

Tecnologías y Plataformas de Agentes

Antonio F. Gómez Skarmeta y Juan A. Botía Blaya

14 de octubre de 2002

Índice General

1	Introducción	1
2	Elementos en la programación de sistemas de agentes	2
3	Teorías, arquitecturas y lenguajes de agentes	3
3.1	Programación de agentes BDI	4
3.2	Construcción de agentes reactivos	6
3.3	Arquitecturas híbridas	7
3.4	Construcción de agentes <i>a la</i> FIPA	8
3.4.1	El agente básico	8
3.4.2	Enviando y recibiendo mensajes	9
3.5	Construcción de sociedades de agentes	10
3.5.1	Relaciones en Zeus	10
3.5.2	SimpleTeams en JACK	12
3.5.3	Coordinación en JADE	12
4	Herramientas para la implementación	15
4.1	IDEs (<i>Integrated Development Enviroment</i>)	15
4.2	Depuración y Mantenimiento de código	15
4.3	Otros elementos en depuración de código	16
5	Conclusiones	17

1 Introducción

Los sistemas de información basados en agentes software [34] se han convertido en una tecnología que, cada vez más, se está afianzando como solución a muchos problemas complejos [20]. A este hecho ha ayudado de manera decisiva no solo la disponibilidad de metodologías [21] que asistan en el proceso de análisis y diseño sino también un número cada vez mayor de herramientas de programación para la implementación de los MAS (*Multi-Agent System*) una vez diseñados. Ya a mediados de los años 80 se apuntaba la necesidad de herramientas de programación para la construcción de sistemas basados en agentes [18].

Este trabajo se centra, precisamente, en ofrecer una perspectiva de las herramientas típicas que se pueden encontrar, tanto en entornos académicos como comerciales, centrándonos en sus características comunes y en algunas particulares de interés. No pretendemos ofrecer una panorámica exhaustiva sino resaltar algunos de los enfoques que pueden emplearse, hoy en día, para la implementación de un sistema MAS partiendo de un diseño completo.

Para enfocar mejor el trabajo hacia la parte del dominio al que apunta, seguimos el enfoque de [30], cuando sostenemos que en la ingeniería de un sistema software complejo existen cuatro fases: análisis, diseño, desarrollo y explotación. Pues bien, este trabajo se centra en el estudio de las características básicas que deben tener herramientas que asistan las tareas típicas de la tercera fase. Esta consiste en la programación de la solución especificada en el diseño, haciendo uso de una herramienta de desarrollo concreta. En la sección 2 identificaremos qué elementos hay que tener en cuenta, para su especificación como código, en la programación de un MAS. La sección 3 está dedicada al estudio de distintos enfoques para la codificación de agentes y sistemas de agentes. Los elementos más importantes que podemos encontrar en una herramienta de desarrollo de estos sistemas se detallan en la sección 4. Por último, la sección 5 presenta las conclusiones más importantes que hemos extraído de este trabajo.

2 Elementos en la programación de sistemas de agentes

De la misma forma que en la programación OO (Orientada a Objetos) tenemos elementos básicos con los que se trabaja de una forma esencial como son las clases, mensajes, objetos, en la programación de sistemas de agentes tendremos también varios elementos clave. El primero en identificar estos elementos fue Shoham [31] cuando llevó a cabo el primer intento de diseñar una programación *orientada a agentes*. Shoham ve los agentes de la siguiente forma:

“los agentes resultan entidades computacionales que poseen versiones formales de estados mentales y en particular versiones formales de creencias, habilidades, obligaciones y, posiblemente, otras cualidades mentales”

El lenguaje diseñado por Shoham se denominó **Agent0**. En la programación usando este lenguaje, un agente consta de las partes que enumeramos a continuación.

- Creencias: sentencias lógicas sobre el mundo que conoce, que pueden cambiar en el tiempo.
- Obligaciones (i.e. *commitments*): sentencias lógicas como las anteriores, que se refieren a la ejecución de acciones pendientes de ejecución.
- Reglas de obligaciones: reglas en las que el antecedente está formado por una condición que implica a posibles mensajes a recibir y estados mentales que deben cumplirse cuando se reciben esos mensajes para que la regla se cumpla. En el consecuente podemos encontrar acciones a realizar. Si se cumple el antecedente al recibir un nuevo mensaje, el agente queda obligado a ejecutar la acción del consecuente.
- Capacidades: acciones que puede llevar a cabo el agente.

