Desarrollo de Agentes Software sobre una Arquitectura Basada en Componentes

4 authors, including:

Mercedes Amor
University of Malaga
34 PUBLICATIONS **293** CITATIONS
SEE PROFILE

Lidia Fuentes
University of Malaga
217 PUBLICATIONS **2,217** CITATIONS
SEE PROFILE

José M. Troya
University of Malaga
253 PUBLICATIONS **3,389** CITATIONS
SEE PROFILE

Desarrollo de Agentes Software sobre una Arquitectura Basada en Componentes

M. Amor, L. Fuentes, L. Mandow, J.M. Troya

Dept. Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain
{pinilla, lff, lawrence, troya}@lcc.uma.es

Resumen. *El uso masivo de Internet ha propiciado el desarrollo de aplicaciones basadas en agentes. El desarrollo de sistemas multiagente se aborda desde los métodos, técnicas y herramientas que ofrece la Ingeniería del Software Orientada a Agentes. Sin embargo, a pesar de la existencia de diversas metodologías y plataformas de desarrollo de agentes software, el trabajo del desarrollador se ve incrementado por una falta de arquitecturas de agentes flexibles. Actualmente las arquitecturas de agente proporcionadas por estas metodologías y plataformas no ofrecen la flexibilidad necesaria para desarrollar agentes software adaptables, poniendo poco énfasis en la reutilización. Este trabajo presenta una arquitectura composicional de agentes software en la que, aplicando la tecnología de componentes y el principio de separación de aspectos en el diseño de agentes software, la funcionalidad de los agentes se encuentra distribuida en componentes independientes y reutilizables. Gracias a esta arquitectura el proceso de desarrollo de un agente software se simplifica y se limita a la descripción mediante documentos XML de los componentes software que formarán parte del agente. Hemos utilizado la potencia de las tecnologías Java y Jess como base para implementar nuestro modelo composicional de agentes software.*

1 Introducción

El creciente uso de Internet para realizar tareas de la vida diaria hace necesario el desarrollo de software capaz de hacer frente a entornos distribuidos abiertos y dinámicos. Frente a otras tecnologías usadas para desarrollar aplicaciones distribuidas en la Web, los agentes software parecen presentar las características necesarias para soportar el desarrollo de sistemas abiertos y flexibles en este tipo de entornos.

La Ingeniería del Software Orientada a Agentes (ISOA) permite abordar el desarrollo de Sistemas Multiagente (SMs). El objetivo de la ISOA es proporcionar métodos, técnicas y herramientas para desarrollar y mantener software basado en agentes [1]. El diseño de los SMs se centra en el modelado de los componentes internos de los diferentes agentes sobre una arquitectura de agente específica de una plataforma de desarrollo como Jade [2], Zeus [3] o FIPA-OS [4].

En el caso de Zeus la arquitectura de un agente se compone de un conjunto de subsistemas que permiten el intercambio de mensajes, la planificación y ejecución de tareas, y el almacenamiento de datos entre otros componentes. La arquitectura interna del agente conecta estos subsistemas a través de referencias explícitas. Al contrario que Zeus, FIPA-OS y Jade consiguen, a través del uso y definición de interfaces, desacoplar (aunque no mucho) los componentes que constituyen al agente. Sin embargo, a pesar de ofrecer una arquitectura más flexible éstas tampoco facilitan la reutilización de tareas y el desarrollador se ve obligado a programar totalmente la funcionalidad del agente para cada aplicación.

