo forman (i.e. componentes complejos). Para ello, puede seguirse un enfoque de reconfiguración centralizado o un enfoque descentralizado. Por un lado, en el enfoque centralizado [12][17] una entidad global describe todas las reglas de reconfiguración del sistema: tanto las reglas del propio sistema como las reglas de reconfiguración de los distintos subsistemas. Este enfoque tiene la ventaja de que todo el proceso de reconfiguración del sistema se encuentra centralizado, facilitando el mantenimiento de los distintos procesos de reconfiguración. Sin embargo, este enfoque no es apropiado, ya que presupone que la arquitectura de cada uno de los subsistemas es accesible para el configurador global (por lo que se viola el principio de encapsulación) y reconfigurable (no todos los subsistemas necesitarán ser

reconfigurados). Por otro lado, siguiendo un enfoque descentralizado [25], tanto las reglas como los mecanismos de reconfiguración se describen dentro de cada subsistema. Esto tiene la ventaja de que cada subsistema determinará si es reconfigurable o no y cómo debe reconfigurarse, por lo que se respeta el principio de encapsulación y se facilita el mantenimiento de cada subsistema.

En este trabajo se presenta una propuesta para describir la reconfiguración programada de sistemas software de un modo descentralizado, con el objetivo de poder diseñar sistemas capaces de reconfigurarse, que a su vez puedan estar formados por otros subsistemas reconfigurables. La propuesta se basa en la identificación de la funcionalidad necesaria para proporcionar soporte a la reconfiguración dinámica, y su encapsulación en aspectos, con el objetivo de beneficiarse de las ventajas derivadas del enfoque AOSD, como la reutilización y la facilidad de mantenimiento.

Es por esta razón que la propuesta de reconfiguración dinámica se ha desarrollado sobre el modelo PRISMA [27], dado que permite la descripción de arquitecturas software orientadas a aspectos, y su compilación posterior a código C#, ejecutable en el middleware PRISMANET [26].

2.1. PRISMA

PRISMA [27] es una propuesta para el desarrollo de arquitecturas software orientadas a aspectos independientes de tecnología. Esta propuesta se caracteriza por definir un modelo de aspectos simétrico [19], en el que toda la funcionalidad de un elemento arquitectónico se encapsula mediante aspectos. Un aspecto representa un conjunto de funcionalidades comunes (*concern*) -seguridad, coordinación, reconfiguración, etc.-, que puede ser importado por aquellos elementos arquitectónicos que necesiten incorporar el comportamiento del “concern” que dicho aspecto define. Como resultado, un elemento arquitectónico es definido por un conjunto de aspectos que describen las distintas facetas del elemento arquitectónico.

Cada elemento arquitectónico PRISMA encapsula su funcionalidad como una caja negra y publica un conjunto de servicios que ofrecen a otros elementos arquitectónicos. Estos servicios, agrupados en interfaces, son publicados a través de los puertos, que son los puntos de interacción entre elementos arquitectónicos. Sin embargo, la

vista interna de dichos elementos arquitectónicos difiere entre simples (componentes y conectores) y complejos (sistemas).

Por un lado, la vista interna de un elemento arquitectónico simple es una composición invasiva de aspectos [1], que puede ser vista como un prisma (ver Figura 1, izquierda). Cada lado del prisma es un aspecto que el elemento arquitectónico importa. Los aspectos son sincronizados entre ellos a través de *weavings*, que indican cómo la ejecución del servicio de un aspecto activará la ejecución de servicios en otros aspectos.

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

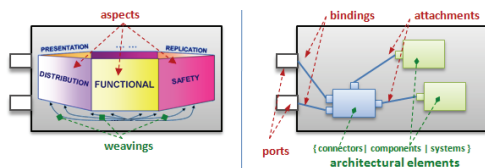


Figura 1. Vista interna y externa de un elemento arquitectónico PRISMA

2.2. Reconfiguración Dinámica en PRISMA: el Componente Evolver

En PRISMA, la capacidad de reconfiguración dinámica se ha modelado a través de un componente especial, denominado *Evolver*. Este componente proporciona los servicios y mecanismos necesarios de reconfiguración, y es importado por cada sistema que requiera ser reconfigurable en tiempo de ejecución. De esta forma, el compilador de modelos PRISMA puede determinar qué sistemas tienen que generarse con la infraestructura necesaria para poder ser reconfigurados dinámicamente y cuáles no.

El componente *Evolver* encapsula el *concern* de reconfiguración dinámica del sistema al que pertenece. El *concern* de reconfiguración

dinámica es proporcionado por cuatro aspectos distintos y complementarios (ver Figura 2), los cuales se describen brevemente a continuación.

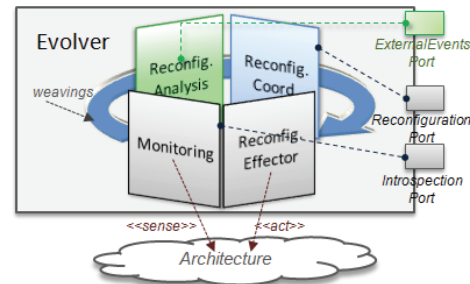


Figura 2. El componente *Evolver* y los aspectos de Reconfiguración

El aspecto *Reconfiguration Analysis* describe el conjunto de reconfiguraciones programadas a realizar en una arquitectura específica (aquella donde ha sido importado el *Evolver*) y el conjunto de eventos que activarán el proceso de reconfiguración.

El aspecto *Reconfiguration Coordination* proporciona los servicios de reconfiguración disponibles, verifica que se cumplan las restricciones estipuladas en el tipo del sistema (i.e. qué reconfiguraciones pueden realizarse y cuáles no), y coordina la ejecución transaccional de los servicios de reconfiguración para así garantizar que las reconfiguraciones se realizan de forma segura y en caso de que algo falle, devolver el sistema a su estado previo. Este aspecto garantiza que sólo los elementos arquitectónicos que van a ser reconfigurados sean detenidos de forma segura (i.e. alcanzando un estado quiescente), y que sólo cuando los elementos afectados por la reconfiguración estén preparados, se realice la reconfiguración.

El aspecto *Monitoring* proporciona servicios para monitorizar la arquitectura y recoger información relevante para activar el proceso de reconfiguración. Este aspecto proporciona tres tipos de información: (i) eventos ocurridos en la arquitectura (e.g. la invocación de servicios); (ii) información relativa a la configuración actual, lo que permite al sistema ser consciente de su estado: cuáles son sus instancias arquitectónicas actualmente en ejecución y cómo están conectadas; y (iii) información interna de los elementos en ejecución, como el estado de

ejecución: *idle* (parado, a la espera), *active* (está procesando servicios) o *blocked* (detenido temporalmente, listo para reconfigurar).

Por último, el aspecto *Reconfiguration Effector* es el que proporciona los servicios dependientes de tecnología para cambiar la arquitectura del sistema en tiempo de ejecución: (i) servicios para parar/iniciar la ejecución de los elementos arquitectónicos, (ii) servicios para crear/destruir instancias, (iii) servicios para crear/destruir conexiones y (iv) servicios para transferir el estado.

Los aspectos Monitoring, Reconfiguration Effector y Reconfiguration Coordination encapsulan los mecanismos genéricos de reconfiguración. Estos aspectos son predefinidos en el metamodelo de PRISMA y son importados por todos los sistemas reconfigurables. De este modo, el código de reconfiguración es centralizado en tres aspectos, facilitando su mantenimiento y reutilización. Por su parte, el aspecto Reconfiguration Analysis describe cómo debe reconfigurarse la arquitectura en la cual el Evolver ha sido importado, por lo que es distinto para cada sistema. Estos aspectos se sincronizan entre sí a través de weavings, los cuales se especifican fuera de los aspectos, en la definición del componente Evolver. Esto hace a los aspectos independientes entre sí, puesto que sus dependencias están aisladas en los weavings.

Por otro lado, el hecho de que el Evolver sea un elemento más de la arquitectura del sistema permite conectarlo con cualquier elemento arquitectónico. Por ejemplo, el puerto *externalEventsPort* (ver Figura 2) permite enviar eventos definidos por el usuario al aspecto Reconfiguration Analysis, y con ello activar el proceso de reconfiguración. Esto tiene sentido cuando se desea reconfigurar una arquitectura a consecuencia de eventos procedentes del exterior del sistema o a consecuencia de un evento específico generado por un componente interno al sistema. Detalles adicionales sobre el Evolver y los aspectos de reconfiguración pueden encontrarse en [10].

3. AgroBot: Uso de la Reconfiguración para la Recuperación Frente a Fallos

La agricultura es un sector especialmente relevante en la economía del levante español. La

robótica de servicios [16], [21] puede jugar un papel fundamental para abordar las dificultades del sector: costes elevados de mano de obra, competencia de países en vías de desarrollo, falta de agua, preocupación medioambiental, control de plagas, etc. Existen granjas inteligentes en las que los robots realizan misiones de interés para la investigación. Esto constituye un banco de pruebas excelente, ya que es posible empezar por casos de estudio muy sencillos y cercanos a la robótica industrial (caracterizados por la realización de tareas repetitivas) e ir aumentando progresivamente la complejidad, siendo dotados de mayor autonomía.

AgroBot es un robot agrícola cuyo objetivo es la supervisión de un área determinada para controlar la presencia de plagas y/o la falta de agua. El sistema de visión artificial juega un papel muy importante para la realización autónoma de tareas, como es el movimiento por el terreno y la inspección de plagas. Este sistema es un elemento crítico en el que debe incorporarse tolerancia a fallos: si éste deja de funcionar, el robot deja de ser autónomo y por tanto deja de ser útil.

Este sistema incorpora una tarjeta capturadora de vídeo y un componente hardware para el procesamiento de las imágenes capturadas. Dicho componente implementa algoritmos específicos para el procesamiento de vídeo, optimizados para procesar la información que necesita el robot. Su implementación se ha realizado en hardware por motivos de rendimiento: los algoritmos implementados directamente en hardware se ejecutan más rápidamente que los implementados en software.

Una forma de proporcionar tolerancia a fallos en el sistema de visión sería a través de la redundancia de los componentes críticos: introducir varias réplicas de la tarjeta capturadora de vídeo, introducir elementos de votación o sumadores para unificar los resultados, etc. Sin embargo, tener más de un componente de procesamiento de vídeo ejecutándose en el sistema es muy costoso, dado que dichos algoritmos consumen muchos recursos.

Una alternativa es dotar al sistema de visión de una implementación *software* del componente de procesamiento de imágenes, que aunque con menor rendimiento, puede instanciarse en tiempo de ejecución a través de la reconfiguración dinámica, permitiendo al sistema recuperarse de

un eventual fallo del componente en su versión *hardware*. A continuación se describe cómo se ha especificado la estrategia de recuperación frente a fallos del sistema de visión del AgroBot, concretamente, para recuperarse de un fallo en el componente de procesamiento de imágenes.

3.1. Arquitectura software del sistema de visión del AgroBot

El sistema AgroBot ha sido especificado en el lenguaje de descripción de arquitecturas de PRISMA, con el objetivo de poder definirlo a un alto nivel de abstracción y posteriormente generar automáticamente el código de la plataforma destino. La arquitectura software del sistema de visión del AgroBot es la que se muestra en la Figura 3.

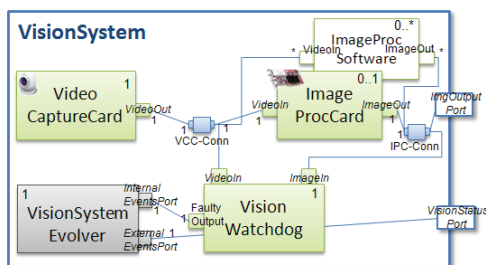


Figura 3. Arquitectura PRISMA del sistema de visión del AgroBot

El componente *VideoCaptureCard* encapsula a la tarjeta capturadora de imágenes. Dicho componente, a través del puerto, envía la secuencia de imágenes capturadas tras un intervalo de tiempo determinado. El componente *ImageProcCard* encapsula a la tarjeta procesadora de imágenes hardware, que procesa las imágenes recibidas de la tarjeta gráfica y extrae la información necesaria que será analizada por el resto de subsistemas del AgroBot. La información extraída de las imágenes es proporcionada al resto de subsistemas a través del puerto *ImgOutputPort*.

El componente *VisionWatchdog* monitoriza la ejecución de los componentes *VideoCaptureCard* e *ImageProcCard*, para comprobar que el funcionamiento es el esperado. Existen distintas estrategias de monitorización, de las cuales se ha elegido una variante del patrón *monitor-actuator* [14]: el monitor recibe parte de los datos de

entrada del componente monitorizado y los procesa lo suficiente para compararlos con los datos de salida del componente monitorizado y detectar si éstos están dentro de unos márgenes de error. La ventaja de esta estrategia de monitorización es que es relativamente fácil de implementar y consume pocos recursos de computación, ya que el algoritmo para generar el resultado esperado del componente monitorizado es sencillo. Si el componente *VisionWatchdog* detecta un fallo (los componentes monitorizados no envían datos o los envían fuera de los márgenes de error), este componente genera un evento "FaultyOutput" indicando el tipo de componente que ha fallado.

El componente *ImageProcSoftware* proporciona una implementación alternativa (vía software) al componente *ImageProcCard*, y se encuentra inicialmente deshabilitado (en blanco).

Estos cuatro componentes se encuentran conectados a través de los conectores VCC-Conn y IPC-Conn, respectivamente (en azul en la figura). Dichos conectores coordinan el envío y recepción de imágenes entre los componentes.

Por último, el componente *VisionSystem Evolver* proporciona al sistema de visión del AgroBot el soporte a la reconfiguración dinámica. Es en este componente donde se especifica cómo debe reconfigurarse el sistema, y es por tanto donde se han descrito las estrategias de recuperación frente a fallos. A través del puerto *ExternalEventsPort* (ver Figura 3), este componente recibe del componente *VisionWatchdog* la notificación de cuándo ha habido un fallo, activándose de este modo el proceso de recuperación frente al fallo notificado.

3.2. Reconfiguración dinámica del sistema de visión del AgroBot

El componente *VisionSystemEvolver* importa los siguientes aspectos: (i) los aspectos predefinidos *Reconfiguration Coordination*, *Monitoring* y *Reconfiguration Effector* (ver sección 2.2), que son los que proporcionan los mecanismos de reconfiguración; y (ii) el aspecto *VisionSystemEvolution*, que es un aspecto del tipo *Reconfiguration Analysis* (ver sección 2.2) y describe cómo debe reconfigurarse el sistema de visión del AgroBot ante determinados eventos (como la presencia de fallos en el sistema).

Los aspectos de tipo Reconfiguration Analysis se dividen en dos secciones: un conjunto de transacciones de configuración y un conjunto de triggers (disparadores) de reconfiguración. Una transacción de configuración es una especificación que describe un conjunto de operaciones de (re)configuración que deben llevarse a cabo transaccionalmente (todo o nada) para alcanzar una nueva configuración válida. De este modo, las solicitudes de servicios de configuración se ejecutarán secuencialmente, y si algo falla, la configuración será deshecha. Un trigger de reconfiguración es una condición que, si se evalúa a cierta, activa una transacción de configuración. Una condición puede utilizar: (a) la información proporcionada por el aspecto Monitoring, con la que puede comprobarse si un servicio ha sido invocado en la arquitectura, si una conexión ha sido creada o destruida, etc.; y/o (b) eventos externos o definidos por el usuario, como es el caso de la detección de un fallo por parte del componente VisionWatchdog.

En la Figura 4 se muestra un fragmento del aspecto VisionSystemEvolution, en el cual se describe cómo reconfigurar la arquitectura del sistema de visión tras detectar el fallo del componente ImageProcCard. La transacción *repairImageProcessingUnit* (ver Figura 4, sección “transactions”) define cómo debe reemplazarse el componente ImageProcCard por el componente ImageProcSoftware. El primer paso (ver proceso PREPARE) es obtener las referencias a las instancias que van a ser modificadas: el componente ImageProcCard que está fallando, y los conectores que lo enlazan al resto de elementos del sistema VCC-Conn y IPC-Conn. El siguiente paso es la reconfiguración (ver proceso RECONF en Figura 4): (1) se instancia el componente software para el procesamiento de imágenes; (2) se crea un attachment con el conector de la tarjeta capturadora de imágenes y (3) el conector que exporta los resultados; (4, 5) se destruyen las conexiones entre el componente con fallos y los respectivos conectores; y (6) se destruye la instancia del componente ImageProcCard, lo que implica desconectar físicamente el recurso hardware asociado (la tarjeta procesadora) para evitar que siga consumiendo energía del robot. Finalmente, el proceso END (7) indica el fin de la transacción de

reconfiguración, para que se confirmen los cambios si todo ha ido correctamente.