Así, un ejemplo de regla de obligación codificada en **Agent0** puede ser el siguiente:

```
(COMMIT
  ( agent, REQUEST, DO(time, action)),
  (B, [now, Friend agent] AND
    CAN(self, action) AND
    NOT [time, CMT(self, anyaction)]),
  self,
  DO (time, action))
```

En la regla podemos ver dos condiciones. La primera tiene que ver con la recepción de un mensaje de tipo **REQUEST** para realizar una acción concreta en un tiempo determinado. La segunda con la creencia de que el agente emisor del mensaje es amigo, que el agente receptor puede realizar la acción y que además no está obligado a realizar otra acción previamente al mismo tiempo. Una vez que se cumplen ambas, el agente

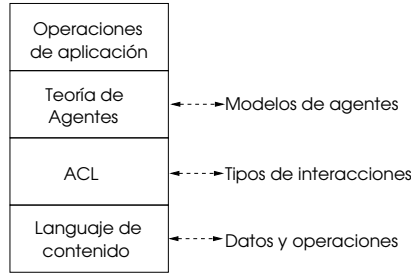


Figura 1: Capas de construcción de un sistema de agentes

queda obligado a realizarla. Este primer lenguaje ha inspirado a muchos trabajos posteriores. Vamos a ver alguno de ellos, identificando primero los elementos necesarios en todo lenguaje de programación de agentes.

Si seguimos las indicaciones de [27], admitiremos que la construcción de un determinado sistema software basado en agentes depende de los siguientes hechos:

- La construcción de un agente particular está basada en una teoría de agentes concreta ya sea esta inspirada semántica ó sintácticamente. Ésta se va a definir por una teoría de agentes, una arquitectura concreta, y un lenguaje de programación con el que especificarlos.
- Una tecnología de agentes concreta se caracteriza, entre otros elementos, por un ACL (*Agent Communication Language*) y un lenguaje de contenidos concreto (e.g. KIF, HTML, etc).
- Los modelos usados para planificación y *scheduling* intervienen en la organización de los modelos de agentes.

En la figura 1 podemos ver una representación por capas de los niveles de desarrollo de agentes. En la capa básica podemos ver el *lenguaje de contenido*, con el cual podemos expresar tanto los datos como las operaciones básicas propias de la aplicación (i.e. ontología). Posteriormente, la siguiente capa corresponde al ACL usado. Éste transporta los contenidos de los mensajes entre agentes, realizándose de este modo las interacciones correspondientes. Sobre el ACL se asienta la capa *teoría de agentes* a través de la cual obtenemos los agentes concretos que responden a modelos determinados a través del trinomio *teoría + arquitectura + lenguaje*. Finalmente, sobre los agentes podemos construir la propia aplicación.

Es interesante hacer notar que, desde el punto de vista del mantenimiento de un sistema de información cuyo desarrollo se ha ajustado a este esquema, la cantidad de cambios potenciales debidos al mantenimiento están también estratificados. Si el modelo de agente se ha escogido con cuidado, este nivel de definición de la aplicación va a ser el que necesite menos cambios. Por otro lado, éstos serán importantes en cuanto al coste necesario para realizarlos. Así mismo, los tipos de interacciones identificadas podrían, eventualmente, necesitar más cambios que los realizados en el modelo de agente. Por ejemplo, el añadir o eliminar un tipo de mensaje (i.e. *performativa*) implicará el modificar el parser de los mensajes y añadir o eliminar funcionalidades. El mayor número de cambios se realizará a nivel del lenguaje de contenidos al estar más cercano al dominio propio de la aplicación en donde, usualmente, se produce un mayor número de cambios en el desarrollo y mantenimiento propios de un proyecto software.

3 Teorías, arquitecturas y lenguajes de agentes

Las dos grandes familias clásicas de agentes son la de los deliberativos y la de los reactivos. Un agente deliberativo es aquel que “*contiene un modelo simbólico del mundo representado internamente de forma*

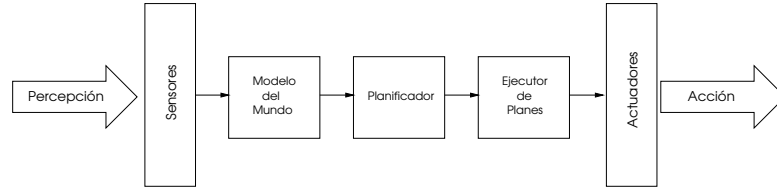


Figura 2: Arquitectura básica de un agente deliberativo.

explícita y que toma decisiones mediante razonamiento lógico [34]. Por otro lado, un agente reactivo es aquel en que *“no incluye un modelo simbólico del mundo ni usa razonamiento simbólico complejo de ningún tipo”* [34]. En los dos apartados siguientes veremos estos dos tipos de agentes, desde un punto de vista ingenieril.

3.1 Programación de agentes BDI

Los agentes deliberativos usan un modelo del mundo, explícitamente representado y funcionan siguiendo el paradigma de los sistemas clásicos planificación de la IA, basado en el ciclo percepción-planificación-acción. Podemos ver la arquitectura genérica representada en el diagrama estructural de la figura 2. En ella aparece representado de forma explícita el mundo, o la visión parcial que pueda tener de él. Aparecen también el planificador, en donde se encuentra toda la lógica deliberativa del mismo, y el ejecutor de los planes que transforma las decisiones del módulo anterior en acciones.