Nosotros proponemos una arquitectura composicional sobre la que desarrollar agentes software que facilita su construcción a partir de componentes software reutilizables. Esta arquitectura descompone la funcionalidad del agente en componentes totalmente independientes, facilitando la incorporación o sustitución componentes, permitiendo un mayor grado de adaptación del agente resultante. Además, para conseguir una mejor descomposición funcional del agente software, aquellas propiedades que están presentes en (*atraviesan*) varios componentes se modelan como *aspectos* de acuerdo a lo que el desarrollo de software orientado a aspectos propone (DSOA, o en ingles AOSD) [5]. De esta forma abstracciones que forman parte del agente como el comportamiento, los protocolos de interacción, y la distribución de mensajes a través de un servicio de transporte están separados internamente en entidades diferentes dentro de la arquitectura. Esta separación permite alterar cada uno de estos componentes en tiempo de ejecución sin que el resto se vea afectado. Así, por ejemplo, el agente podrá adaptarse al entorno entrenándose en un nuevo protocolo de interacción o transformarse en otro agente cambiando su funcionalidad sin necesidad de incluirla de antemano como parte del agente.

Implementar agentes software sobre alguna de las plataformas de agentes disponibles es una tarea tediosa y propensa a errores en la que se requiere del desarrollador conocimientos de cierto nivel en algún lenguaje de programación y de la arquitectura elegida para el desarrollo. Sin embargo sobre esta arquitectura el desarrollo de nuevos agentes consiste en proporcionar, mediante ficheros de despliegue, la información necesaria sobre la funcionalidad, los protocolos de interacción y la distribución de mensajes que componen el agente inicialmente. Esta propuesta presenta claras ventajas para el desarrollador: Durante el desarrollo se producen menos errores ya que es posible reutilizar componentes probados y libres de errores (componentes COTS –Commercial Off-the-Shelf, o servicios Web); Es posible guiar al desarrollador a través de herramientas capaces de comprobar la corrección de la composición establecida; minimiza la aparición de errores de programación ya que centra el esfuerzo en el ensamblado de componentes, y se acorta el tiempo de desarrollo de aplicaciones basadas en agentes.

En la siguiente sección presentamos nuestra arquitectura composicional para el desarrollo de agentes software, incluyendo la implementación sobre Java y Jess del conector, componente encargado de la coordinación del agente. Posteriormente, presentamos el proceso de desarrollo de este tipo de aplicaciones utilizando este enfoque mediante un ejemplo. Finalmente, en la última sección se extraen algunas conclusiones sobre el trabajo presentado.

2 Arquitectura Composicional de Agentes Software

La Figura 1 muestra el diagrama de clases UML de la arquitectura composicional propuesta. Usamos estereotipos UML para modelar las entidades de nuestro modelo, denominadas <<Component>>, <<Connector>>, <<Mediator>>, <<Distribution>>, e <<Interface>>.

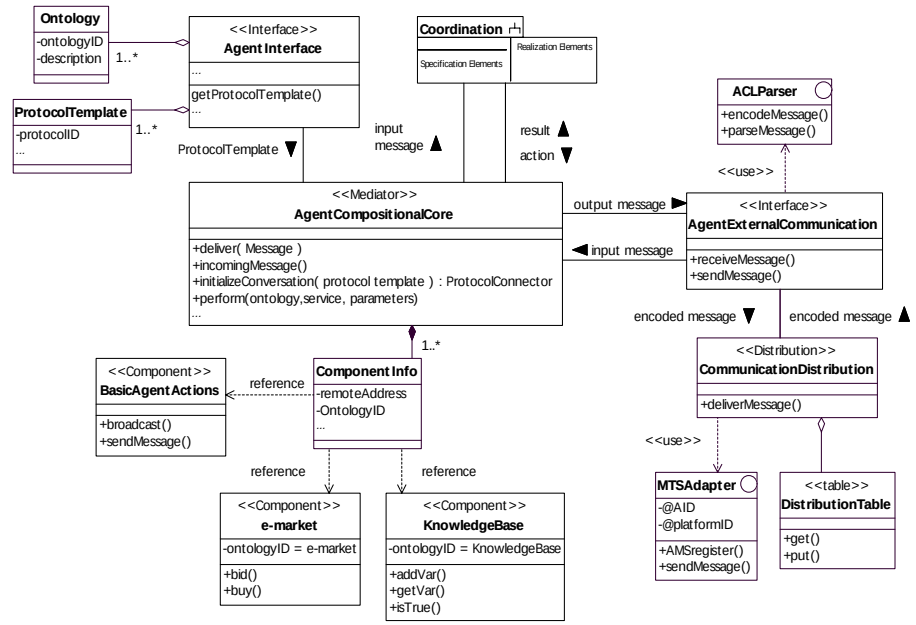

Figura 1. Diagrama de clases UML de la Arquitectura Composicional de Agentes Software.