La activación de este proceso es realizada cuando se recibe el evento “FaultyOutput”, junto al tipo del componente que está fallando, *failingComponent*, por parte del componente VisionWatchdog. Este hecho se describe en la sección de *triggers* (ver Figura 4).

```

Reconfiguration Analysis aspect
  VisionSystemEvolution
...
Transactions
RepairImageProcessingUnit() :
  PREPARE ::=
    oldImProcCardID=imageProcCard-list[0] -->
    VCCConnID=VCC-Conn-list[0] -->
    IPCConnID=IPC-Conn-list[0] --> RECONF;
  RECONF ::=
1   create-ImageProcSoftware!(cameraPos,
    output newImProcID) -->
2   attach-Att VCCConn IPCSW!(VCCConnID,
    newImProcID, output newAttID) -->
3   attach-Att_IPCSW_IPCConn!(newImProcID,
    IPCConnID, output newAttID) -->
4   detach-Att_VCCConn_IPC!(VCCConnID,
    oldImProcCardID) -->
5   detach-Att_IPC_IPCConn!(oldImProcCardID,
    IPCConnID) -->
6   destroy-ImageProcCard!(oldImProcCardID) -->
7   END;
...
Triggers
RepairImageProcessingUnit() when
{failingComponent="ImageProcCard"}
  faultyOutput?(failingComponent);
...
End_Aspect VisionSystemEvolution;

```

Figura 4. Especificación del proceso de recuperación frente al fallo en el componente *ImageProcCard*

La ventaja de este enfoque es que, además de evitar el uso de la redundancia para la tolerancia a fallos, dota al sistema también de mecanismos para poder adaptarse a cualquier cambio en tiempo de ejecución: por ejemplo, reorganizar su arquitectura para que, en condiciones de baja energía, usar menos componentes o replicar aquellos que están siendo utilizados de una forma intensiva. Además, las reconfiguraciones programadas definidas en el aspecto Reconfiguration Analysis también pueden ser modificadas en tiempo de ejecución para permitir al sistema adaptarse a cambios no previstos inicialmente. Esto es realizado a través de la sustitución dinámica de aspectos [8], [9].

4. Trabajos relacionados

En los últimos años, numerosas propuestas han abordado el problema de la reconfiguración dinámica de sistemas software [3], [4], [11]. Sin embargo, pocos trabajos han tratado a la reconfiguración dinámica como un *crosscutting-concern*, con lo que con frecuencia el código de reconfiguración tiende a mezclarse con el código funcional. Rasche y Polze [30] utilizan aspectos para separar el código funcional del código de configuración del sistema, definiendo unos servicios de reconfiguración básicos. Otros trabajos, como Trap/J [31] o Cámara et al. [6] han utilizado el desarrollo orientado a aspectos para soportar la adaptación dinámica del software, a través de la agregación dinámica de aspectos. Sin embargo, estos trabajos están orientados hacia el uso de una tecnología concreta, mientras que nuestra propuesta se orienta a un nivel de abstracción mayor, las arquitecturas software.

En el área de las arquitecturas software, la reconfiguración dinámica puede proporcionarse de forma centralizada o de forma descentralizada. Por un lado, los enfoques centralizados tienen en común el uso de una entidad global que describe y gestiona cómo debe reconfigurarse todo el sistema. Los trabajos de Dashofy [12] y Garlan [17] proporcionan soporte a la reconfiguración dinámica programada, y la utilizan para describir la auto-reparación del sistema, basándose en modelos que describen la arquitectura válida del sistema. El inconveniente de estos trabajos es precisamente la no descentralización de la reconfiguración, bajo la suposición de que todos los subsistemas deben ser reconfigurables y accesibles. Por otro lado, la mayoría de enfoques descentralizados tienen en común el soporte de la reconfiguración dinámica a través de primitivas específicas del lenguaje, como LEDA [7], PiLaR [11], Plastik [2] o SOFA [5]. Estas primitivas pueden utilizarse en cualquier elemento arquitectónico para describir cuándo y cómo debe reconfigurarse la arquitectura del sistema. Sin embargo, esto provoca que el código correspondiente a la reconfiguración se entremezcle con el código funcional, dificultando así su mantenimiento. Morrison [25] describe una propuesta de evolución descentralizada muy cercana a nuestro trabajo, y aunque el código de evolución no se ha encapsulado en aspectos, sí ha

sido separado convenientemente del código funcional del subsistema al que pertenece.

Existen algunos trabajos en los que se ha utilizado la reconfiguración dinámica para diseñar sistemas tolerantes a fallos, aunque tan sólo se ha realizado de forma parcial o incompleta. Yurcik y Doss [29] describen las ventajas de utilizar la reconfiguración para la tolerancia a fallos, y con ello minimizar el uso de la redundancia, pero no describen cómo soportar la reconfiguración dinámica ni cómo facilitar el mantenimiento de dichos sistemas. En las arquitecturas Simplex [18], la reconfiguración se usa tan sólo para cambiar conexiones: si un elemento falla, el sistema se reconfigura para pasarle el control a otra de las réplicas que estaban en ejecución. Sin embargo, la descripción de la reconfiguración es dependiente de la tecnología y no se desarrolla la capacidad de crear dinámicamente nuevas instancias (cuya creación puede abstraer el hecho de activar físicamente un componente hardware que estaba instalado pero no en ejecución). De Lemos [13] describe un enfoque para la tolerancia a fallos basado en la gestión de excepciones y en la reconfiguración dinámica. En esta propuesta los fallos generan excepciones, que son manejadas por los elementos arquitectónicos, indicando cómo deben ser reconfigurados. El código de la reconfiguración se encuentra definido dentro de cada elemento arquitectónico, pero separado del código funcional, facilitando su mantenimiento. Sin embargo, no se describe cómo el sistema puede obtener información acerca de su configuración ni los mecanismos necesarios para ello.

5. Conclusiones y trabajos futuros

En este trabajo se ha presentado una aplicación de la reconfiguración dinámica de arquitecturas software para abordar el diseño de sistemas tolerantes a fallos.

Una de las contribuciones de esta propuesta es la encapsulación de toda la funcionalidad relativa a la reconfiguración dinámica en distintos aspectos: (i) Monitoring, que supervisa la arquitectura y proporciona información sobre la configuración actual del sistema; (ii) Reconfiguration Analysis, que define cuándo debe activarse el proceso de reconfiguración y cómo debe realizarse; (iii) Reconfiguration

Coordination, que coordina el proceso de reconfiguración para que se realice de una forma segura, y (iv) Reconfiguration Effector, que proporciona los servicios que modifican la arquitectura. De esta forma: (i) el código de reconfiguración se mantiene aislado del código funcional, (ii) las políticas de reconfiguración se mantienen separadas de los servicios que las llevan a cabo, y (iii) la reconfiguración se describe a un alto nivel de abstracción, en términos de la arquitectura que debe configurarse.

Esta propuesta se ha aplicado a un robot agrícola para describir, de modo independiente de tecnología, cómo el subsistema de visión debe recuperarse ante el fallo de uno de sus componentes y reemplazarlo en tiempo de ejecución por otro componente alternativo. El hecho de usar la reconfiguración dinámica en lugar de técnicas de redundancia permite no sólo reducir la complejidad de la arquitectura del robot, sino que además le dota de mecanismos para poder adaptarse a cualquier cambio en tiempo de ejecución: por ejemplo, reorganizar su arquitectura para que, en condiciones de baja energía, utilizar menos componentes o replicar aquellos que están siendo utilizados intensivamente.

Sin embargo, en un sistema robótico puede haber sistemas críticos cuya tolerancia fallos deba ser necesariamente implementada a través de la redundancia. Uno de ellos es el propio sistema de reconfiguración: como puede fallar, también necesita ser replicado. El otro caso son aquellos subsistemas en los que el tiempo de respuesta es crítico: la reconfiguración necesita un tiempo mínimo para aplicar las políticas de recuperación, mientras que con la replicación la respuesta es inmediata, pues existe un conjunto de réplicas que ya estaban proporcionando la misma funcionalidad del subsistema que ha fallado. El análisis de esta problemática se deja para trabajos posteriores.

Agradecimientos

Este trabajo ha sido financiado por el CICYT (Comisión Interministerial de Ciencia y Tecnología) proyecto MULTIPLE (TIN2009-13838).

Referencias

- [1] Aßmann, U. Invasive Software Composition. Springer, 2003.
- [2] Batista, T., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-Based Systems. 2nd European Workshop on Software Architectures (EWSA'05). LNCS, vol. 3527. Springer, 2005.
- [3] Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. First Workshop on Self-Managed Systems (WOSS'04).
- [4] Buckley, J., Mens, T., Zenger, M., et al.: Towards a taxonomy of software change. Journal on Software Maintenance and Evolution 17(5). Wiley, 2005.
- [5] Bures, T., Hnetyinka P., Plasil F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. 4th Int. Conf. on Software Eng. Research, Management and Applications (SERA'06). Seattle, WA, 2006.
- [6] Cámara, J., Canal, C., Cubo, J., Murillo, J.M.: Enabling Adaptivity in User Interfaces. 1st European Conf. on Software Architecture (ECSA'07). LNCS, vol. 4758. Springer, 2007.
- [7] Canal, C., Pimentel, E., Troya, J.M.: Specification and Refinement of Dynamic Software Architectures. Working IFIP Conf. on Software Architecture (WICSA'99). San Antonio, Texas, USA, 1999.
- [8] Costa, C., Pérez, J., Carsí, J.A.: Dynamic Adaptation of Aspect-Oriented Components. 10th Int. Symp. on Component-Based Software Engineering (CBSE'07). LNCS, vol. 4608. Springer, 2007.
- [9] Costa-Soria C., Hervás-Muñoz D., Pérez J., Carsí J.A. A Reflective Approach for Supporting the Dynamic Evolution of Component Types. 14th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'09). Potsdam, Germany, 2009.
- [10] Costa-Soria C., Pérez J., Carsí J.A. An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures. 2nd Workshop on Autonomic and SELF-adaptive Systems (WASELF'09). San Sebastián, Spain, 2009.
- [11] Cuesta, C.E., Fuente, P.d.l., Barrio-Solázano, M.: Dynamic Coordination

- Architecture through the use of Reflection. ACM Symp. on Applied Computing (SAC'01). USA, 2001.
- [12] Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards Architecture-Based Self-Healing Systems. First Workshop on Self-Healing Systems (WOSS'02), 2002.
 - [13] de Lemos, R.: Architectural reconfiguration using coordinated atomic actions. Intern. Workshop on Self-Adaptation and Self-Managing Systems (SEAMS'06). ACM, 2006
 - [14] Douglass, B.P.: Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley Object Technology, 1999.
 - [15] Endler, M., & Wei, J.: Programming Generic Dynamic Reconfigurations for Distributed Applications. 1st International Workshop on Configurable Distributed Systems, 1992.
 - [16] Fernández, C., Iborra, A., Álvarez, B., et al.: Ship Shape in Europe: Co-operative Robots in the Ship Repair Industry. Robotics and Automation Magazine, 12(3). IEEE, 2005.
 - [17] Garlan, D., Cheng, S., Huang, S., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer, 37. IEEE 2004.
 - [18] German-Rivera, J., et al.: An Architectural Description of the Simplex Architecture, Tech. Report CMU/SEI-96-TR-006, 1996.
 - [19] 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.
 - [20] Heimerdinger, W.L., Weinstock, C.B.: A Conceptual framework for System Fault Tolerance. Tech. Report CMU/SEI-92-TR-033, 1992.
 - [21] Iborra, A., Pastor, J.A., Álvarez, B., Fernández, C., Fernández, J.M.: Robots in Radioactive Environments. Robotics and Automation Magazine, 10(4), 2003.
 - [22] Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect-Oriented Programming. In: ECOOP'97. LNCS, vol. 1241. Springer, 1997.
 - [23] Kramer, J., Magee, J.: Dynamic configuration for distributed systems. IEEE Trans. Software Engineering, 11(4), 1985.
 - [24] McKinley, P.K., Sadjadi, S.M., Kasten, E.P., and Cheng, B.H.C.: Composing Adaptive Software. Computer, 37(7):56-64. IEEE, 2004.
 - [25] Morrison, R., Balasubramaniam, D., Kirby, G., et al: A Framework for Supporting Dynamic Systems Co-Evolution. Autom. Software Eng. 14(3). Springer, 2007.
 - [26] Pérez, J, Ali, N, Costa, C, Carsí, J.A., Ramos, I.: Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. 3rd Int. Conference on .NET Technologies. Pilsen, Czech Republic, 2005.
 - [27] Pérez, J. PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Universidad Politécnica de Valencia, 2006.
 - [28] Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. 9th Symp. on Component-Based Software Engineering (CBSE06). LNCS, vol. 4063. Springer, 2006.
 - [29] Perry, D. E., & Wolf, A. L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Eng. Notes, 17(4), 1992
 - [30] Rasche, A., Polze, A.: Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET. 6th Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'03), 2003.
 - [31] Sadjadi, SM, McKinley, PK, Cheng, BHC, Kurt, RE: TRAP/J: Transparent generation of adaptable java programs. Int. Symp. on Distributed Objects and Applications (DOA'04). Agia, Cyprus, 2004.
 - [32] Yurcik, W., Doss, D.: Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration. IEEE Software, 18(4), 2001.

Modelos Arquitectónicos Orientados a Servicios: Definición y Desarrollo Formal sobre un Caso de Estudio

Marcos López-Sanz
Kybele, Dep. LSI II
Universidad Rey Juan Carlos
Móstoles 28933 Madrid
marcos.lopez@urjc.es

Carlos E. Cuesta
Kybele, Dep. LSI II
Universidad Rey Juan Carlos
Móstoles 28933 Madrid
carlos.cuesta@urjc.es

Esperanza Marcos
Kybele, Dep. LSI II
Universidad Rey Juan Carlos
Móstoles 28933 Madrid
esperanza.marcos@urjc.es

Resumen

En los últimos años el paradigma orientado a servicios y la propuesta MDA se han postulado como los enfoques más adecuados para abordar los retos tecnológicos a los que se enfrenta la empresa actual. A pesar de las múltiples ventajas de aplicar la propuesta MDA al desarrollo orientado a servicios, aún existen algunas carencias que necesitan ser solucionadas. A nivel PIM, por ejemplo, no existe la posibilidad de tener una versión ejecutable del sistema ya que este nivel comprende únicamente modelos independientes de la tecnología. Para resolver esta deficiencia, es posible centrarse en la vista de la arquitectura y en su capacidad para dar una representación completa tanto de la estructura como del comportamiento del sistema en desarrollo. Por otra parte, lenguajes de descripción de arquitecturas como π -ADL permiten especificar arquitecturas que posteriormente pueden ser ejecutadas. Teniendo en cuenta este aspecto, en este artículo se presenta la definición, y posterior desarrollo formal utilizando π -ADL, de un modelo de arquitecturas orientado a servicios a nivel PIM aplicado a un caso de estudio real.

Palabras clave.

Modelado de Arquitecturas Orientadas a Servicios (SOA), Lenguajes de Descripción de Arquitecturas (ADL), Arquitecturas Dirigidas por Modelos (MDA), π -ADL

1. Introducción

La orientación a servicios surgió hace unos años como respuesta a las necesidades de integración de las empresas que necesitaban solucionar el

problema de la distribución de los recursos. El paradigma orientado a servicios (SOC, *Service Oriented Computing*) [15] ha sufrido una rápida evolución debido, por un lado, al auge de Internet que ha favorecido el interés en desarrollar lenguajes y herramientas tecnológicas enfocadas a este tipo de aproximaciones y, por otro, por la ampliación del ámbito en el que están siendo utilizados los principios de la orientación a servicios, dejando de ser una solución tecnológica y convirtiéndose en una forma de afrontar el desarrollo de sistemas siguiendo dichos principios.

Sin embargo, a pesar del gran avance de SOC y de la aparición de múltiples iniciativas de desarrollo basadas en este paradigma, aún existen muchas cuestiones no resueltas cuando se piensa en metodologías o estrategias integrales de desarrollo basadas en servicios. Uno de los ámbitos en los que aún existen carencias y que aún tienen que madurar es el de las metodologías de desarrollo.

De manera paralela al avance de SOC, el desarrollo dirigido por modelos (MDD, *Model Driven Development*) [13] se ha posicionado como uno de los enfoques de desarrollo que mejor se adapta a las necesidades actuales de desarrollo. MDD utiliza el concepto de modelo como artefacto central para el desarrollo de aplicaciones, propugnando, como enfoque de ingeniería software, la evolución a lo largo del ciclo de desarrollo mediante transformaciones automáticas entre modelos, y por lo tanto, la generación automática de modelos. De entre las propuestas actuales de enfoques dirigidos por modelos destacan propuestas como *Software Factories* de Microsoft [6] o MDA de OMG [13]. Esta última se caracteriza por definir una arquitectura de modelos dividida en diferentes niveles de abstracción: CIM, PIM y PSM.

En trabajos anteriores se ha argumentado que una de las deficiencias de la propuesta de MDA es la falta de especificación del rol de la arquitectura del sistema dentro de la arquitectura de modelos propuesta. En [10] se concluyó que la arquitectura debe formar parte del proceso de desarrollo basado en MDA y que deben definirse modelos arquitectónicos que evolucionen y dirijan el proceso de desarrollo. En [8] se presentó un metamodelo que recoge los elementos que han de utilizarse en la definición de modelos arquitectónicos orientados a servicios dentro de una metodología llamada MIDAS [2].