El modelo de agente más representativo del tipo de los deliberativos es el BDI (*Belief-Desire-Intention*). Estos agentes se caracterizan por tener “ciertas actitudes mentales” [28] como son las creencias, deseos e intenciones.

Las **creencias** corresponden al conocimiento que el agente posee sobre el resto del mundo. Desde el punto de vista de la implementación, las creencias pueden ser desde un conjunto de variables hasta una base de datos, pasando por un conjunto de expresiones lógicas. En definitiva, cualquier estructura de datos.

Por otro lado, los **deseos** se refieren a cómo se ordenan, en términos de prioridades o coste, los múltiples objetivos que tiene actualmente el agente. Así, podemos decir que guían las motivaciones de éste. Podemos verlos como una estructura de programación que ordena los objetivos, entendidos estos como una representación concreta de un estado del mundo similar estructuralmente a la de las creencias.

Finalmente, recordemos que un agente es una entidad software situada en su entorno que actúa en él esperando modificarlo en algún sentido. Una vez que se actúa sobre el entorno, la acción escogida determina la intención que inmediatamente manifiesta el agente. La parte de las **intenciones** es la que representa la estrategia de acción que el agente está siguiendo actualmente. Por ello, es posible que las intenciones actuales estén sujetas a reconsideración. Desde el punto de vista de la programación, se trata de representar la última acción o secuencia de acciones llevadas a cabo en el entorno.

La estructura de código de agente DBI clásica de Rao y Georgeff [28] consiste en un intérprete de ciclo infinito, que accede a tres estructuras para leer y escribir en ellas como son las de creencias, deseos e intenciones. El ciclo comienza estudiando los eventos obtenidos a la entrada. Este estudio genera un conjunto de opciones usando para ello las creencias y deseos. A partir de un proceso de deliberación se seleccionan aquellas opciones que serán llevadas a cabo. Se transformarán así en intenciones. Una vez hecho esto, las intenciones se llevan a cabo (i.e. se ejecutan) y se obtienen los eventos que el sistema genera a partir de su ejecución. Posteriormente, se eliminan tanto los deseos satisfechos como los no alcanzables. Lo mismo se hace con las intenciones. Tras esto el ciclo vuelve a comenzar. Podemos ver su representación procedural, extraída de [28], en la figura 3.1.

```

BDI-interpretor
initialize-state();
repeat
  options := option-generator(event-queue);
  selected-options := deliberate(options);
  update-intentions(selected-options);
  execute();
  get-new-external-events();
  drop-successful-attitudes();
  drop-impossible-attitudes();
end repeat

```

Figura 3: Estructura en forma de código del intérprete BDI propuesto por Rao y Georgeff.

Pero también podemos especificar agentes BDI usando editores visuales. Cuando hablamos de *editores BDI* [30], como los llaman Ricordel y Demazeau, nos referimos a herramientas visuales como las que incluyen AgentBuilder [29] y Zeus [6]. Por ejemplo, el editor de agentes de Zeus incluye cuatro definiciones que componen el agente en su conjunto:

- Definición del agente: en donde se especifica las tareas, recursos disponibles y capacidad de planificación.
- Definición de tareas: en donde se especifican las funcionalidades del agente.
- Organización del agente: en donde se define el *contexto social* del agente. Básicamente se trata de especificar qué agentes conoce (i.e. sus *acquaintances* [18]).
- Coordinación del agente: en donde se indica para cada agente qué interacciones es capaz de desempeñar.

Todo esto se especifica visualmente, con un editor diferente para cada uno de los cuatro módulos. Los editores BDI son muy sencillos e intuitivos a la hora de definir los agentes. Por otro lado, al obligar al programador a especificar cada uno de los items de los cuatro grupos, el modelo de agente que hay detrás se hace explícito y esto redundante en la sencillez de manejo y en un rápido prototipado de agentes sencillos. Por otro lado, el corsé que se impone al programador en este tipo de enfoque de programación es muy severo.