En nuestra arquitectura la funcionalidad del agente se proporciona por parte de componentes software. Normalmente dentro de un mismo componente encontramos, además de su comportamiento interno, código relativo a la comunicación con el resto de componentes. Esto hace difícil reutilizar sólo la parte funcional de un componente cuando varían los componentes con los cuales se coordina. Aplicando el principio de separación de aspectos modelamos la coordinación entre componentes en una nueva entidad, a la que llamaremos conector (se darán más detalles en la sección 2.1). Los conectores, representados en este diagrama por el subsistema *Coordination*, coordinan las diferentes interacciones o conversaciones en las que participa el agente de acuerdo a un protocolo de comunicación. Un agente puede participar en más de una conversación de forma simultánea, y cada conversación será controlada por un conector diferente. Realmente todos los conectores instanciados son iguales, y sólo difieren en el protocolo que coordinan el cual es suministrado al conector cuando es instanciado.

En general los componentes etiquetados como <<*Component*>> encapsulan datos y comportamiento. Algunos componentes están siempre presentes en la arquitectura ofreciendo la funcionalidad básica del agente como por ejemplo enviar un mensaje

(componente *BasicAgentActions*) o almacenar datos (componente *KnowledgeBase*). Los componentes software también ofrecen funcionalidad específica de un dominio de aplicación, como por ejemplo negociar, pujar o comprar en un mercado electrónico (componente *e-market*). En realidad cualquier componente software puede incorporarse como parte de la funcionalidad del agente, desde un componente COTS (*Commercial Off-The-Shelf*) hasta un servicio Web [6]. Esta flexibilidad se debe a que extendemos el uso de DAML-S [7], una ontología aplicada a la descripción de servicios, para describir, independientemente de la implementación, la interfaz pública de los componentes que proporcionan la funcionalidad del agente. La información acerca de los componentes registrados en la arquitectura, como su identificador y su localización, se almacena en objetos *ComponentInfo*.

Cuando el agente inicia una nueva conversación el componente Mediator *AgentCompositionalCore* (*ACC*) se encarga de crear un nuevo conector que controle la participación en dicha conversación según un protocolo de interacción. Asimismo durante la ejecución del protocolo este componente es el encargado de llevar a cabo la composición dinámica entre los componentes y los conectores estableciendo la correspondencia entre el servicio solicitado por el conector y los servicios ofrecidos por los distintos componentes <<*Component*>> registrados. Como expusimos anteriormente, una ontología nos sirve para describir los servicios ofrecidos por los componentes software. Pero también es utilizada por los conectores, que solicitan la invocación de un servicio utilizando la descripción contenida en la ontología. Una vez resuelto qué componente es el encargado de ofrecer dicho servicio, el ACC localiza el componente a partir de la información del objeto *ComponentInfo* correspondiente e invoca el servicio.

En nuestra arquitectura existen otras abstracciones de los agentes que han sido separadas en entidades independientes, como es la comunicación con otros agentes o recursos del entorno a través de distintos servicios de transporte. Separando este aspecto, denominado de distribución, podemos hacer independiente la participación del agente en una interacción del servicio de transporte utilizado para intercambiar mensajes con otros agentes. Por lo tanto el componente de distribución *CommunicationDistribution* permite que el agente se comunique con otros agentes a través de distintas plataformas de agentes y servicios de transporte haciéndolo más versátil y adaptable. Este componente, mediante el uso de adaptadores, se encarga de enviar y recibir mensajes a través de distintos servicios de transporte. Cada adaptador, que realiza la interfaz *MTSAdapter*, es el encargado de encapsular aquellos aspectos dependientes del servicio como el formato de la envoltura y el protocolo de transporte.