Sin embargo, a pesar de las ventajas que proporciona usar un enfoque dirigido por modelos para desarrollar soluciones orientadas a servicios, en especial cuando se pretende que la arquitectura sea el centro del proceso [1][5], aún existen carencias que hacen que este tipo de metodologías no puedan considerarse completas. Estas deficiencias incluyen aspectos tales como una definición aún difusa de las relaciones entre los modelos de los distintos niveles de MDA, o las relaciones de dichos modelos con los de la arquitectura. Otra de las principales críticas que se le hace al enfoque MDA es que habitualmente resulta complicado disponer un prototipo o versión ejecutable en etapas tempranas del desarrollo, que es de gran utilidad en sistemas orientados a servicios.

Para solucionar este problema, este trabajo se centra en la definición formal de la arquitectura de servicios utilizando un lenguaje de descripción de arquitecturas (ADL, *Architecture Description Language*) que pueda solventar este problema. En concreto, se ha escogido representar los modelos arquitectónicos definidos en anteriores trabajos utilizando π -ADL [14]. π -ADL es un lenguaje cuyas propiedades se adecúan a las características y principios de la orientación a servicios, desde el soporte para la especificación de aspectos de dinamismo a través del envío de conexiones a través de conexiones hasta la disponibilidad de un entorno de ejecución de arquitecturas descritas utilizando este lenguaje (que es precisamente lo que se busca).

Este artículo se centrará, por lo tanto, en describir cómo puede utilizarse el lenguaje π -ADL para describir los modelos arquitectónicos orientados a servicios definidos para MDA e ilustrarlo a través de la utilización de un caso de estudio real: la creación de una pasarela para el

envío masivo de mensajes de texto (SMS, *Short Message Service*) entre clientes y proveedores especializados.

La estructura del artículo es la siguiente: el apartado 2 recoge el contexto en el que se estudia la obtención de versiones ejecutables de arquitecturas orientadas a servicios; la metodología MIDAS y el lenguaje π -ADL; el apartado 3 describe los modelos SOA de nivel PIM; el apartado 4 presenta un caso de estudio en el que se aplican los conceptos previamente explicados y, finalmente, el apartado 5 recoge las principales conclusiones de este trabajo así como algunas de las posibles líneas de investigación futuras.

2. Contexto de este trabajo

2.1. Marco metodológico: MIDAS

La principal contribución de este artículo (la definición y posterior desarrollo formal de modelos arquitectónicos orientados a servicios) forma parte de un esfuerzo de investigación mayor: el refinamiento de MIDAS, un marco metodológico de desarrollo completo basado en los principios de MDA. MIDAS define una arquitectura de modelos que puede observarse desde 3 dimensiones ortogonales:

- **Dimensión Vertical:** la primera clasificación de modelos se realiza de acuerdo a la propuesta MDA de OMG y define tres niveles de abstracción: modelos independientes de computación (CIM), modelos independientes de la plataforma (PIM) y modelos dependientes de la plataforma (PSM). Esta dimensión permite progresar en el desarrollo de un sistema desde los conceptos asociados al dominio del problema, en el nivel CIM, hasta la representación del código final a través de modelos de nivel PSM.
- **Dimensión horizontal:** los modelos se organizan en función de los diferentes aspectos involucrados en el desarrollo del sistema. De este modo, se definen los aspectos de Contenido, Comportamiento e Interacción.
- **Dimensión transversal:** Existen ciertos aspectos que no afectan a un nivel de abstracción concreto o aspecto de desarrollo, sino que tienen influencia sobre todos ellos,

describiendo los elementos que deberán incluir e incluso decidir si deben o no crearse unos u otros modelos. En este caso, encontramos aspectos como la Arquitectura, considerado como el aspecto que guía el proceso de desarrollo, o la Semántica, que permite concretar los significados de los términos utilizados en cada uno de los demás modelos y procesos modelados.

Como se ha comentado, la arquitectura es un aspecto que afecta tanto al nivel PIM como al nivel PSM. El principal motivo para ello es que MIDAS sigue un enfoque ACMDA (*Architecture-Centric Model-Driven Architecture*), es decir, define un método para el desarrollo de sistemas de información basado en modelos y dirigido por la arquitectura. Consideramos que la arquitectura es el aspecto que guía el proceso de desarrollo puesto que permite especificar qué modelos, elementos dentro de los modelos o relaciones entre modelos deberían ser creadas durante el proceso de desarrollo de software.

Mediante la definición de una vista arquitectónica a nivel PIM aseguramos que no existe ningún tipo de restricción, ya sea tecnológica o de implementación, dentro del modelo. Además, se facilita la definición de diferentes modelos de nivel PSM para diferentes plataformas de implementación cuyas descripciones se obtienen de un mismo modelo de nivel PIM. Como contrapartida, al no disponer de una tecnología de implementación asociada a la semántica del modelo, a nivel PIM se presenta el problema de no disponer de una versión ejecutable del sistema.

2.2. El lenguaje π -ADL

π -ADL es un lenguaje de descripción de arquitecturas diseñado para definir arquitecturas software basado en la lógica tipada de primer orden de cálculo- π [11]. Una de las características diferenciadoras de π -ADL sobre otros ADL es la existencia de herramientas asociadas al lenguaje que permiten mayor facilidad en su uso y manejo. Entre las mismas destaca π -ADL.Net, el *toolset* que se ha utilizado para la compilación, simulación y validación de la especificación arquitectónica desarrollada para el caso de estudio central de este artículo.

En una especificación en π -ADL los elementos de primer nivel están representados por comportamientos y abstracciones. Cada definición de comportamiento (*behaviour*) se entiende como un hilo de ejecución independiente, de tal forma que la ejecución de la arquitectura creará tantos hilos de ejecución como comportamientos se hayan definido. Las abstracciones (*abstraction*) se consideran como plantillas de comportamiento reutilizables cuya funcionalidad puede ser utilizada tanto por comportamientos como por otras abstracciones. La diferencia entre un comportamiento y una abstracción se encuentra en que una abstracción define un argumento cuyo valor es necesario para completar la especificación de la abstracción.

El cuerpo de un elemento de tipo comportamiento o de tipo abstracción contiene declaraciones, operaciones de renombrado (asignación de variables), acciones sobre el recurso controlado (en el caso de los servicios) y acciones de envío y recepción sobre conexiones que conformarán la funcionalidad básica de la abstracción o comportamiento. Las conexiones (*connections*) ofrecen una funcionalidad análoga a los canales del cálculo- π : dos elementos especificados en la arquitectura se comunican de forma síncrona a través de conexiones. Las conexiones pueden comunicar indistintamente abstracciones y comportamientos. Los elementos de tipo conexión son tipados, de tal forma que pueden enviar y recibir cualquiera de los tipos definidos, incluso abstracciones, comportamientos u otras conexiones.

π -ADL soporta además la especificación de secuencias de acciones concurrentes tal y como se definen en cálculo- π . Permite especificar comportamientos tanto deterministas (*compose*) como no deterministas (utilizando la estructura *choose*). Con respecto al soporte para la repetición de procesos, π -ADL define el constructor *replicate* cuya semántica se basa en la repetición infinita de un bloque en hilos de ejecución independientes. Además, π -ADL permite utilizar estructuras de control, tanto de bifurcación como de repetición (bucles) en la especificación del comportamiento de cada uno de los elementos involucrados en la definición de la arquitectura.

Todas estas características hacen que π -ADL resulte apropiado para la especificación de sistemas dinámicos. Por ello, lo hemos escogido

para proporcionar una correspondencia formal del metamodelo arquitectónico orientado a servicios que se desarrolla a continuación.

3. Definición de modelos arquitectónicos orientados a servicios

En esta sección se explican los elementos del metamodelo utilizado para definir modelos arquitectónicos orientados a servicios de nivel PIM dentro de MIDAS. La Figura 1 describe el metamodelo de servicios que contiene las bases según las que se define de este modelo.

3.1. Proveedores de servicios

Las Arquitecturas Orientadas a Servicios (SOA, *Service Oriented Architectures*) se construyen sobre entidades independientes que ofrecen y gestionan servicios. SOA se concibió inicialmente como un paradigma que permitiera la integración de aplicaciones dentro del ámbito empresarial, por ello, la existencia de estos proveedores de servicios se justifica por la necesidad de proyectar, en el modelo arquitectónico, las organizaciones de negocio involucradas. Estos proveedores actúan de contenedores de servicios encargados de promocionar los servicios que contienen al exterior.

Los proveedores de servicios pueden clasificarse en dos grupos: proveedores internos de servicios (*innerProvider* en el metamodelo), que son entidades internas del sistema que se está modelando (normalmente la parte de la solución software que se está diseñando en detalle); y proveedores externos de servicios (*outerProvider*), que se identifican como entidades que contienen servicios que colaboran con el sistema para llevar a cabo una tarea específica de valor para el sistema pero que no está bajo su control o bien cuya estructura interna no se conoce o no es de interés para el desarrollo del modelo arquitectónico actual.

La relación entre dos proveedores de servicios se modela a través de una dependencia, que se

concibe como un contrato de negocio (*business Contract*) en el que se representa una forma concreta de comunicar entre sí a los servicios que pertenecen a estos proveedores.

3.2. Descripción y propiedades del servicio

Los servicios son entidades computacionales responsables de un recurso y que ofrecen funcionalidades asociadas a dicho recurso en la forma de operaciones. Aparte de estas operaciones, los servicios tienen una propiedad denominada *servid* que permite la identificación unívoca del servicio dentro de un entorno SOA y del modelo PIM. Todo servicio tiene que poder ser buscado, localizado y accedido y, por lo tanto, debe ser identificado. Otra de las razones que nos llevan a incluir un identificador dentro de la definición de un modelo a nivel PIM es que este modelo representa *instancias* de servicios (y no clases o tipos como se hace normalmente en aproximaciones orientadas a objetos).

Las operaciones de los servicios se entienden como funcionalidades atómicas que colaboran para construir una descripción conjunta del servicio. Además, representan la única forma de interactuar con un servicio puesto que conforman la interfaz del mismo. Los atributos de las operaciones incluyen el nombre, argumentos, valor devuelto y tipo de operación. Cabe resaltar la distinción entre diferentes tipos de operaciones dependiendo de su *sincronicidad*, que afecta directamente al tipo específico de servicio que se estará definiendo. En las operaciones asíncronas (*AsynchronousOp* en el metamodelo) el solicitante de la operación no espera por la respuesta a la operación o valor devuelto (si es que hay alguno); por el contrario, en operaciones síncronas (*SynchronousOp*), el solicitante deberá esperar por la respuesta o valor devuelto (que siempre existe). Como se comentará más adelante, cualquier servicio que implemente al menos una operación de tipo síncrono deberá ser modelado como servicio de interacción.

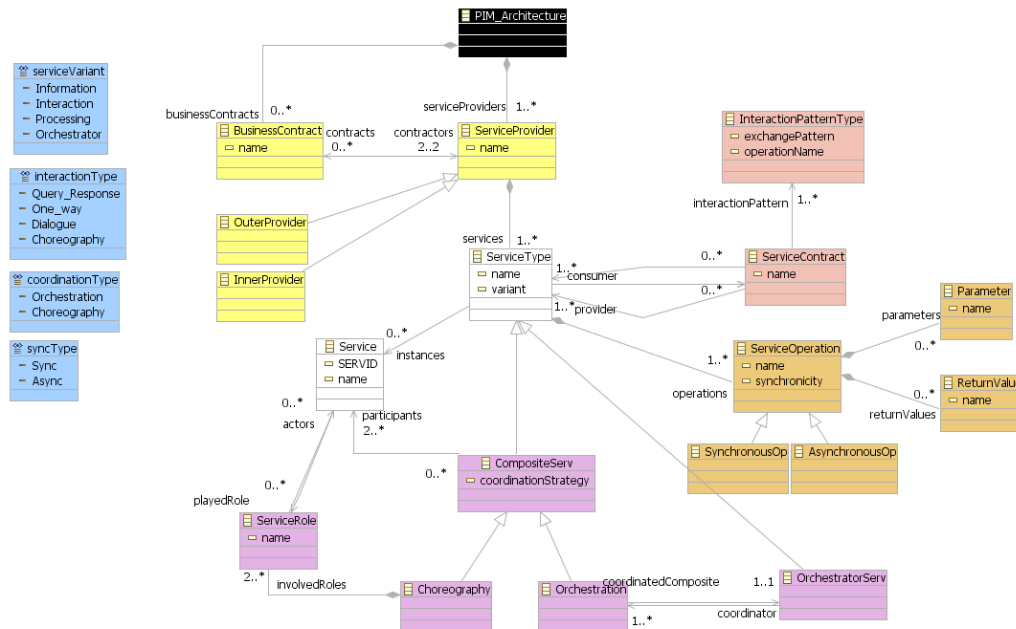


Figura 1. Metamodelo Arquitectónico de Servicios de Nivel PIM.

3.3. Relaciones entre servicios

Los servicios se relacionan, comunican e interactúan entre sí a través de *contratos* (*ServiceContract*). Las operaciones que participan en el contrato establecido entre los servicios pueden entenderse como los roles de cada uno de los servicios relacionados. En la descripción arquitectónica de los modelos de servicios, los contratos se definen como conectores, especificando relaciones punto a punto entre los servicios que “firman” dichos contratos. La propiedad principal de un contrato es el *patrón de intercambio de mensajes*, que describe el flujo de mensajes intercambiados entre los servicios relacionados. La descripción explícita del comportamiento debe ser respetada tanto por el solicitante como por el proveedor del servicio.

3.4. Tipos de servicios

El rol principal de los servicios en SOA es el de dar soporte a las funcionalidades ofrecidas por el

sistema y derivadas de los requisitos obtenidos a partir de los clientes y la información recogida en los modelos de los niveles superiores. Nuestra concepción del término *servicio* (a nivel PIM) se corresponde con la especificada en el modelo de referencia de OASIS [12] para servicios:

Un servicio es un mecanismo que permite el acceso a un conjunto de una o más capacidades donde dicho acceso es ofrecido a través de una interfaz bien conocida y se lleva a cabo de acuerdo a las restricciones y políticas especificadas en la descripción del servicio.

Un aspecto especialmente significativo, y que tiene una presencia explícita en el metamodelo, es el referido a la *atomicidad* del servicio. Los servicios *simples* representan un comportamiento del sistema que no implica la invocación de ningún otro servicio modelado en el sistema. Su funcionalidad es satisfecha íntegramente mediante las operaciones que definen. Por el contrario, los servicios *compuestos* (dependientes de la clase *CompositeServ* en el metamodelo) se definen por tener, al menos, una operación en la que, debido a su complejidad o naturaleza, es necesario acceder a las funcionalidades ofrecidas por otros servicios.

Esta naturaleza compuesta obliga a que se produzca un proceso de *coordinación* entre los diferentes servicios que forman parte de la composición. La coordinación entre servicios puede conseguirse siguiendo dos enfoques claramente diferenciados: mediante el uso de *coreografías* o a través de la *orquestración* de servicios. La orquestración se basa en la definición de un servicio orquestador (*OrchestratorServ*) cuya finalidad es la de llevar a cabo un flujo de ejecución definido como una operación compleja, formada por un conjunto de invocaciones a las operaciones de los servicios que forman parte del compuesto. Por otro lado, las coreografías de servicios constituyen un entorno de coordinación en el que cada servicio es autónomo, y en el que ninguno de los participantes tiene más control que otro; pero en el que todos se sincronizan de acuerdo con un plan preestablecido.

4. Especificación en π -ADL de modelos SOA: Aplicación a un caso de estudio

Una vez expuestos los conceptos incluidos en la definición de modelos arquitectónicos de servicios, en este apartado se expone la especificación de la arquitectura de un caso de estudio utilizando π -ADL. El caso de estudio seleccionado consiste en la emulación de una pasarela de envío de mensajes SMS basada en el protocolo SMPP [16] (*Short Message Peer-to-Peer Protocol*) mediante servicios. SMPP es un protocolo industrial de telecomunicaciones que define el intercambio de mensajes de texto (SMS) entre entidades homogéneas (*peers*) denominadas SMSC (*Short Message Service Centers*). A través de una interfaz determinada (por ejemplo, Web) un usuario es capaz de enviar mensajes de texto a múltiples destinatarios.

Aunque un sistema SMPP está compuesto de múltiples y variados elementos, en nuestro caso sólo nos centraremos en los siguientes bloques y funcionalidades:

- **Subsistema de Recepción:** su principal objetivo es recibir peticiones de envío de SMS directamente del usuario. Contiene un único servicio (*ReceptionService*) que ofrece varias operaciones útiles para el usuario tales como *sendSMS*, que permiten indicar al sistema su deseo de enviar SMS a un conjunto de destinatarios almacenados previamente.

- **Subsistema de Almacenamiento:** se trata de un proveedor de servicios que almacena la información relativa a los clientes (potenciales usuarios del sistema de envío de SMS) y los textos de los mensajes SMS así como información relacionada con éstos. Contiene dos servicios:

- **Secure Data Service:** realiza operaciones que requieren una conexión segura o la encriptación de los datos. Las operaciones que ofrece incluyen *authenticate* (utilizada para verificar las credenciales del cliente que desea utilizar la pasarela de envío de SMS) y *updateCredit* (utilizada para actualizar la información económica asociada al cliente en función de la operación solicitada al sistema).

- **SMS Manager Service:** servicio encargado de gestionar y almacenar toda la información relativa a los mensajes de texto SMS (texto, estado, duración, etc.) y de crear listados e informes asociados a un cliente específico.