Por otra parte los componentes de interfaz se encargan de las interacciones del agente que no dependen del servicio de transporte ni del protocolo de interacción. El componente *AgentExternalCommunication* (*AEC*) se encarga de codificar y decodificar los mensajes de entrada y salida en diferentes representaciones del lenguaje de comunicación de agentes (ACL), procesando los mensajes de entrada y descartando aquellos que contienen errores sintácticos de acuerdo a su representación. El componente *AgentInterface* contiene la interfaz pública del agente, que es una extensión de la interfaz pública tradicional de los componentes software para el caso de los agentes. En nuestro caso incluye la interfaz de la funcionalidad ofrecida por el agente (y proporcionada por los componentes) y una lista de protocolos de

comunicación que el agente soporta, entre otros elementos. Este componente guarda las descripciones de los protocolos soportados y las ontologías que describen los servicios ofrecidos por los componentes registrados. Las descripciones de los protocolos serán utilizadas por el ACC para crear nuevos conectores cuando el agente inicia o participa en una nueva interacción gobernada por estos protocolos, y las ontologías para realizar la composición dinámica de componentes y conectores.

A continuación describimos brevemente la especificación de un protocolo de comunicación y detallamos la implementación del conector que utiliza esa descripción para coordinar la ejecución del protocolo.

2.1 Implementación del Conector

Aunque dentro de nuestra arquitectura los conectores se encarguen de coordinar la ejecución de un protocolo, los patrones de interacción no se encuentran codificados dentro de los conectores como es usual. En su lugar, aceptan una descripción de un protocolo de coordinación y controlan su ejecución en base a ésta. Una descripción define no sólo el intercambio de mensajes que se lleva a cabo, sino también qué acciones internas lleva a cabo el agente durante la ejecución del protocolo. De esta forma es posible enlazar o conectar el protocolo de interacción con la funcionalidad del agente.

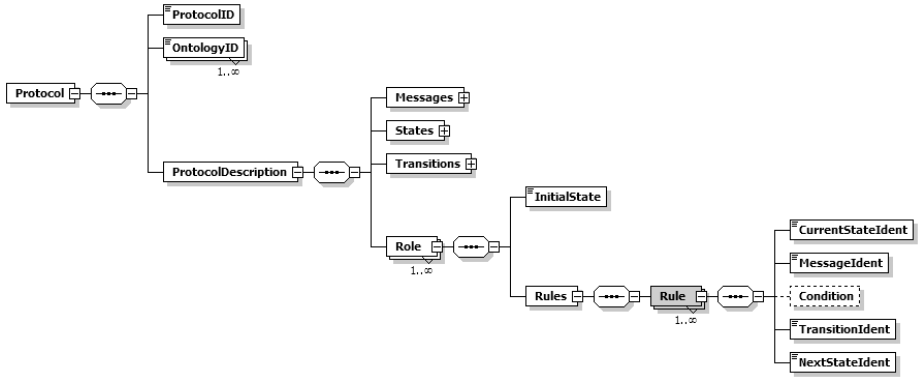

Figura 2. Esquema XML para la descripción de un protocolo de interacción.

Para agregar al agente un nuevo protocolo de comunicación sólo es necesario editar un documento XML que siga el esquema de la Figura 2 describiendo el protocolo. Este documento define una plantilla en la que se describen los mensajes que son intercambiados durante la interacción, los estados por los que pasa el protocolo y el comportamiento del protocolo que se describe, para cada rol participante. La descripción del comportamiento del protocolo consiste en un diagrama de transición que considera, para cada uno de los posibles mensajes de entrada, las acciones a llevar a cabo. Estas acciones, englobadas en una transición, hacen referencia a la funcionalidad interna del agente utilizando los servicios descritos en ontologías. El diagrama de transición es representado mediante un

conjunto de reglas de transición de estados que determinan para cada mensaje válido de entrada la transición correspondiente. Una descripción más completa sobre la especificación del protocolo se encuentra en [8].

En tiempo de ejecución, el conector debe interpretar esta descripción y coordinar la ejecución del protocolo. Con el objetivo de proporcionar un único conector que coordine cualquier protocolo hacemos uso de JESS[9], un lenguaje para el desarrollo de sistemas basados en reglas escrito en Java. Implementamos un conector genérico que es capaz de interpretar y encapsular cualquier descripción de un protocolo descrito en XML que se ajuste al esquema de la Figura 2.

El diagrama de clases UML de la implementación en Java del componente Conector se muestra en la Figura 3. Se ha separado en clases independientes las diferentes funciones de la coordinación. La clase *MessageMatcher* se encarga de establecer la correspondencia entre los mensajes recibidos y los mensajes del protocolo descritos en la plantilla. La clase *FSM* encapsula un motor de coordinación que controla el comportamiento del protocolo según una máquina de estados. Y la clase *ExecutionContext* es la encargada de planificar y monitorizar la ejecución de transiciones.

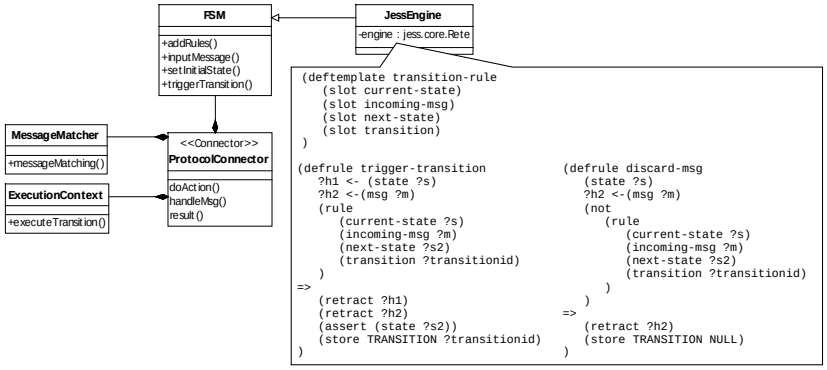

Figura 3. Implementación del componente Conector.

La creación de un conector para coordinar un protocolo implica la instanciación de estos tres objetos, que forman parte del componente <<Connector>> de nuestro modelo. Todos son inicializados para un protocolo concreto descrito en XML. El objeto *MessageMatcher* utilizará la descripción de los mensajes para establecer la correspondencia (método *messageMatching*) entre los mensajes de entrada y los mensajes válidos definidos en el protocolo, devolviendo el identificador del mensaje asignado en la descripción del protocolo. El objeto *FSM* controlará la ejecución basándose en el diagrama de transición incluido en la descripción. Y el objeto *ExecutionContext* utilizará la descripción de las transiciones del protocolo para su ejecución y monitorización.

La subclase *JessEngine* ofrece una implementación de un motor de coordinación utilizando Jess. Esta implementación usa la programación basada en reglas para coordinar la ejecución de un protocolo de comunicación. Durante la ejecución del protocolo esta clase, a partir de reglas que definen el diagrama de transición del protocolo, infiere para cada mensaje de entrada que transición se debe ejecutar y

modifica el estado del protocolo a un nuevo estado. La base de conocimiento contiene dos reglas de inferencia que nos sirven para modelar cualquier diagrama de transición, y un conjunto de hechos que representan el diagrama de transición definidos siguiendo la plantilla mostrada en la Figura 3, y dos hechos que representan el estado del protocolo indicado por el estado actual y el último mensaje recibido, por ejemplo *(state idle)* y *(msg cfp)*. Los hechos que representan el comportamiento del protocolo se añadirán a la lista de hechos inicial del objeto *JessEngine* según la descripción del protocolo, mientras que los hechos que representan el estado irán variando durante la ejecución.