- **Subsistema de Procesamiento:** este subsistema se encarga de recuperar, procesar y enviar los mensajes SMS junto con la información relacionada a los servidores de envío de SMS especializados. Está compuesto por dos servicios:

- **SMS Sender Service:** recupera los textos de los mensajes SMS del subsistema de almacenamiento y los envía a los centros especializados de envío de mensajes.

- **Directory Service:** la principal tarea que lleva a cabo este servicio es devolver el identificador de servicio que se corresponde con el SMSC que debe ser utilizado para enviar un determinado SMS a un destinatario concreto.

- **Massive SMS System:** esta entidad representa los centros de envío de SMS especializados que son capaces de enviar el mismo mensaje SMS a un número predeterminado de destinatarios. Su funcionalidad está encapsulada en diversos servicios que recibe tanto los mensajes SMS como la lista de destinatarios (*SMSCenter*).

En la Figura 2 se indica la estructura de todos estos servicios mediante un modelo que sigue la estructura del metamodelo descrito anteriormente.

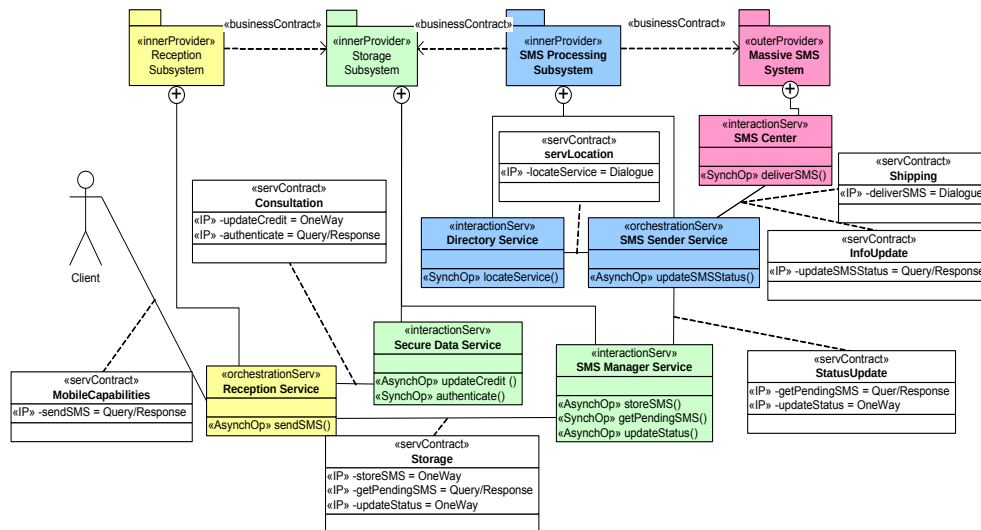


Figura 2. Modelo Arquitectónico (a Nivel PIM) del Caso de Estudio

A continuación se presenta parte de la especificación del modelo arquitectónico con π -ADL, poniendo especial énfasis en los aspectos de este lenguaje que permiten definir la solución concreta al sistema, así como en la explicación de cómo las estructuras y principios de π -ADL se adaptan a nuestra visión de arquitectura orientada a servicios a nivel PIM.

Representación de un servicio y sus operaciones. Los servicios representan entidades computacionales que realizan un comportamiento específico dentro de la arquitectura del sistema y, por lo tanto, se especifican en π -ADL a través de *abstracciones*.

Cada abstracción de servicio define sus propios canales de comunicación a través de conexiones de entrada y salida. Los datos obtenidos y enviados por estas conexiones se componen de la descripción de la operación y los datos asociados a dicha operación. En función de la operación solicitada, la abstracción de servicio transferirá el control de la ejecución a una u otra abstracción de operación.

Las operaciones, de forma equivalente, se especifican también mediante abstracciones, puesto que encapsulan parte de la funcionalidad ofrecida por los servicios. Al igual que cualquier otra abstracción utilizada en la especificación de la arquitectura de servicios, las abstracciones de

operación son procesos que recibirán la información que necesitan para realizar su cometido a través de conexiones, devolviendo una respuesta en el caso que sea necesario.

```

value ReceptionService is abstraction
() {
  outConn:Connection[view[op:String,
                           data: any]];
  output:view[op:String,data:any];
  inConn:Connection[view[op: String,
                           data: any]];
  input:view[op: String, data: any];
  if (input::operation=="sendSMS")do{
    via sendSMS send input::data
    where {resultConn renames
           resultConn};
    via resultConn receive result;
    compose {
      via outConn send result;
      and
      done;   }}

```

Listado 3.1. Especificación de *ReceptionService*.

En π -ADL, la comunicación realizada a través de conexiones se realiza de forma síncrona. Esto significa que la comunicación mediante operaciones es siempre síncrona. Por lo tanto, la semántica asociada en nuestro metamodelo a las operaciones asíncronas se pierde desde el

momento en que una abstracción permanece bloqueada en una operación de envío (*send* a través de una conexión) hasta que cualquier otra abstracción o comportamiento de la arquitectura realice una operación de lectura sobre ese mismo canal (*receive* sobre la misma conexión). Con el fin de poder emular el comportamiento de las operaciones asíncronas, la especificación puede ser encapsulada dentro de uno de los sub-bloques de un operador *compose*, dividiendo el flujo de ejecución en dos, uno que se bloquea y otro que continúa con la operación.

Representación de contratos. Como se ha mencionado anteriormente, los servicios se relacionan y comunican a través de contratos. Dentro de la arquitectura, estos contratos se consideran *conectores activos*. Tienen la responsabilidad de permitir el intercambio de mensajes entre servicios, de acuerdo a un determinado patrón, descrito mediante una máquina de estados. En π -ADL los contratos se representarán también como abstracciones con un comportamiento determinado.

En un entorno *estático* de servicios, en el que los contratos se establecen en tiempo de diseño, toda la información que necesita un contrato para cumplir con su cometido (patrón de intercambio de mensajes, servicios que firman el contrato y conexiones por las que comunicarse con ellos) se define e inicializa dentro de la abstracción que contiene el contrato. Sin embargo, en entornos *dinámicos*, al menos parte de la información que necesita un contrato se debe enviar a través de un canal que se abre a la vez que se inicia el hilo de ejecución de la abstracción que lo contiene.

```
...
if (state::via_servid == "S") do{
  compose {
    via outConnectionS send inData;
    and
    via dynamic
      (input::ServConnGroup(0)::servid)
      send Void
    where {
      outConnectionS renames inConn,
      inConnectionS renames outConn;
    }
  } else do{
    via outConnectionC send inData;
  }
}
...
```

Listado 3.2. Fragmento del contrato *Shipping*

El listado 3.2 muestra parte del análisis de un estado de la ejecución del patrón de intercambio de mensajes dentro de la especificación de un contrato con π -ADL. Se puede observar cómo se usa el operador *compose* para separar el envío de los datos de la ejecución de la abstracción.

Debido a la naturaleza dinámica de las arquitecturas de servicios, las abstracciones de contrato pueden (y deben) ser reutilizadas, puesto que las instancias de los servicios que comunican pueden variar a lo largo del ciclo de vida del sistema. Para alcanzar este comportamiento, los contratos (sus abstracciones) deben ser capaces de instanciar dinámicamente el canal que han de utilizar para enviar o recibir los datos en cada momento (*connections*). Para dar soporte a este aspecto, π -ADL define un operador especial (*dynamic* <nombre de la conexión>). Este operador permite instanciar un nombre obtenido a través de una conexión. De esta forma, π -ADL implementa parte de la lógica necesaria para la transferencia de conexiones a través de conexiones, característica del cálculo- π .

Representación del dinamismo. En el caso de estudio presentado, para enviar los mensajes SMS almacenados por el sistema de almacenamiento, es necesario conocer qué servicio *SMSCenter* específico debe utilizarse. Para ello, es esencial disponer de un servicio especializado (*Directory Service*) que atiende a las peticiones de los otros servicios, que necesitan utilizar las funcionalidades ofrecidas por servicios inicialmente desconocidos. Estas capacidades establecen las bases de un entorno *dinámico* en el que se necesita utilizar un canal de comunicación que no existía en tiempo de diseño, y que debe ser creado en tiempo de ejecución.

En un entorno orientado a servicios, el dinamismo puede emerger de dos situaciones: aquella en la que es necesario crear un contrato nuevo entre servicios, o aquella en la que se añade a la arquitectura del sistema un nuevo servicio (o tipo de servicio) que no existía previamente. En estos casos es obligatorio disponer de un servicio especial, como el Servicio de Directorio ya mencionado. Junto a éste, se dispone del soporte para utilizar las cuatro operaciones básicas de cualquier sistema dinámico [9]: crear (*create*), destruir (*destroy*), enlazar (*link*) y deslazar (*unlink*). En nuestro caso, y en este ejemplo, únicamente nos centramos en la definición de nuevas abstracciones de contrato.

```

value locateService is abstraction (
  locationConn:view[inConn:Connection
  [view[operation:String,data:any]],
  input:view[op: String, data : any],
  connSMSCenter: Connection[view[op:
    String, data : any]]){
  requestData =
    locationConn::input::data;
  unobservable; //analizar petición
  compose {
  // desc. del intercambio de msjes.
  contractStartInfo::MEP = "";
  contractStartInfo::connServ1 =
    locationConn::connSMSCenter;
  contractStartInfo::connServ2 =
    shContractClientConn;
  // preparar y crear conector:
via Shipping send contractStartInfo;
  and
  clientDataResponse::contractConn =
    shContractClientConn;
  // info operaciones y semántica
  clientDataResponse::servMetaInfo="";
  clientResponse::data =
  clientDataResponse;
  clientResponse::op = "locService";
  via locationConn::inConn send
  clientResponse; and done; }}

```

Listado 3.3. Especificación de la operación *locateService*

Partiendo de la premisa de que el Servicio de Directorio conoce los *servid* de cualquier servicio especializado de envío de mensajes solicitado, las tareas que realizará este servicio incluyen: la creación del contrato que se necesita para que un servicio pueda comunicarse con el *SMSCenter* y la creación de la conexión que será utilizada por los servicios que firman dicho contrato. La información devuelta por el Servicio de Directorio comprende tanto la conexión con el contrato que permite la comunicación con el *SMSCenter*. El nuevo contrato creado recibirá, como información inicial, el *servid* del Servicio especializado de envío de mensajes que conectará y el canal de comunicación que se utilizará para transferirle los datos que sean necesarios.

La consulta específica realizada al Servicio de Directorio puede ser de tres tipos: consulta por un *servid* concreto, consulta “semántica”, en la que se indica la funcionalidad que se desea que cumpla el servicio buscado o consulta por un formato de contrato específico. En nuestro caso nos hemos centrado únicamente en la información que

necesita cada uno de los elementos involucrados en el proceso dinámico por lo que la definición de cada uno de estos tres tipos de consultas se ha dejado para trabajos de investigación posteriores.

En nuestro caso de estudio uno de estos servicios es el Servicio de envío de mensajes SMS que coordina el acceso al subsistema de almacenamiento (utilizando la funcionalidad ofrecida por el Servicio de Gestión de SMS), la recuperación de la información necesaria para utilizar un Servicio *SMSCenter* concreto (consultando para ello al Servicio de Directorio) y finalmente el envío del mensaje SMS concreto al Servicio *SMSCenter* para completar la funcionalidad requerida por el cliente del sistema.

5. Conclusiones y trabajos futuros

En los últimos años, la combinación del paradigma orientado a servicios y del enfoque de desarrollo propuesto por OMG a través de MDA ha demostrado ser un gran avance en lo referente al desarrollo de soluciones flexibles centradas en aspectos tales como la integración de recursos distribuidos. Por otra parte, la arquitectura se considera el elemento adecuado para guiar el desarrollo del sistema. En este sentido, han aparecido múltiples propuestas metodológicas basadas en los principios de SOA y MDA y centradas en la arquitectura. Una de estas metodologías es MIDAS, que define metamodelos y perfiles UML que permiten el modelado de arquitecturas orientadas a servicios dentro de un proceso de desarrollo dirigido por modelos basado en MDA. Sin embargo, aún existen algunas carencias que han de ser resueltas.

En este trabajo, y con la intención de resolver la ausencia de versiones ejecutables en etapas tempranas de los desarrollos basados en MDA, hemos propuesto la definición formal de la arquitectura de un sistema orientado a servicios utilizando π -ADL, un lenguaje de descripción de arquitecturas ejecutable basado en cálculo- π . Se ha escogido este lenguaje puesto que permite representar tanto arquitecturas dinámicas como la evolución de las mismas, además de contar con un conjunto de herramientas de compilación y desarrollo asociadas a dicho lenguaje. Asimismo, mediante el uso de una representación formal de la arquitectura, es posible utilizar formalismos matemáticos que permiten comprobar los modelos

UML creados para cada uno de los niveles de abstracción definidos en MIDAS.

Existen múltiples líneas de investigación que surgen a partir del trabajo presentado. Una de ellas es, teniendo en cuenta tanto la notación UML existente asociada al lenguaje π -ADL, la definición de reglas de transformación entre el metamodelo de la arquitectura de servicios de nivel PIM de MIDAS y el metamodelo de π -ADL con el fin de poder automatizar la definición formal del modelo PIM de SOA en π -ADL. Otra línea de investigación abierta se centra en la definición de los modelos arquitectónicos de servicios de nivel PSM así como la influencia que tiene el haber escogido π -ADL, como lenguaje de descripción de la arquitectura, en el momento de dar una representación formal de arquitecturas basadas en tecnologías de servicios concretas.

6. Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación dentro de los proyectos MODEL-CAOS (TIN2008-03582), y Tecnologías del Acuerdo (AT, CONSOLIDER CSD2007-0022).

Referencias

- [1] Broy, M. Model Driven, Architecture-Centric Modeling in Software Development. In Proceedings of 9th Intl. Conf. in Engineering Complex Computer Systems (ICECCS'04), pp. 3-12, IEEE Computer Society, April 2004.
- [2] Cáceres P., Marcos E. and Vela B. A MDA-Based Approach for Web Information System Development. Workshop in Software Model Engineering (WISME 2003).
- [3] De Castro V., López-Sanz M., Marcos E. Business Process Development based on Web Services: A Web Information System for Medical Images Management and Processing. Proceedings of IEEE International Conference on Web Services. IEEE Computer Society, 2006, pp. 807-814.
- [4] De Castro, V., Marcos, E., López-Sanz, M., A Model Driven Method for Service Composition Modeling: A Case Study. International Journal of Web Engineering and Technology, Vol. 2(4), 2006. Eds: Inderscience Enterprise Ltd, pp. 335-353.
- [5] Gomaa, H. Architecture-Centric Evolution in Software Product Lines. ECOOP'2005 Workshop on Architecture-Centric Evolution (ACE'2005), Glasgow, July 2005.
- [6] Greenfield J., Short K., Cook S., Kent S. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, 2004.
- [7] IEEE AWG. IEEE RP-1471-2000: Recommended Practice for Architectural Description for Software-Intensive Systems. IEEE Computer Society Press, 2000.
- [8] López-Sanz M., Acuña C. J., Cuesta C. E., Marcos E. Defining Service-Oriented Software Architecture Models for a MDA-based Development Process at the PIM level. In Proc of WICSA 2008. pp.309-312, IEEE Computer Society Press, Vancouver, 2008.
- [9] Magee J., Kramer J., and Sloman M. Constructing Distributed Systems in Conic. IEEE Transactions on Software Engineering, 1989. 15(6): pp. 663-675.
- [10] Marcos E., Acuña C. J., Cuesta C. E. Integrating Software Architecture into a MDA Framework. In Proc. of EWSA 2006, pp: 127-143. Nantes, France.
- [11] Milner R. The Polyadic π -Calculus: A Tutorial. Logic and Algebra of Specification, Springer-Verlag, 1993
- [12] OASIS. Reference Model for Service Oriented Architecture. Committee draft 1.0. Retrieved Feb. 2007, from: <http://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf>
- [13] OMG. Model Driven Architecture. Eds.: Miller, J., Mukerji, J. Document No. ormsc/2001-07-01. Retrieved May 2006, from <http://www.omg.com/mda>
- [14] 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, No. 3, May 2004.
- [15] Papazoglou M. P. Service-Oriented Computing: Concepts, Characteristics and Directions. In Proc. of WISE'03. pp. 3-12. Roma, Italy.
- [16] SMPP Forum. SMPP v5.0 Specification. Retrieved Sept. 2007 from <http://www.smsforum.net/>

Marco de trabajo orientado a aspectos para el diseño de arquitecturas software adaptables dinámicamente¹

Amparo Navasa, Miguel Angel Perez-Toledano, Juan Manuel Murillo

[amparonm, toledano, juanmamu]@unex.es

Quercus Software Engineering Group

Departamento de Ingeniería de Sistemas Informáticos y Telemáticos
Universidad de Extremadura

Resumen

Al evolucionar los sistemas software son necesarias herramientas que faciliten su adaptación al cambio. Dos técnicas de la ingeniería del software permiten reducir la complejidad de estos sistemas y mejorar su mantenimiento: la arquitectura del software, que facilita la definición estructural de grandes sistemas; y el desarrollo de sistemas orientado a aspectos que permite la incorporación o sustitución de requisitos de una manera transparente y no intrusiva, facilitando la adaptación dinámica del sistema ante cambios en los requisitos. La combinación de ambas técnicas da lugar al diseño arquitectónico orientado a aspectos. En este sentido, para especificar el diseño de sistemas orientados a aspectos, en este artículo se describe *AOSA Space*, un marco de trabajo que permite realizar adaptaciones en los sistemas de manera sencilla. En él se consideran los cambios desde un punto de vista estructural, siguiendo el principio de la separación de aspectos. En este artículo se muestran, de un modo general, las características del marco de trabajo desarrollado que consta de un modelo de desarrollo, una metodología de trabajo, un lenguaje de descripción arquitectónica y un juego de herramientas.

1. Introducción

La complejidad de los sistemas software actuales y la evolución constante de los sistemas para adaptarlos a nuevas circunstancias o para incluir nuevos servicios, son razones por las que la ingeniería del software debe disponer de técnicas,

herramientas y modelos que le permitan afrontar estos retos.

La Arquitectura del Software (AS), el Desarrollo Software Orientado a Aspectos (DSOA) y el Desarrollo Software Basado en Componentes (DSBC) han mostrado ser herramientas útiles para resolver estas cuestiones.

Este trabajo trata conjuntamente los conceptos de AS y de DSOA, dentro del marco del DSBC, pues ambas aproximaciones facilitan la gestión de la complejidad de los sistemas y su adaptación a unos requisitos cambiantes. La conveniencia de definir un diseño arquitectónico orientado a aspectos surge cuando se observa que la complejidad de las dependencias entre los componentes arquitectónicos da lugar a sistemas difíciles de hacer evolucionar. Este problema es más evidente cuando se pretenden integrar nuevas funcionalidades en sistemas ya construidos debido a las dependencias transversales entre los componentes del sistema.

Para resolver estas cuestiones se han considerado dos estrategias de trabajo. Por una parte, la definición de modelos que faciliten la construcción de sistemas complejos y mejoren su adaptación a los cambios, de modo que los sistemas sean más “modularizables” y “mantenibles”. Por otra parte, la definición de Lenguajes de Descripción Arquitectónica (LDA) que permiten representar formalmente el diseño de los sistemas para asegurar la corrección de la arquitectura que describen.

Este artículo presenta un marco de trabajo que permite el desarrollo de sistemas complejos y su evolución mediante la integración de aspectos en la descripción arquitectónica de un sistema. Consta de los siguientes elementos:

¹ Trabajo parcialmente financiado por el proyecto TIN2008-02985 y el Fondo Europeo de Desarrollo Regional (FEDER)

- i) La definición de un modelo que integre DSOA y DSBC en la fase de diseño arquitectónico.
- ii) La definición de un LDA orientado a aspectos, para dotar de un soporte formal a la arquitectura obtenida tras la aplicación del modelo.
- iii) Una metodología de trabajo para guiar el desarrollo.

La arquitectura del software es una actividad del diseño que ayuda a los ingenieros de software a reducir la complejidad de los sistemas y su desarrollo. Uno de los mecanismos en los que se apoya es en la definición de los LDA. No obstante, estos no representan adecuadamente el problema de modelar ciertas propiedades transversales del sistema. El marco de trabajo que se presenta proporciona una solución a esta cuestión desde un punto de vista arquitectónico. Para llevar a cabo esta tarea fue necesario considerar las características del DSOA.

El DSOA es un paradigma que propone técnicas para “modularizar” las propiedades transversales de los sistemas a lo largo del ciclo de vida. Dado que el DSOA permite modelar como elementos software este tipo de propiedades, los sistemas se pueden adaptar más fácilmente a los cambios, frente a la utilización de paradigmas tradicionales, en los que el código transversal queda disperso. Así, el DSOA mejora la calidad de los sistemas haciendo los diseños más claros y potenciando además la reutilización del software. Este paradigma se basa en el principio de separación de aspectos y proporciona técnicas para la identificación y extracción de aspectos, y su subsiguiente composición. Dentro de esta aproximación, parece lógico utilizar LDA para formalizar la arquitectura de los sistemas desarrollados bajo este paradigma. Sin embargo, es necesario que estos LDA sean adecuados a las peculiaridades del DSOA. Varios trabajos han determinado la características que deberían tener este tipo de lenguajes, habiéndose publicado varios hasta ahora [11, 2].

Por último, el DSBC potencia la reutilización de componentes en distintos entornos, facilitando las tareas de mantenimiento y adaptación.

En este artículo se presenta el marco de trabajo *AOSA Space* [13], que pretende aprovechar los beneficios del uso conjunto de las tres aproximaciones mencionadas.

Las características de los elementos del marco de trabajo propuesto en este trabajo se resumen en los apartados que siguen: en el apartado 2 se describe el modelo y en el 3 la metodología. El apartado 4 describe las características del lenguaje OA que se ha definido y las herramientas de las que dispone el ingeniero de software. En el apartado 5 se muestra cómo se puede considerar la evolución de los sistemas como un problema de coordinación. El apartado 6 resume el marco de trabajo, el 7 algunos trabajos relacionados; y en el 8 se comentan algunos trabajos futuros y conclusiones extraídas.

2. *AOSA Model*

AOSA Model es un modelo que describe arquitecturas software de sistemas complejos. En él, los *concerns* (materias de interés) que atraviesan los módulos arquitectónicos se han considerado como aspectos arquitectónicos que, a su vez, son requisitos identificados en etapas tempranas del ciclo de vida, durante la evolución o el mantenimiento. Los aspectos se han considerado como componentes arquitectónicos.

En *AOSA Model* [12], los componentes del sistema, conteniendo su funcionalidad, se han tratado como cajas negras. La interacción entre los aspectos y los componentes se lleva a cabo de un modo transparente a éstos, siendo los aspectos independientes del contexto, por lo que se potencia su reutilización. Los aspectos se insertan en el sistema mediante la identificación de los puntos de corte localizados en las interfaces públicas de los componentes. La interacción entre los componentes del sistema y los componentes de aspecto se ha especificado en *AOSA Model* mediante la inclusión de unos componentes arquitectónicos que llevan a cabo las tareas de coordinación y composición de los aspectos y el sistema en el que se insertan; estos componentes responden a las características de los elementos coordinadores definidos en el modelo de coordinación en el que se basa su definición. Las acciones de composición se ejecutan dinámicamente en tiempo de ejecución, considerando la prioridad de ejecución entre aspectos si fuera necesario.

El modelo de coordinación en el que se basa el marco de trabajo permite que los nuevos elementos sean insertados de un modo

transparente a los componentes del sistema inicial. En concreto, utiliza un modelo de coordinación exógeno orientado a control, *Coordinated Roles* [10], que permite la inclusión en tiempo de ejecución de nuevos elementos utilizando protocolos de notificación de eventos.

El modelo proporciona las siguientes características para mejorar la adaptabilidad de los sistemas:

- *AOSA* facilita la inclusión de aspectos durante el diseño arquitectónico de un sistema. En particular para adaptar los sistemas al incorporar o modificar requisitos.
- El modelo permite el desarrollo de sistemas OA desde un punto estructural, a partir de las especificaciones que definen el sistema y un juego de requisitos que se incluyen en él en tiempo de diseño. Se sigue un modelo OA asimétrico durante la fase de diseño arquitectónico, lo que facilita la reutilización tanto de componentes como de aspectos.
- La definición estructural del sistema extendido con aspectos, está formada por dos nivel superpuestos, como se muestra en la figura 1:
 - El sistema inicial (*nivel base*) contiene todos sus componentes y las conexiones entre ellos. *AOSA Model* asume que se especifica antes de incorporar los aspectos – nuevos requisitos–.

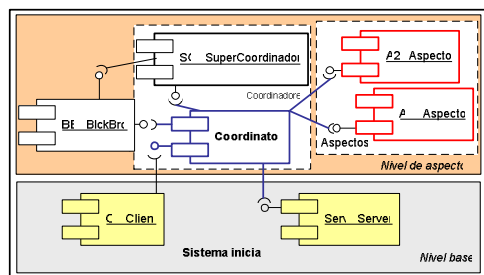


Figura 1. Estructura arquitectónica de *AOSA Model*.

- A este nivel base se superpone un meta-nivel que incluye: el conjunto de aspectos a ser insertados en el sistema inicial, varios coordinadores –elementos que introduce el modelo para gestionar la interacción entre los componentes del sistema inicial y los de aspecto–, y un componente “blackboard” que proporciona los servicios de lectura y

escritura sobre una estructura a la que acceden los coordinadores. De este modo, el sistema extendido tiene capacidades reflexivas para adaptar el comportamiento del sistema inicial. Los nuevos requisitos (considerados como aspectos) se mantienen independientes del contexto en el que se insertan, siendo los elementos coordinadores los que contemplan la información contextual.

3. Metodología para *AOSA Model*

Para trabajar con el marco de trabajo se ha desarrollado una metodología. Sus características principales se basan en las siguientes etapas como se ve en la figura 2:

- 1) La primera etapa considera el sistema en términos de su especificación de requisitos. El sistema inicial se describe mediante casos de uso.
- 2) A continuación se especifica el modelo de diseño para el sistema inicial y se describe su arquitectura mediante un LDA. El lenguaje elegido es LEDA [3], lenguaje convencional con fuerte base formal que además, permite la ejecución de un prototipo.
- 3) Durante la evolución del sistema se incorporan los nuevos requisitos como aspectos. Estos se integran en el diagrama de casos de uso inicial. Además, es necesario especificar las condiciones de aplicación de cada uno. A continuación se describen los diagramas de secuencia asociados a la modificación realizada, en los que se representan los aspectos incorporados como componentes de diseño.
- 4) El siguiente paso es generar la descripción arquitectónica para el nuevo sistema en un LDA. La figura 3 resume las características del LDA-OA que se ha propuesto para expresar la arquitectura de un sistema orientado a aspectos, según se explica en el siguiente apartado. Además, un juego de herramientas permite traducir esta descripción a un lenguaje convencional.
- 5) Finalmente es posible generar un prototipo del sistema extendido y chequear su corrección.

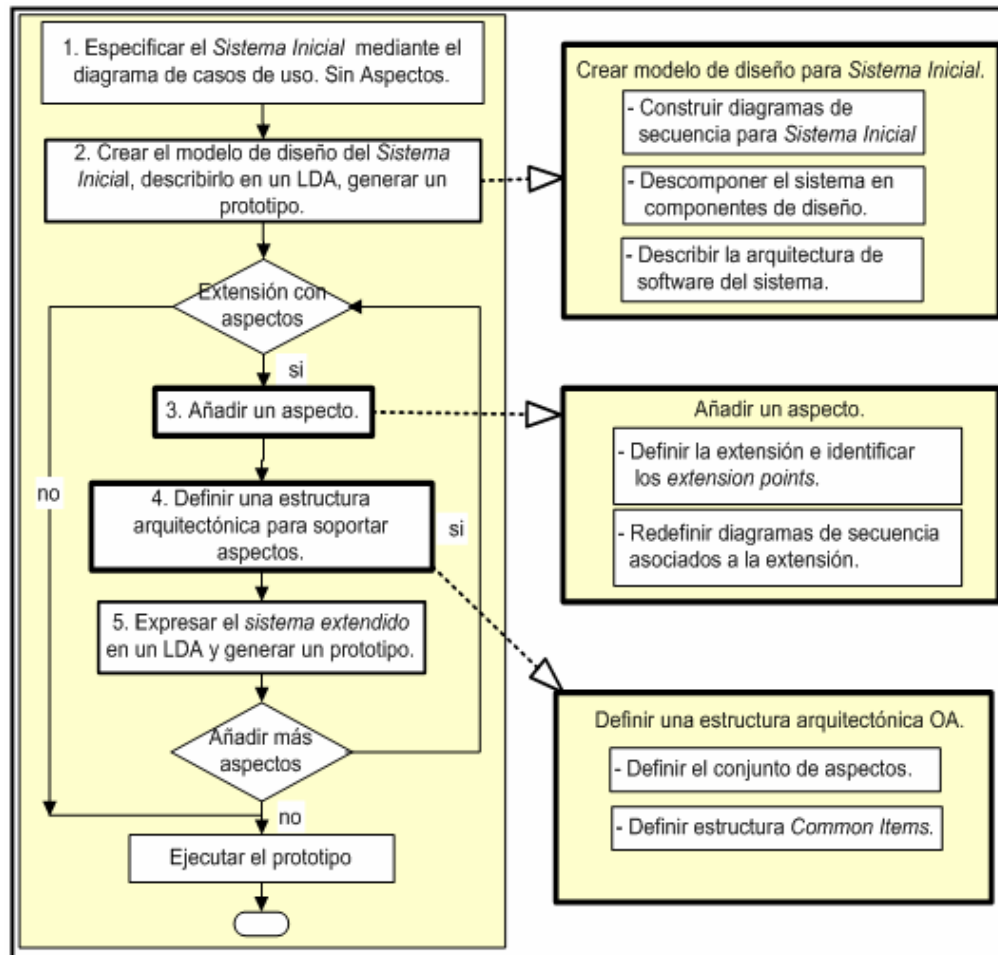


Figura 2. Etapas y pasos de la metodología.

4. LDA-OA para AOSA Model

En los trabajos relacionados se pueden encontrar varias propuestas para desarrollar sistemas OA [4], pero sólo algunos proporcionan soporte lingüístico. Para dotar a los sistemas OA de soporte formal *AOSA Space* dispone de un lenguaje de descripción arquitectónica:

AspectLEDA [14] es el LDA Orientado a Aspectos que se ha desarrollado y permite insertar nuevos requisitos (como aspectos) en el sistema, en tiempo de arquitectura y/o durante su evolución, sin tener que modificar el código.

Además, este lenguaje amplía un LDA convencional, LEDA, y que cumple las premisas descritas en [11]. Sus principales características son:

- Se ha definido con un pequeño conjunto de instrucciones que extiende el lenguaje en el que se basa –LEDA-. Así, la descripción arquitectónica de un sistema en *AspectLEDA* está constituida por 5 secciones, según se muestra en la figura 3, y definen el componente sistema, los componentes que lo forman, las asociaciones entre ellos, una declaración de prioridad y la de instancia.

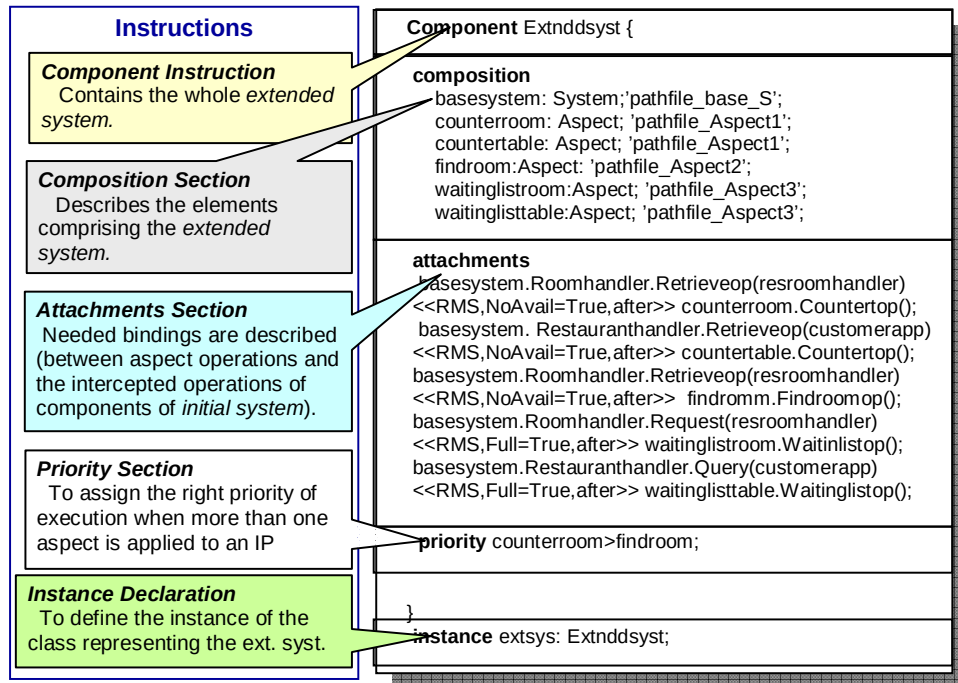


Figura 3. Descripción arquitectónica en *AspectLEDA*.

- Las especificaciones arquitectónicas que describe se traducen a LEDA.
- La base formal de LEDA permite razonar sobre las propiedades de la arquitectura software del sistema.
- Los aspectos se consideran como componentes del lenguaje que se insertan en el sistema de modo transparente a los componentes que lo forman.
- Dota de soporte formal a la descripción arquitectónica de los sistemas orientados a aspectos.
- La nueva arquitectura se puede verificar utilizando herramientas de validación adecuadas.
- Es especialmente interesante la característica de generación de código ejecutable en tiempo de diseño.

Todas estas características del lenguaje hacen que sea adecuado para que los sistemas desarrollados sean fácilmente adaptables.

Además, para comprobar la corrección de la arquitectura se ha elegido un LDA con soporte formal fuerte (Pi Calculus [9]) que permite aplicar herramientas de validación y chequeo, a fin de

asegurar que la arquitectura expresa adecuadamente los requisitos propuestos, sin que sea necesario esperar a realizar esta comprobación en tiempo de implementación.