La regla de inferencia *trigger-transition*, mostrada en la Figura 3, se activa cuando se añade un hecho representando la recepción de un mensaje y existe una regla de transición para el estado actual y dicho mensaje. Como resultado de la aplicación de esta regla se produce un cambio de estado y se indica (en la variable *TRANSITION*) el identificador de la transición que será ejecutada. La segunda regla *discard-msg*, también mostrada en la Figura 3 se activa cuando es necesario descartar un mensaje que, a pesar de pertenecer al protocolo, no produce ninguna transición.

Hemos de hacer notar que el conjunto de reglas de inferencia propuestas permite modelar el comportamiento de cualquier protocolo definido como una máquina de estados, y no es necesario implementar diferentes conectores para cada posible protocolo.

La mayoría de las arquitecturas de agentes descomponen la funcionalidad interna del agente en tareas que son llevadas a cabo durante una interacción. La definición de estas tareas es dependiente de la plataforma sobre la que son desarrolladas. Cada plataforma ofrece una biblioteca de interfaces y de clases sobre las que programar la funcionalidad, dado que incluyen código relativo a la coordinación con otras tareas. Esta dependencia dificulta la reutilización de la funcionalidad programada en otras aplicaciones y plataformas. En nuestra arquitectura las tareas se corresponden con los servicios que ofrecen los componentes <<*Component*>>. Sin embargo no existen dependencias ya que lo conectores, encargados de planificar e invocar tareas dentro de una conversación no lo hacen directamente. Los servicios son invocados por el componente *ACC* que se encarga de realizar la composición dinámica cuando un conector solicita la ejecución de un servicio.

3 Desarrollo de Agentes Software

En esta sección mostraremos el proceso de desarrollo de un nuevo agente sobre nuestra propuesta. Para ilustrar este proceso nos referiremos al desarrollo de un agente software para participar en un sistema de subastas en Internet. En este ejemplo el agente debe ser capaz de intercambiar mensajes siguiendo el protocolo de subasta inglesa con un agente subastador FIPA-OS. Durante la subasta el agente debe tener la capacidad de pujar y finalmente, si resulta ganador, debe comprar. Nuestro objetivo es reutilizar el agente FIPA-OS y desarrollar un agente a partir de esta especificación. Partiendo de la especificación dada hemos de:

- Localizar componentes software que ofrezcan la funcionalidad necesaria para pujar y comprar. La arquitectura del agente y el mecanismo de composición no

imponen ninguna restricción sobre los componentes software y únicamente deben ir acompañados de la ontología que cumplen (descrita en DAML-S). En el ejemplo que nos ocupa disponemos de un componente desarrollado en Java que ofrece a través de los métodos *bid* y *buy* la funcionalidad necesaria para pujar y comprar. Este componente está descrito en la ontología *file:///c:/onto/e-market.daml* e implementado en la clase *ca3.behaviour.emarket.class*.

– A continuación es necesario describir en XML del protocolo de comunicación *EnglishAuction* que utilizará el agente para comunicarse con otros durante la subasta. A diferencia de otras arquitecturas no es necesario generar código relativo al comportamiento del agente durante el protocolo. La Figura 4 muestra cómo a partir de una diagrama de estados que modela el comportamiento del participante durante la subasta se genera una descripción en XML de este diagrama que formará parte de la descripción del protocolo. Cuando se comience la subasta y se inicie una conversación el conector añadirá, dentro del objeto *JessEngine* a la lista de hecho inicial los mostrados en la parte inferior izquierda de la Figura 4. Estos hechos son los que utilizará Jess para inferir el comportamiento del protocolo. La descripción de este protocolo de negociación se realizará solamente una vez y podrá reutilizarse para otras aplicaciones.