4.1. Juego de herramientas

El marco de trabajo dispone de un juego de herramientas que facilitan las tareas de traducción y obtención de un prototipo del sistema extendido, en tiempo de diseño. Las herramientas son las siguientes:

- *AOSA Tool/LEDA* facilita la traducción de la arquitectura del sistema extendido, expresada en *AspectLEDA*, al LDA en el que se apoya (LEDA), mediante un proceso de interpretación en dos pasos. Conviene decir que durante el proceso de generación del sistema extendido se realizan labores de validación sintáctica y semántica, y de coherencia de datos. La descripción arquitectónica que se obtiene puede ser modificada, refinada o corregida usando la herramienta [8].

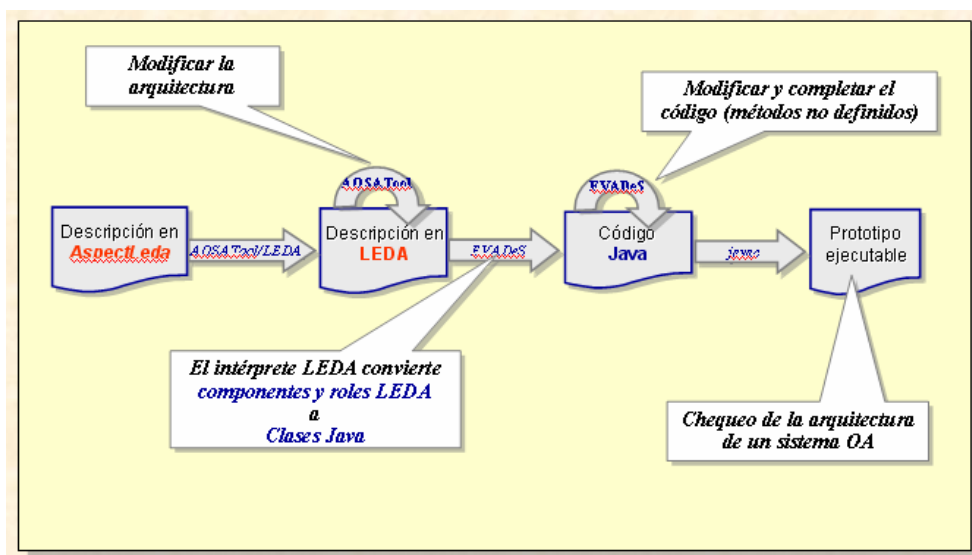


Figura 4. Pasos para generar un prototipo siguiendo AOSA Model.

- Realizada la traducción a LEDA, otra herramienta (EVADeS [6]) facilita la generación y ejecución de un prototipo de la nueva arquitectura en Java. Para ello, los elementos que describen la arquitectura en LEDA se convierten en clases java. El código generado puede ser modificado, refinado y completando usando esta herramienta. El prototipo se obtiene al invocar el intérprete de java. Tras su ejecución se puede evaluar si el sistema se comporta del modo esperado o no. Esto permite detectar errores que de otro modo aparecerían en etapas posteriores.

La figura 4 muestra gráficamente los pasos del proceso de traducción de un sistema descrito en *AspectLEDA* hasta obtener un prototipo ejecutable, después de aplicar ambas. Finalmente, se pueden aplicar herramientas de chequeo de la arquitectura [15].

5. Evolución de sistemas: ¿un problema de coordinación?

Tras conocer la metodología y el proceso de obtención una arquitectura OA, se puede decir que el proceso de extender sistemas existentes con nuevos requisitos se basa en la idea de considerar

la integración de aspectos como un problema de coordinación. Bajo esta idea subyacen dos conceptos:

- Por una parte, el concepto de coordinación que propone la separación o extracción de la información relativa a la interacción entre elementos; también propone mecanismos para la reorganización y composición de esos elementos de modo que el sistema se comporte como si nada se hubiera separado.
- Por otra parte, el concepto de separación de aspectos propone separar y extraer ciertas partes del código. Igualmente se proporcionan mecanismos para permitir que se integren en el sistema, y éste se comporte como si el código extraído no se hubiera separado.

Por tanto, se puede establecer una relación entre estos dos conceptos, y así se puede decir que “la adaptación de sistemas usando la separación de aspectos a nivel arquitectónico, tiene algunas similitudes con los problemas de coordinación”, que ya han sido resueltos aplicando modelos de coordinación. Esta relación se hace más evidente cuando se considera que la especificación de los aspectos, identificados en el sistema, atraviesan la especificación de los componentes, de modo que la interacción entre estos elementos debe considerarse de un modo diferente.

Establecer estas nuevas interacciones supone, por una parte, que para cada componente hay que identificar los puntos en el que se han extraído los aspectos o en donde se van a insertar. Es decir, hay que especificar los join points², así como las conexiones entre los aspectos y estos join points. Además, estas conexiones deben describir también las condiciones de ejecución de cada aspecto (cuándo y cómo aplicarlos). Incluso los conectores serán responsables de la gestión de la ejecución del sistema extendido con aspectos.

Por tanto, se pueden abordar problemas de separación de aspectos como si fueran cuestiones de coordinación. Los modelos de coordinación [1], en particular los modelos de coordinación exógenos se pueden utilizar para gestionar la interacción entre los componentes funcionales y de aspecto. En concreto, *Coordinated Roles* [10] es un modelo de coordinación exógeno orientado a control en el cual una entidad coordinadora se encarga de atender los eventos que ocurren en los objetos que coordina. Además estas acciones de coordinación son transparentes a los objetos coordinados.

Por esta razón, y después de estudiar los modelos de coordinación en general y *Coordinated Roles* en particular, podemos decir que las características de la separación de aspectos durante el diseño arquitectónico se puede gestionar como un problema de coordinación. Esta característica fue considerada en la definición de *AOSA Model*, siendo ésta una de las características distintivas del modelo y una de las principales contribuciones. Además, esta característica lo hace especialmente útil en el desarrollo de sistemas, facilitando la evolución y la adaptación de nuevos elementos al permitir que los componentes permanezcan ajenos (inconscientes) a la inserción de los requisitos asociados a la evolución.

6. Marco de trabajo

En esta sección se muestra, de modo resumido, el marco de trabajo que se ha descrito en este artículo, después de presentar los elementos que lo componen. La figura 5 lo representa gráficamente.

² *Join Points* en [5] se define como lugares en el flujo de un programa al que se asocia comportamiento adicional.

En ella, las elipses con líneas finas representan especificaciones en UML; las elipses con líneas gruesas simbolizan la representación formal de la arquitectura (en un LDA). En negrita y fondo más oscuro se representa el sistema OA, expresado en UML y su descripción arquitectónica formal. Las flechas representan las acciones ejecutadas por el marco de trabajo. Los pasos que se siguen son los siguientes:

- (1) El sistema inicial se representa mediante un conjunto de especificaciones expresadas en su diagrama de casos de uso y los correspondientes diagramas de secuencia (elipse 1). Cada elemento debe describirse completamente a fin de elaborar la oportuna documentación. Esta descripción debe completarse desde el punto de vista arquitectónico en un LDA (1' en la figura 5). A continuación, se debe expresar la modificación del comportamiento del sistema inicial incorporando la nueva funcionalidad: requisitos que se consideran como aspectos. Las incorporaciones deben describirse adecuadamente como *use case extension* siguiendo la propuesta de Jacobson en [7]. Los nuevos requisitos se consideran componentes arquitectónicos (1a en la figura 5) y, por tanto, deben expresarse en el LDA (1a').
- (2) La especificación del sistema OA, sistema extendido, se obtiene incorporando a las especificaciones del sistema inicial las de los nuevos requisitos. El diagrama de casos de uso inicial se extiende con las especificaciones de aspecto. Por tanto, los diagramas de secuencia también resultarán modificados. Para realizar la inserción de cada aspecto hay que considerar cierta información extra: sus condiciones de aplicación, dónde, cuándo y cómo deben aplicarse.
- (3) La descripción arquitectónica del nuevo sistema, se debe expresar en un LDA-OA. De este modo, se puede chequear la consistencia y corrección del sistema extendido.
- (4) La representación del sistema en *AspectLEDA* se traduce a un LDA convencional (LEDA), que permite a su vez generar un prototipo del sistema cuando se traduce a Java (pues el lenguaje elegido lo permite).

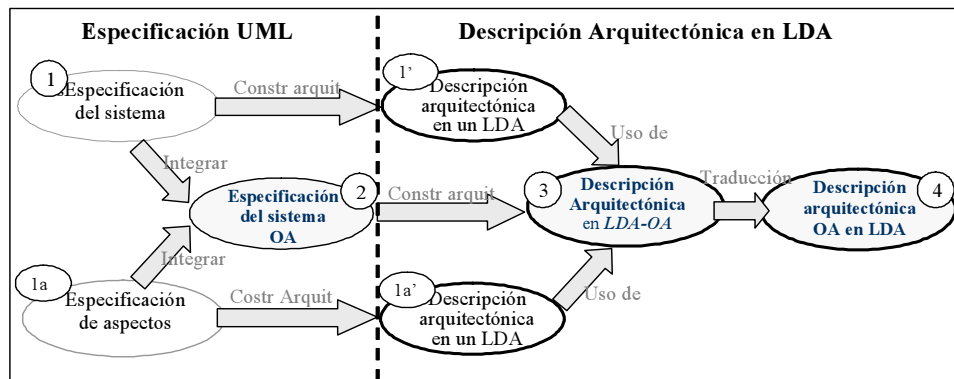


Figura 5. Marco de trabajo propuesto.

7. Trabajos relacionados

En esta sección se comentan brevemente algunos trabajos en el área de la arquitectura del software y DSOA, y su relación con el marco de trabajo descrito.

Nuestro estudio de la bibliografía en el área de la AS muestra el interés de esta disciplina para el desarrollo software, y en particular en el uso de los LDA en la descripción formal de los sistemas. Se han definido varios [13]: Darwin, Unicon, ArchJava, Wright, LEDA, Rapide, PiLAR, entre otros. Sin embargo, ninguno permite modelar las propiedades transversales presentes en ciertos sistemas. Por ello, fue necesario estudiar otros modelos y LDA que resolvieran esta cuestión.

Así, en el área del DSOA estudiamos varios modelos que consideraran las características de las propiedades transversales. Se pueden mencionar: MDSoc y AOSD Use Cases que están orientados a la especificación de requisitos; y Theme, PCS, CAD/DAOP, Prisma Model y TranSAT Method que ponen especial atención en las características arquitectónicas de los sistemas [4].

En la aproximación DSOA parece natural usar LDA para formalizar las arquitecturas de los sistemas OA, pero los LDA convencionales no resuelven la cuestión de las propiedades transversales. Por esta razón, se han propuesto varios LDA-OA, cada uno con sus propias características.

Sin embargo, se hace patente la falta de un modelo arquitectónico que cumpla con los

siguientes criterios: asimetría, considerar los aspectos como componentes, definición de la interacción de los aspectos internamente a los componente del sistema, estar soportado por un LDA con suficiente base formal, disponer de herramientas que faciliten la aplicación del modelo y que permitan validar la arquitectura generada, y obtener un prototipo ejecutable. Este es, en definitiva, el objetivo último de este trabajo.

8. Conclusiones y trabajos futuros

Aunque la propuesta presentada se ha realizado en el ámbito del desarrollo de sistemas orientados a aspectos, a lo largo de este artículo se ha mostrado cómo es posible utilizarlo en la adaptación y evolución de sistemas: la evolución de los sistemas se puede gestionar considerando los cambios (nuevos requisitos) como aspectos. Para lograrlo, los aspectos se extraen, se definen o se identifican durante la etapa de diseño arquitectónico o durante la evolución y se consideran artefactos de diseño que describen su comportamiento a alto nivel. Los aspectos, además, se incorporan de un modo transparente y se pueden manipular a lo largo del desarrollo. De este modo, se dispone de reconfiguración arquitectónica *ad hoc* del sistema cuando un arquitecto de software necesita incluir un nuevo requisito o un cambio no previsto, obteniéndose así un nuevo diseño a partir del existente, sin cambiar los componentes de diseño que formaban el sistema inicial. Por tanto, el diseño obtenido es

claro, y la reutilización y evolución pueden ser abordadas de un modo más sencillos.

En consecuencia, de la aplicación conjunta de los conceptos de AS y de DSOA y de la utilización de un modelo de coordinación exógeno orientado a control, como es *Coordinated Roles*, se obtiene la principal contribución de este trabajo: reducir el problema de la integración de aspectos a uno de coordinación. En este sentido, se puede colegir que, según lo expuesto, el problema de la adaptación de nuevos requisitos a los sistemas se puede reducir a uno de coordinación.

Otra contribución es el modelo arquitectónico descrito, *AOSA Model*, que se caracteriza por ser reflexivo y con dinamismo estructural, ya que el modelo propone incluir nuevos componentes a partir de un sistema ya diseñado y el lenguaje en el que se apoya la propuesta lo soporta, pues permite la creación y destrucción de componentes, y de los enlaces correspondientes.

La reconfiguración del sistema se lleva a cabo en tiempo de diseño en cuanto que *AOSA Model* propone redefinir la arquitectura del sistema incluyendo los elementos que representan la ampliación (como aspectos). Igualmente, se dispone de reconfiguración dinámica una vez extendido el sistema, ya que los componentes de aspecto, si bien están ya especificados, no siempre tienen instancias activas o, incluso, puede haber varias activas a la vez.

Permite, además, reconfiguración programada, según la clasificación propuesta por Endler.

El modelo arquitectónico resultante es altamente dinámico por las facilidades que proporciona el LDA que se utiliza, que tiene gran capacidad de dinamismo; permite diseñar sistemas orientados a aspectos independientes de la plataforma y de la tecnología subyacente.

Los componentes no tienen que haber sido definidos bajo ningún modelo de componentes específico, sino que se basa en la especificación genérica del modelo de componentes de UML 2.0.

En cuanto a los trabajos futuros se pueden mencionar los siguientes:

- En este momento se estudia cómo llevar a cabo una ampliación de *AOSA Tool/LEDA* para que, a partir de la especificación LEDA generada por la herramienta, se pueda obtener automáticamente la conversión a especificaciones en Pi-Cálculus.

- Igualmente es necesario completar el trabajo aquí presentado desarrollando algún entorno de trabajo o herramienta que facilite el testeo, validación y prueba de la arquitectura generada.
- En un estado de estudio preliminar se encuentra la definición de un entorno que integre las tres herramientas desarrolladas, así como la mejora de cada una de ellas.

Referencias

- [1] Arbab, F., What Do You Mean Coordination? Bulletin of the Dutch Association'96. LNCS 1061. Springer-Verlag. Cesena. April 1996.
- [2] Batista, T., Chavez, C., García, A., Kulesza, U., Sant'Anna, C., Lucena, Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs. Proceedings of the ACM Sigsoft. XX Brazilian Symposium on Software Engineering (SBES'06) Florianópolis, Brazil. October, 2006.
- [3] Canal, C., Pimentel, E., Troya, M. Compatibility and Inheritance in Software Architecture. Review Scene and Computing Programming. vol 41 no. 2, pp 105-130. 2001.
- [4] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. Lancaster University, Lancaster, AOSD-Europe Deliverable D11, <http://www.aosdeurope.net/>. AOSD-Europe-ULA NC-9. May, 2005.
- [5] Filman, R. et al. Aspect Oriented Software Development. Addison Wesley. 2005.
- [6] Garcia, C., Navasa, A. EVADeS: Un Entorno Visual de Asistencia al Desarrollo de Sistemas en LEDA. TR-30/2008 University of Extremadura. 2008.
- [7] Jacobson, I., Ng Pan-Wei. Aspect-Oriented Software Development with Use Cases. ISBN: 0-321-26888-1. Addison Wesley. 2005.
- [8] Merín, I., Navasa, A. AspectLEDA: Un Lenguaje de Descripción Arquitectónica Orientado a Aspectos. TR 31/2008. University of Extremadura. 2008.
- [9] Milner R., Parrow J. and Walker D. "A Calculus of Mobile Processes". Information and Computation 100(1) pag: 1-40, Sept 1992.
- [10] Murillo, J.M., Hernandez, J., Sánchez, F., Alvarez, L. Coordinated Roles: Promoting Re-Usability of Coordinated Active Objects Using Event Notification Protocol. Coordination Proceeding, pp53-68, 1999.

- [11] Navasa, A., Pérez, M. A., Murillo, J. M., Hernández, J. Aspect Oriented Software Architecture: A structural Perspective. WS on Early Aspect. AOSD Conf. 2002. Enschede Holland. <http://trese.cs.utwente.nl/AOSDEarlyAspectsWS/works hop papers.htm>.
- [12] Navasa, A., Perez, M. A., Murillo, J. M. Aspect Modelling at Architecture Design. EWSA 2005. LNCS 3527, pp. 41-58, Ed. R. Morrison and F. Oquendo. Springer Verlag. Berlin Heidelberg. ISBN: 3-540-26275-X. 2005.
- [13] Navasa, A. Marco de trabajo para el desarrollo de arquitecturas software orientado a aspectos. Tesis doctoral. 2008.
- [14] Navasa, A. Pérez-Toledano, M. A., Murillo, J. M. An ADL dealing with aspects at software architecture stage. Information and Software Technology, 51 (2009) pp 306-324.
- [15] Pérez-Toledano M. A., Navasa, A., Murillo, J. M., Canal, C. A Safe Dynamic Adaptation Marco de trabajo for Aspect-Oriented Software Development. [Journal of Universal Computer Science](#). Special Issue on Practical Approaches to Software Adaptation. September, 2008.

Auto-Adaptación en Organizaciones de Agentes: de la Iniciativa al Acuerdo Emergente

J. Santiago Pérez-Sotelo
CETINIA

Universidad Rey Juan Carlos
Móstoles 28933 Madrid
josesantiago.perez@ia.urjc.es

Carlos E. Cuesta

Kybele, Dep. LSI II
Universidad Rey Juan Carlos
Móstoles 28933 Madrid
carlos.cuesta@urjc.es

Sascha Ossowski

CETINIA
Universidad Rey Juan Carlos
Móstoles 28933 Madrid
sascha.ossowski@urjc.es

Resumen

Con la creciente complejidad de los problemas a resolver necesitan cada vez más de sistemas software que presenten autonomía, robustez, flexibilidad y adaptabilidad como características sobresalientes. Las estrategias de desarrollo y mantenimiento, por lo tanto, están siendo reconcebidas. Una de las tendencias es que gran parte de estas tareas deberían ser realizadas por los propios sistemas, considerando la auto-adaptación de los mismos como una cuestión básica de arquitectura. Paralelamente, los sistemas multiagentes son muy populares en el área de la inteligencia artificial como enfoque para resolver problemas complejos. Ciertos enfoques avanzados recurren al concepto de organizaciones para proveer nuevas estructuras, tomando la forma de arquitecturas complejas. Entre ellos, las Tecnologías del Acuerdo proporcionan una visión explícita de esas abstracciones arquitectónicas. Sin embargo, aún no proveen mecanismos para cambiar sus patrones de composición y tipos de elementos, necesarios para lograr una real auto-adaptabilidad. En este trabajo, se propone una solución arquitectónica: el dinamismo requerido será soportado por un acuerdo emergente – una estructura arquitectónica evolutiva, basada en la combinación de controles y protocolos predefinidos. Estos son tratados en el contexto de un marco orientado a servicios, basado en agentes y centrado en organizaciones, definido en las Tecnologías del Acuerdo y provisto por la implementación en la plataforma THOMAS. El trabajo proporciona las primeras abstracciones arquitectónicas para soportar esta estructura emergente, presenta también la utilidad de este enfoque y, finalmente, se tienen algunas conclusiones acerca de su aplicabilidad.

1. Introducción

Es sabido que en los últimos años los sistemas de software se han vuelto cada vez más complejos. Esta complejidad, que puede ser llamada “social” según [14], está impulsando a los diseñadores de software a repensar las estrategias para su desarrollo. Muchas de las tareas rutinarias que previamente estaban reservadas a usuarios humanos ahora están siendo manejadas por los propios sistemas, incluso acciones relacionadas con las funciones propias del sistema. Los sistemas complejos pueden ahora observarse a sí mismos, y adaptar su estructura y comportamiento si fuera necesario. Por tanto, como este enfoque tiene una influencia global en el sistema y a muchos niveles [18], se puede considerar la auto-adaptación como una cuestión básica de la arquitectura [19].

Paralelamente, los Sistemas Multi-Agente (SMA) se han desarrollado como un enfoque genérico en Inteligencia Artificial (IA) para la resolución de problemas complejos. Los SMA describen estructuras auto-conscientes, pueden tener capacidad de aprendizaje, son diseñados para ser flexibles y capaces de adaptarse a diferentes situaciones, entre otras características. Ciertos enfoques utilizan el concepto de organizaciones para proveer nuevas estructuras, las que originan arquitecturas complejas de agentes. Sin embargo, las estructuras existentes aún pueden tener limitaciones al intentar alcanzar una real auto-adaptación, es decir, no sólo tener la capacidad de cambiar su configuración, sino también su propia composición o tipos de elementos constituyentes. En este trabajo se intenta ir más allá de las tecnologías “clásicas” de agentes y proponer una solución basada en las denominadas Tecnologías

del Acuerdo (*Agreement Technologies*) [1][24] para alcanzar el dinamismo.

En el enfoque propuesto los servicios no se ofrecen directamente a través de los agentes, sino a través de las organizaciones. El principal objetivo es proporcionar organizaciones adaptativas y el énfasis está en que el mecanismo de coordinación sea adaptativo, independientemente de cómo los agentes estructuran la forma de exportar sus servicios.

Este trabajo está organizado de la siguiente manera: en la segunda sección se presentan escenarios como motivación para ilustrar las ideas principales y el enfoque propuesto, definido como orientado a servicios, centrado en organizaciones y basado en agentes. En la tercera sección se describe la arquitectura que sirve de soporte a la propuesta: la plataforma THOMAS [3], la que incluye modificaciones para hacerla compatible con las especificaciones OSGi[22]. La plataforma, en sus actuales características presenta ciertas limitaciones, por ejemplo en la adaptación a los cambios en el entorno, en la estructura de las organizaciones, con la aparición de nuevas normas y relaciones sociales entre agentes, etc. las que son motivo de investigación en el proyecto OVAMAH[25]. La sección cuarta discute el núcleo del enfoque propuesto, del que emerge una arquitectura adaptativa dentro de un contexto de SMA. Por último se presentan algunas conclusiones y futuras líneas de trabajo.

2. Motivación: Ejemplos de Uso

Con el fin de ilustrar la situación en la que una arquitectura adaptativa puede ser tomada como solución, en esta sección se presentan escenarios en el dominio de las urgencias médicas. Éstas son hipotéticas pero están basadas en situaciones reales y se relacionan con el módulo demostrador mHealth (*mobile-Health*) del proyecto Tecnologías del Acuerdo [1], prototipo actualmente en desarrollo con la colaboración del servicio SUMMA112 [30]. Esta entidad es la encargada de gestionar las emergencias médicas en la Comunidad Autónoma de Madrid, España. En [8] se puede encontrar más información sobre apoyo a emergencias médicas mediante el uso de un SMA estándar.

Como el escenario propuesto es altamente dinámico, es necesario dotar al sistema de

capacidades que permitan una respuesta dinámica frente a los cambios, por medio de la adaptación y/o evolución de la propia organización.

La urgencia hipotética se refiere a un incendio en un parque urbano situado al oeste del centro de la ciudad de Madrid, el que contiene una amplia zona de ocio. SUMMA112 recibe la información correspondiente y decide que son necesarios 5 ambulancias y un helicóptero. Es también urgente realizar la coordinación con los departamentos Bomberos y Policía. Estos últimos informan que enviarán 3 camiones bomberos y 5 coches policiales, respectivamente. Desde un enfoque organizacional, todos estos elementos formarían una organización, llamémosla O1. En un entorno SMA, cada uno de estos actores se correspondería con un agente. Visto así se tienen 14 agentes que van a interactuar en O1.

Situación 1: el grupo de agentes llega a la zona y debe organizarse para enfrentar la urgencia. Cada uno de ellos tendrá su función, objetivos y planes, por lo tanto para formar parte del grupo serán necesarias negociaciones a distintos niveles, así como también para determinar un mando coordinado. Para ello puede pensarse en una serie de protocolos para elección de un líder, lo que llevaría a una organización preliminar. Cuando los individuos siguen este tipo de protocolo definen patrones estructurales implícitos.

Situación 2: la organización está ya actuando en la zona de desastre y uno de los servicios esenciales, brindado por un agente u otra organización, deja de estar disponible. Diferentes razones pueden ocasionar esta situación: el agente es requerido en otra urgencia, por falta de recursos u otro impedimento no puede llegar a la zona, etc. Ante esto la organización primero debe detectar que ese agente no está disponible y luego buscar una solución alternativa, reemplazar el servicio esencial con alguno similar, por ejemplo. Esta situación muestra que la adaptación no solo requiere la reorganización de los agentes, sino también nuevas configuraciones del sistema y hacer frente a nuevos problemas de coordinación.

Queda claro que en ambas situaciones las organizaciones son dinámicas y por lo tanto se hace necesario modificar sus estructuras, configuraciones y coordinación.

Como el entorno está orientado a agentes el objetivo es lograr una reconfiguración automática: el sistema debe llevar a cabo una serie de pasos

evolutivos hasta que encuentre un punto óptimo. Esto perfectamente puede ser un proceso continuo, ya que el propio escenario va evolucionando. El comportamiento no debería ser completamente pre-diseñado, sino emergente y la coordinación debe lograrse dentro de una arquitectura que es, básicamente, un ecosistema de servicios. En las secciones siguientes se presentan herramientas y conceptos desarrollados con el fin de definir las bases de una solución y como validación del enfoque propuesto.

3. Estado Actual y Evolución del Entorno THOMAS - AT MAS

Como se ha dicho anteriormente, la arquitectura base debe soportar el modelo definido como orientado a servicios, centrado en organizaciones y basado en agentes. Las investigaciones actuales en la plataforma están orientadas a lograr una mayor capacidad y funcionalidad, aprovechando las características de los SMA. Desde este punto de vista, los servicios ofrecidos son utilizados para lograr interoperabilidad. La idea principal es exportar el sistema de *agentes* como un *sistema de servicios*. El ecosistema de servicios resultante será soportado no sólo tecnológicamente sino también metodológicamente [28]. A continuación se presenta el estado actual y la evolución del entorno THOMAS – AT MAS (Agreement Technologies Multi-Agent Systems). En primera instancia, los principales conceptos se construyen sobre la plataforma THOMAS [3]. La misma, incluido su middleware (Figura 1), está estructurada en tres niveles - aunque no son estrictamente capas -, ortogonalmente soportados por cuatro componentes específicos que se incluyen como formando parte de tres diferentes subsistemas.

Los tres niveles citados en primera instancia son:

- *Platform Kernel (PK)*. Kernel del middleware, incluye tanto la capa de red como el componente Agent Management System (AMS). A este nivel de descripción, la plataforma es un SMA abierto.
- *Service & Organization Management*. Nivel conceptual compuesto por los componentes Organization Management System (OMS) y Service Facilitator (SF). No es estrictamente un subsistema, aunque provee las

características y abstracciones relevantes para el Execution Framework.

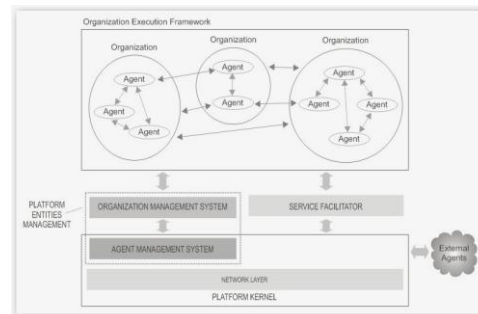


Figura 1. Arquitectura Técnica en Tres Niveles de THOMAS (inspirada en [3])

- *Organization Execution Framework*. Es el “espacio” donde todas las entidades computacionales “viven” y realizan sus tareas. Los agentes, sus organizaciones y los servicios que ofrecen están conceptualmente ubicados allí. Cada aplicación específica se concibe, diseña y ejecuta en este nivel de abstracción.

El subsistema *Platform Entities Management* está a su vez formado por capas (AMS y OMS) que se utilizan para proporcionar las capacidades a los diferentes niveles de la plataforma.

Los componentes mencionados son:

- *AMS*, proporciona las capacidades y funciones necesarias para la gestión de agentes;
- *OMS*, hace lo propio con las organizaciones a la vez que mantiene unido al sistema en su conjunto;
- *SF*, proporciona las capacidades y funciones necesarias para permitir que cierta selección de operaciones de una organización se comporte como un servicio unificado.

Para mayor detalle ver [3].

Los progresos actuales se dan a nivel de los servicios con su adaptación a la plataforma OSGi [22]. OSGi es considerado en la actualidad como un estándar de facto para las plataformas basadas en servicios. La idea principal de su especificación es modularizar las aplicaciones en entidades más pequeñas denominadas *bundles*. Estas entidades pueden ser instaladas, actualizadas o eliminadas dinámicamente, proporcionando la posibilidad de cambiar el comportamiento del sistema.

Entre los servicios proporcionados por el estándar se destaca, para el enfoque de este trabajo, el *Service Tracker*, que permite rastrear otros servicios registrados en la plataforma. Se utiliza para asegurarse que los servicios que se han proveídos aún están disponibles, o en su defecto descubrir que no lo están. Este servicio es básico para enfrentar la segunda situación planteada en la sección 2. En el caso de un sistema basado en *bundles*, tanto productores como consumidores de servicios pueden aparecer y desaparecer en cualquier momento. Para lograr que el pase de mensajes, síncronos o asíncronos, entre dos entidades que pertenecen a distintos *bundles* la especificación proporciona una herramienta: el patrón *Whiteboard*. Éste utiliza el registro de servicios de OSGi para mantener un registro de los *listeners* del sistema. Uno de los beneficios de esta herramienta es que el control del ciclo de vida de productores y consumidores de eventos se delega en la plataforma, que notifica a consumidores la desaparición de productores y viceversa.

Las actuales investigaciones, que se incluyen como parte del proyecto OVAMAH, se desarrollan ampliando los objetivos de la plataforma THOMAS. Además de proveer la tecnología necesaria para el desarrollo de organizaciones virtuales en entornos abiertos, permitirá dar respuestas dinámicas a posibles situaciones cambiantes por medio de la adaptación y/o evolución de las propias organizaciones. Por ejemplo, los agentes integrantes de una unidad organizacional podrían crear (o eliminar) una unidad, afectando a los grupos del sistema; con respecto a las normas, definir en qué momento es necesario agregarlas o eliminarlas; el tipo de relación social entre roles podría cambiar en tiempo de ejecución, tanto las condiciones para activarlas/desactivarlas como la cardinalidad entre roles; la topología del sistema (dada por las relaciones) debería poder cambiarse (agregar/eliminar relaciones) también en tiempo de ejecución y luego verificarse que este cambio es consistente con los objetivos y el tipo organizacional; los servicios podrían ser asociados a nuevos roles que surjan en el sistema; además, como cada objetivo está asociado a unidades organizacionales es necesario también que un objetivo pueda ser dividido dinámicamente en otros más concretos; etc.

Las investigaciones se desarrollan tanto a nivel de servicios como de agentes y SMA, afectando incluso la eficiencia, escalabilidad, performance, etc.

4. Organizaciones Adaptativas: Tipos y Funciones

En esta sección se presentan los conceptos relacionados con dos versiones de organizaciones adaptativas que actualmente se están desarrollando en el proyecto OVAMAH, y también de sus patrones de adaptación. En primer lugar, se tratan las organizaciones que están en un proceso de crecimiento, llamadas *iniciativas*; posteriormente, las organizaciones que modifican su comportamiento, llamadas provisoriamente *reconfiguraciones*.

4.1. Concepto de Organización Adaptativa

Teniendo en cuenta sus objetivos, un grupo de individuos puede ser ordenado según ciertas estructuras (una jerarquía, sociedad, arquitectura, entre otras). Estas estructuras pueden formarse mediante el uso de dos tipos diferentes de elementos, los *controles* y *protocolos*, que limitan el rango de acciones válidas.

Los primeros pueden ser considerados mecanismos que tanto *imponen* como *prohíben* interacciones específicas (o conexiones arquitectónicas). Las estructuras auto-adaptativas, siendo típicamente centralizadas [2], manifiestan mayormente lazos explícitos de control, inspirados en los *reguladores* de la teoría de control clásica.

Por otro lado, los *protocolos*, pueden *permitir* o *canalizar* comportamientos y se basan en el consenso y en acuerdos. Pueden ser descriptos, genéricamente, como una manera de controlar las estructuras descentralizadas (o incluso distribuidas) [16]. Cuando los protocolos están presentes cada agente conoce la forma de interactuar con el resto; es necesario acordar con ellos para poder comunicarse, pero al mismo tiempo, también *regulan* el desarrollo de la estructura de interacción.

Estos dos mecanismos definen una amplia gama de regulaciones, en los que las organizaciones de agentes y sus arquitecturas son simultáneamente reglados por controles unitarios, atómicos (como

las normas, límites, bloqueos, lazos de control o restricciones), y protocolos múltiples, conectivos (concentradores, puentes, canales o espacios).

En arquitectura de software ya son conocidos muchos patrones y soluciones basadas en controles implícitos y restricciones normativas. El enfoque de este trabajo basa la solución en el consenso: también se trata de un enfoque a nivel arquitectónico, ya que la descripción de las interacciones es pertinente a este nivel.

Es importante señalar que el propósito de estos mecanismos es “descubrir” una estructura adecuada de controles y protocolos para que pueda emerger una estructura global. Estos elementos harán posible definir las principales estructuras internas a fin de obtener organizaciones basadas en acuerdos. Por tanto, aunque no es una idea nueva sí, es muy relevante que los agentes se agrupen en organizaciones, a diferencia del clásico diseño de SMA, como se señaló anteriormente.

Una vez que una estructura primaria pueda ser definida, un grupo elemental surge como una organización preliminar, a la que se hará referencia como *una iniciativa*. Esto se explica con mayor detalle en la siguiente subsección.