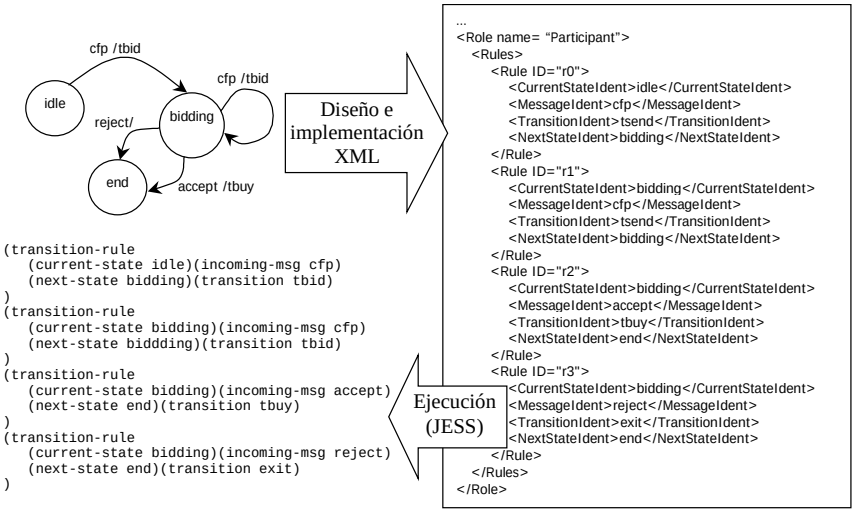

Figura 4. Descripción e Implementación de un Protocolo de Comunicación

En esta especificación dentro de la descripción de las transiciones *tbid* y *tbuy* se hará referencia a los servicios *bid* y *buy* utilizando la descripción DAML-S del componente *emarket*. La descripción XML del protocolo se depositará en una localización accesible para el agente mediante una URI (*file:///C:/xml/EnglishAuction.xml*)

– A continuación, y como parte del desarrollo del agente es necesario localizar adaptadores para los distintos servicios de transporte que se utilizarán para el intercambio de mensajes. Un adaptador deberá realizar la interfaz Java *MTSAdapter* para ser incluido en la arquitectura. Actualmente se proporcionan

adaptadores para los servicios de transporte de las plataformas de agentes FIPA-OS, JADE y Zeus.

– Finalmente, es necesario editar un fichero de despliegue que incluya toda esta configuración, especificando los componentes y descripciones localizados en los pasos anteriores.

En el ejemplo que nos ocupa el documento sería el mostrado en la Figura 5. Este fichero, cuya estructura esta definida en un esquema XML, también contiene información sobre los posibles formatos de codificación de los mensajes. En este caso sabemos a priori que se codificarán los mensajes ACL como una cadena de caracteres. A continuación se detallan los componentes software (componente *emarket*) que serán registrados en el agente como parte de su funcionalidad, indicando también su localización.