4.2. Organizaciones Auto-organizadas: las Iniciativas

Como se señaló anteriormente, se puede utilizar un conjunto de controles y protocolos para generar dinámicamente una organización preliminar dentro de un grupo de individuos (agentes, en este trabajo, aunque también podrían ser componentes genéricos).

En el enfoque propuesto se define y utiliza un conjunto de controles y protocolos para generar cierta estructura (por tanto, varios de ellos son considerados como *controles generativos* y *protocolos generativos*). Esta estructura da lugar a una organización que crece con la dinámica del entorno. La organización emergente es lo que se llama *una iniciativa*: todavía no está plenamente establecida, sigue evolucionando.

La *iniciativa* puede seguir creciendo y mutando debido a su naturaleza adaptativa, pero cuando se tiene algún tipo de estructura estable, ya es posible llamarla *organización*. Esta estructura estable se logra cuando todos los participantes pueden llegar al acuerdo necesario con el fin de solucionar el

problema o conseguir el objetivo principal que provocó su unión. La organización resultante es conceptualmente similar a otras organizaciones en varios enfoques de SMA, incluso en el de THOMAS [3].

El párrafo anterior implica tres conceptos importantes en este trabajo:

- *Una iniciativa*. Es un grupo preliminar de individuos (agentes) que se reúnen en una determinada estructura, generada por un conjunto de controles y protocolos, así como también por ciertos patrones asociativos;
- *Una organización*. Se trata de un grupo establecido, es dinámicamente originado desde una *iniciativa* (aunque también hay organizaciones estáticas, una vez que se crean, ambos tipos son funcionalmente equivalentes);
- *Un acuerdo*. Es el acto por el cual una *iniciativa* se convierte en una organización estabilizada. De hecho, puede verse como el consenso que se alcanza entre los individuos dentro de la “semilla” inicial del grupo.

Según este proceso el sistema se mueve a un nuevo estado, en el que la estructura del “pasado” es suplantada por una “nueva estructura emergente”. Obviamente, esta novel estructura admite nuevos elementos debido al entorno dinámico, pero ahora uno de sus objetivos es reforzar su naturaleza y persistir. Por tanto, uno de los objetivos de una *iniciativa* es crecer y, contraponiéndose a esto, el objetivo principal de una organización es mantenerse.

Considerando el ejemplo motivador (Sección 2), en los primeros momentos de la urgencia E1 varios coches policiales llegan al lugar, pero ninguno es líder del grupo. Siguiendo un protocolo interno pueden elegir uno (incluso la jerarquía es un protocolo) y este acuerdo genera una organización preliminar. Cuando se sigue este tipo de protocolo - *protocolo generativo* - se pueden definir patrones estructurales implícitos. Una iniciativa puede generarse a partir de estos patrones, los *patrones del acuerdo*. Son pre-diseñados a partir de los servicios necesarios por una iniciativa y para el refinamiento semántico correspondiente. Algunos ya han sido identificados: *Façade*, *Mediator*, o *Surveyor*, entre otros (ver Subsección 4.4). Es importante notar que si bien algunos de estos nombres son típicos de en el ámbito de los patrones de diseño, se

definen en un contexto completamente diferente. En particular, *no* son clásicos patrones orientados a objeto, aunque sí son patrones de arquitectura.

Los patrones representan una estructura estática (un fragmento) que conduce a una dinámica, la *iniciativa*, alcanzando una forma estable, la organización. Como ya se señaló, el sistema es en última instancia concebido como una arquitectura *orientada a servicios*, por tanto, metodológicamente las organizaciones estables deben ser concebidas como proveedoras de ciertos servicios de alto nivel. Es así que estos servicios deben ser propuestos como punto de partida para la definición funcional de las organizaciones estáticas.

La descomposición funcional de estos servicios (o una descomposición jerárquica, desde otro punto de vista) también serán utilizados para diseñar la estructura jerárquica de las organizaciones. El concepto de *proceso de servicio*, en este contexto, se propone para ofrecer una clara perspectiva semántica de la funcionalidad del servicio, al describirla como un flujo de trabajo. Cada servicio (de alto nivel) se desdobla en un proceso (semántico), que describe la coordinación entre los servicios de bajo nivel; cada servicio está provisto por una organización de bajo nivel, proporcionando la descomposición estructural de la organización de alto nivel anterior. Este proceso guía la definición (semántica) de las organizaciones orientadas a servicio, y se utiliza para definir las estructuras estáticas [28].

Por otra parte, los acuerdos emergentes y dinámicos, *deben* ser coherentes con estas definiciones semánticas, de modo que este proceso proporcione también un método para descubrir los patrones necesarios para las *iniciativas*. Por supuesto, esto aún tiene que ser refinado, y la metodología deseada todavía tiene que adquirir una forma definitiva.

4.3. Organizaciones Adaptativas “Puras”: las Reconfiguraciones

En la subsección anterior se presentaron conceptos relacionados con organizaciones en proceso de crecimiento, en las cuales la autoorganización es clave debido a la importancia que tiene la creación de las estructuras que las contiene. A continuación se discuten conceptos referidos a los cambios que sufren las

organizaciones, que en cierto modo se encuentran ya establecidas. En este caso, llámese “adaptación pura”, la importancia recae en la forma en que una organización existente debe adaptarse a un nuevo comportamiento. En primer lugar debe saber que ha ocurrido un cambio, es decir que el cambio puede emerger de manera intrínseca [29], y luego hacer frente dinámicamente a ese cambio, adaptarse.

Como se había planteado en la Sección 2, cuando la organización ya está actuando en la zona de desastre, un servicio esencial puede pasar a no estar disponible. Es de destacar que el sistema ha llegado a un estado de reposo o seguro para la adaptación.

Ante esta situación, de momento, se han reconocido cuatro alternativas para la adaptación:

Caso 1 - sin modificar el objetivo principal de la organización: internamente se realiza la búsqueda del servicio que más se asemeje al que ha dejado de estar disponible entre los servicios que ofrece la organización. La idea central es el reemplazo directo del servicio.

Caso 2 - sin modificar el objetivo principal: la búsqueda interna anterior solo permite encontrar un servicio de mínima semejanza con el no disponible. En este caso la organización encomienda a los responsables de ese servicio el aprendizaje correspondiente para responder ante los requerimientos del servicio actualmente no disponible. El aprendizaje es perfectamente factible ya que el enfoque propuesto se desarrolla en entornos multiagente. El tiempo utilizado en esta tarea debería ser razonable, de acuerdo con las características del escenario.

Caso 3 - sin modificar el objetivo principal: como alternativa a los dos casos anteriores, es decir si la búsqueda interna falla, la organización estaría facultada para realizar una búsqueda externa a ella. Debido a esto puede considerarse que la organización cambia de estado, volviendo nuevamente al nivel de una *iniciativa*, hasta tanto los acuerdos le permitan estabilizarse, llegar a un estado quiescente.

Caso 4 - modificando su objetivo principal: en este caso la organización se ve “forzada” a modificar su objetivo o dividirlo en objetivos parciales. No resulta posible volver a ofrecer el servicio original.

Estos cuatro casos presentados son los que inicialmente se estudian para una adaptación real de las organizaciones, ya que se modifican tanto la

Nombre	Descripción y funcionamiento
<i>Façade</i>	Un agente ha de representar a la propia organización, en términos de interacción, para poder interactuar fácilmente con una que aún carece de una estructura definida. Este agente será la <i>fachada</i> que redirige cualquier comunicación que haya recibido; no es necesario que sea también un supervisor.
<i>Mediator</i>	Mientras está aún emergiendo, la organización aún no está constituida, y los servicios de datos probablemente no estén funcionando. Por lo tanto, algún agente debe actuar como <i>mediador</i> , lo que hace posible el acceso a fuentes de datos, aunque sea indirectamente, y también para realizar las traducciones necesarias – incluyéndose también varios tipos de traducciones semánticas.
<i>Surveyor</i>	Durante el proceso de surgimiento, al menos un agente debe controlar el crecimiento de la propia iniciativa, para decidir cuándo se introducen nuevos elementos, y también cuándo el grupo se estabiliza para formar una organización. El <i>observador</i> tiene acceso a la biblioteca de patrones – decide cuándo cierto patrón es el acertado y debe ser activado.

Tabla 1: Patrones del Acuerdo

estructura como el tipo de elementos que las constituyen. En un mediano plazo se prevén obtener más casos de adaptación como los mencionados.

4.4. Patrones de Adaptación

Como ya se ha mencionado, los *patrones del acuerdo* son pre-diseñados a partir de los servicios necesarios por una *iniciativa* y para el refinamiento semántico correspondiente. En la Tabla 1 se describen tres patrones identificados: *Façade*, *Mediator* y *Surveyor*.

Este listado no es definitivo ya que existen más patrones; y no todos ellos describen de forma tan clara un *roles* en la organización, sino que más bien intentan definir *procesos*. Por ejemplo, el patrón *Surveyor Election* define el protocolo (uno entre muchos) para decidir cuál será el próximo *surveyor*; y el *Surveyor Change* describe un protocolo para degradar al actual *surveyor* a la categoría de simple participante, para transmitir su conocimiento al nuevo *surveyor*.

5. Conclusiones y Trabajos Futuros

En este trabajo se han explorado conceptos estructurales como base de un enfoque arquitectónico para proporcionar auto-adaptabilidad a sistemas software. El concepto propuesto de *iniciativa* debe ser considerada como

punto de partida para proporcionar mecanismos que permitan cambiar los patrones de composición y tipos de elementos dentro de tales sistemas.

El dinamismo requerido puede ser soportado por un acuerdo emergente - una estructura arquitectónica evolutiva, basada en la combinación de controles y protocolos predefinidos. Estos son tratados en el contexto de un marco orientado a servicios, basado en agentes y centrado en organizaciones, definido por las Tecnologías del Acuerdo, provisto por la implementación en la plataforma THOMAS, el proyecto OVAMAH, y con servicios compatibles con la especificación OSGi [22].

La idea clave es crear un contexto arquitectónico en el que los agentes son *coordinados* y *reorganizados* mediante su inclusión en estructuras preliminares -los patrones del acuerdo - y luego en organizaciones estables. La plataforma descrita, conjuntamente con las modificaciones que serán aportadas por el proyecto OVAMAH, incluye los servicios y facilidades necesarios para llevar a cabo las reconfiguraciones. Los conceptos propuestos pueden considerarse como punto de partida para establecer las estructuras necesarias para lograr una auto-adaptabilidad real.

Tecnológicamente, el trabajo existente es compatible con FIPA [15] y también capaz de interactuar con agentes Jade [17]. De hecho, el enfoque propuesto es prometedor en el sentido de

que es posible lograr la auto-adaptación, pero estos son los primeros pasos y todavía queda mucho trabajo por realizar.

En los futuros trabajos se desarrollarán e implementarán variantes del enfoque a fin de lograr su refinamiento. Los conceptos están evolucionando, pero incluso en esta fase inicial, el enfoque ya ha demostrado su utilidad y su poder expresivo. Los resultados actuales sugieren que la arquitectura adaptativa es posible porque la infraestructura que se ha creado puede crecer simplemente añadiendo nuevos patrones de adaptación. Esto podría satisfacer la promesa de generalizar la utilidad y extensión del enfoque de SMA, adaptándolo a nuevas y más ágiles tecnologías.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación de España, a través de los proyectos *Tecnologías del Acuerdo* (CONSOLIDER CSD2007-0022, INGENIO 2010), OVAMAH (TIN2009-13839-C03-02) y MULTIPLE (TIN2009-13838); así como por el *European Union RTD Framework Programme*, a través de la Acción COST Agreement Technologies (COST Action IC0801).

Referencias

- [1] Agreement Technologies (AT) Project: <http://www.agreement-technologies.org/> (2009)
- [2] Andersson, J.; de Lemos, R.; Malek, S.; and Weyns, D. Modeling Dimensions of Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems*, vol. 5525 Lecture Notes in Computer Science, pages 27-47, Springer, 2009.
- [3] Argente, E., Botti, V., Carrascosa, C., Giret, A., Julian, V., and Rebollo, M.: An Abstract Architecture for Virtual Organizations: The THOMAS Project. Technical report, DSIC, Universidad Politécnica de Valencia (2008).
- [4] Argente, E., Julian, V., and Botti, V.: Multi-Agent System Development based on Organizations. *Electronic Notes in Theoretical Computer Science* 150(3):55-71 (2006).
- [5] Atienza, M., Schorlemmer, M.: I-SSA - Interaction-situated Semantic Alignment. *Proc Int. Conf. on Cooperative Information Systems (CoopIS 2008)* (2008).
- [6] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D.: *Web Services Architecture*. W3C WSA Working Group, W3 Consortium (2004)
- [7] Casadei, M., and Viroli, M.: Applying self-organising coordination to the emergent tuple organization in distributed networks. (Brueckner, S. et. al. editors) 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008).
- [8] Centeno, R., Fagundes, M., Billhardt, H., and Ossowski, S.: Supporting Medical Emergencies by SMA. In *Agent and Multi-Agent Systems: Technologies and Applications*. LNCS, vol. 5559:823-833. Springer (2009).
- [9] Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S.: *Web Services Description Language (WSDL) 1.1*. W3C Consortium. W3C Note (2001)
- [10] Ciancarini, P.: Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300-302 (1996)
- [11] Cuesta, C. E., Fuente, P., Barrio, M., Beato, E. Dynamic Coordination Architecture through the use of Reflection. *Proc. 16th ACM Symposium on Applied Computing (SAC 2001)*, pages 134-140, ACM Press, March 2001.
- [12] DeLoach, S.: Moving multi-agent systems from research to practice. *International Journal of Agent-Oriented Software Engineering* - Vol. 3, N°. 4, pages 378 – 382 (2009)
- [13] Esteban, J., Laskey, K., McCabe, F., and Thornton, D.: Reference Architecture for Service Oriented Architecture 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2008).
- [14] Fiadeiro, J. L.: Designing for Software's Social Complexity. In *Computer - IEEE Computer Society*, pages 34-39 (2007).
- [15] FIPA. FIPA Abstract Architecture Specification. Technical Report SC00001L, Foundation for Intelligent Physical Agents. FIPA TC Architecture (2002).
- [16] Galloway, A.R. *Protocol: How Control Exists after Decentralization*. MIT Press, 2004.

- [17] JADE - Java Agent DEvelopment Framework. <http://jade.tilab.com/>
- [18] Kephart, J.O. and Chess, D.M. The Vision of Autonomic Computing. IEEE Computer 36(1):41-50, January 2003.
- [19] Kramer, J. and Magee, J. Self-Managed Systems: an Architectural Challenge. In Future of Software Engineering (FOSE@ ICSE'2007), pages 259-268, IEEE, May 2007.
- [20] MacKenzie, C., Laskey, K., McCabe, F., Brown, P., and Metz, R.: Reference Model for Service Oriented Architecture 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2006).
- [21] OMA: The Open Mashup Alliance: <http://www.openmashup.org/>
- [22] OSGi: formerly known as the Open Services Gateway initiative, now an obsolete name. <http://www.osgi.org/>
- [23] Ossowski, S.: Co-ordination in Artificial Agent Societies. LNAI 1535. Springer (1999).
- [24] Ossowski, S.: Coordination in Multi-Agent Systems: Towards a Technology of Agreement. LNCS 5244. Springer (2008).
- [25] OVAMAH - Organizaciones Virtuales Adaptativas: Técnicas y Mecanismos de Descripción y Adaptación <http://www.cetinia.urjc.es/es/node/353>
- [26] Papadopoulos, G. A., and Arbab, F.: Coordination models and languages. In Zelkowitz, M. V., editor, The Engineering of Large Systems, volume 46 of Advances in Computers, pages 329-400. Academic Press (1998).
- [27] Pérez, J. S., Cuesta, C., and Ossowski, S.: Agreement as an Adaptive Architecture for Open Multi-Agent Systems. II Workshop on Autonomic and SELF-adaptive Systems - WASELF 2009. TJISBD, volume 3, n. 4, pages 62-76, Ed. SISTEDES (2009).
- [28] Pérez, J. S., Cuesta, C., and Ossowski, S.: Agreement Technologies for Adaptive, Service-Oriented Multi-Agent Systems. II Workshop on Agreement Technologies WAT - XIII Conferencia de la Asociación Española para la Inteligencia Artificial CAEPIA (2009)
- [29] Prokopenko, M., Boschetti, F., and Ryan, A.J.: An Information-Theoretic Primer on Complexity, Self-Organization, and Emergence. Complexity, Volume 15 Issue 1, Pages 11 – 28. Wiley Periodicals, Inc. (2008).
- [30] Rosenschein, J., and Zlotkin, G.: Rules of Encounter – Designing Conventions for Automated Negotiation among Computers. MIT Press (1994).
- [31] Sierra, C., Debenham, J.: Information-Based Agency. Proc Intl. Joint Conference on AI (IJCAI-2007). AAAI Press, pages 1513-1518 (2007).
- [32] SUMMA112: http://www.madrid.org/cs/Satellite?language=es&pagename=SUMMA112%2FPage%2FS112_home (2009).
- [33] Weyns, D., Helleboogh, A., and Holvoet, T.: How to get multi-agent systems accepted in industry? International Journal of Agent-Oriented Software Engineering - Vol. 3, N°. 4, pages 383 – 390 (2009).