Dentro del elemento *protocol* se incluye una referencia a la descripción del protocolo *EnglishAuction* que utilizará el agente para participar en una subasta (en el documento *EnglishAuction.xml)* y su identificador (*EnglishAuction*). Por último, se incluye como información de despliegue una referencia a la implementación del adaptador a la plataforma de comunicación FIPA-OS.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\map\tesis\xml\DeploymentSchema.xsd"/>
<!-- Fichero de despliegue de un agente de subasta-->
   <aclRepresentation>
      <format>
         <ID>String</ID>
         <URI>file:///c:/code/ca3.acladapter.StringEncoding.class</URI>
      </format>
   </aclRepresentation>
   <behaviour>
      <component>
         <ontology>file:///c:/onto/e-market.daml</ontology>
         <URI>file:///c:/code/ca3.behaviour.emarket.class</URI>
      </component>
   </behaviour>
   <coordination>
      <protocol>
         <ID>EnglishAuction</ID>
         <URI>file:///C:/xml/EnglishAuction.xml</URI>
      </protocol>
   </coordination>
   <distribution>
      <platform>
         <adapterTo>FIPAOS</adapterTo>
         <URI>file:///c:/code/ca3.distribution.FIPAOSAdapter.class</URI>
      </platform>
   </distribution>
</deployment>
```

Figura 5. Fichero de despliegue de un agente participante en una subasta.

En el mejor de los casos no será necesario programar nueva funcionalidad, ya que si se desarrollan diferentes sistemas de agentes dentro del dominio del comercio electrónico es posible reutilizar componentes que ofrezcan funcionalidad para pujar o comprar. Y es un hecho probado que no supone ningún esfuerzo de programación incorporar al agente un nuevo protocolo de comunicación. Por otra parte, hemos de puntualizar que esta configuración no tiene que permanecer fija una vez que el agente es creado. El agente es capaz de registrar nuevos componentes o actualizar los registrados en tiempo de ejecución para incorporar nuevas versiones, o puede "aprender" nuevos protocolos de negociación cargando nuevas plantillas, o incorporar nuevos adaptadores, codificadores y decodificadores sin necesidad de suspender la ejecución del agente o reemplazarlo por otro que soporte nueva funcionalidad.

4 Conclusiones

En este trabajo presentamos una arquitectura composicional de agentes software que proporciona una infraestructura para construir agentes a partir de componentes software reutilizables. Esta arquitectura combina componentes y separación de aspectos y separa en entidades distintas la funcionalidad, la coordinación del agente, y la distribución de mensajes. La separación de la coordinación en un componente independiente y el uso de JESS nos permite reutilizar un mismo conector para coordinar cualquier protocolo de comunicación. Tan sólo es necesario proporcionar al conector una descripción XML del protocolo especificando el comportamiento, modelado mediante una máquina de estados, los mensajes válidos y las acciones que el agente lleva a cabo como consecuencia de la interacción.

El proceso de desarrollo de un nuevo agente se reduce, en el mejor de los casos, a la descripción en un documento de despliegue de la composición inicial del agente en cuanto a funcionalidad, protocolos de coordinación y mecanismos de comunicación con otros agentes. La arquitectura composicional propuesta permite al agente adaptarse a nuevos requisitos y modificar su configuración en tiempo de ejecución sin que se vean afectados el resto de los componentes e incluso la actividad en curso del agente, ya que cada conversación es coordinada por conectores independientes.

Actualmente estamos trabajando sobre un entorno gráfico que facilite el desarrollo de agentes software guiando al desarrollador en la especificación de nuevos protocolos y en el desarrollo de nuevos agentes software mediante la generación automática de los documentos de descripción de protocolos y de despliegue del agente.

Referencias

[1] Julián V.J., Botti V., *Estudio de Métodos de Desarrollo de Sistemas Multiagente.* Revista Iberoamericana de Inteligencia Artificial, nº 18 (2003) pp. 65-80.

[2] TILAB: Java Agent Development Framework. http://jade.cselt.it

[3] BtexaCT: The Zeus Agent Building Toolkit. http://193.113.209.147/projects/agents/zeus

[4] Emorphia: FIPA-OS. http://fipa-os.sourceforge.net

[5] Aspect-Oriented Software Development: http://www.aosd.net

[6] Amor, M., Fuentes, L. y Troya, J.M.: Putting Together Web Services and Compositional Software Agents. ICWE 2003. Oviedo, Spain (2003).

[7] DARPA: DAML-S. http://www.daml.org

[8] Amor, M., Fuentes, L. y Troya, J.M.: Training Compositional Agents in Negotiation Protocols. Próxima publicación en Integrated Computer-Aided Engineering International Journal (2003).

[9] Distributed Computing Systems, Sandia National Laboratiories: Jess, The Java Expert System Shell. http://herzberg.ca.sandia.gov/jess/