Otro medio para la programación de agentes BDI es usar un ADL (*Agent Definition Language*). A partir de ese ADL, usamos un traductor que convierte el código a un determinado lenguaje de alto nivel. Por ejemplo, JACK [4] sigue ese enfoque haciendo uso de Java como lenguaje de alto nivel. Se trata de añadir palabras clave y marcas para, programando sobre Java, definir mensajes, planes, agentes, etc. Su programación está basada en eventos. Los eventos “*motivan a los agentes a realizar acciones*” [1]. Los hay de tres tipos y se denominan eventos internos, externos y motivaciones. Los internos gobiernan la ejecución del agente, los externos se refieren básicamente a mensajes entrantes de otros agentes o estímulos del entorno. Finalmente, las motivaciones se refieren al disparo de la consecución de objetivos que el agente pueda tener. Por ejemplo, el fragmento de código siguiente

```

event PingEvent extends MessageEvent {
  int value;
}

```

```

    #posted as ping(int value)
    {
        this.value = value;
    }
}

plan BouncingPlan extends Plan {
    #handles event PingEvent ev;
    #sends event PingEvent pev;

    body()
    {
        @send( ev.from, pev.ping(ev.value + 1) );
    }
}

agent PingAgent extends Agent {
    #handles event PingEvent;
    #uses plan BouncingPlan;
    #posts event PingEvent pev;

    void ping (String other)
    {
        send(other, pev.ping(1));
    }
}

```

define un agente, por medio de la especificación de los mensajes que recibe, y los planes que lleva a cabo. Los mensajes a recibir se definen a través de eventos. Cuando un mensaje se recibe, se detecta a partir de la recepción de un evento. El definir un plan consiste en especificar qué genera la ejecución del mismo (un evento en este caso) y el cuerpo del mismo. Cuando se llama el método `ping(String)` sobre el agente, se genera un evento del tipo `PingEvent` que recibe el agente destino.

3.2 Construcción de agentes reactivos

Agentes reactivos, como se hacía notar arriba, no presentan representación explícita de conocimiento simbólico complejo. Se trata de ofrecer respuestas inmediatas a estímulos del entorno. A estos agentes Nilsson los denomina también agentes estímulo-respuesta [26]. Se trata de agentes que tienen una percepción concreta a partir de la cual se disparan las acciones pertinentes, dependiendo de las condiciones activadas en el proceso perceptivo. De entre las técnicas básicas que propone Nilsson para la construcción de este tipo de agente nos quedamos con la que hace uso de sistemas de producción al ser los más usados. Un sistema de producción es un conjunto ordenado de reglas de producción ó producciones del tipo $c \rightarrow a$ en donde c se refiere a la parte de las condiciones y a se refiere a la acción pertinente a ejecutar. Su funcionamiento más simple evalúa por orden el conjunto de reglas y cuando en una de esas reglas se evalúa el antecedente a 1 se ejecuta la acción del consecuente. Así, los objetivos más importantes del agente corresponden a los antecedentes de las reglas que están más arriba en el sistema de producción. Si representamos la arquitectura típica de un agente reactivo, tendremos el diagrama estructural de la figura 4. Un agente de este tipo se puede codificar usando cualquier paquete que implemente un mecanismo de ACL y un sistema de reglas de producción. Por ejemplo, podemos usar JATLite [23] y Jess [17]. El primero es una infraestructura de comunicaciones a través de un ACL que proporciona un mecanismo de intercambio de mensajes a las entidades software que lo utilizan, que han de estar programadas en Java. El segundo es un conjunto de clases Java a través de las cuales podemos manejar sistemas de producción.

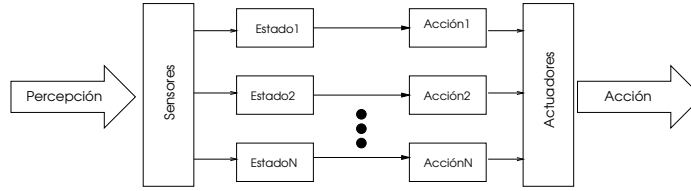


Figura 4: Diagrama estructural de la arquitectura de un agente reactivo.

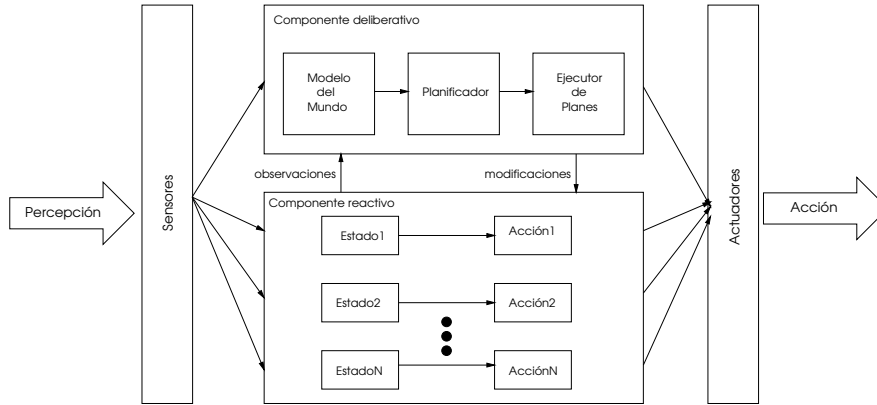


Figura 5: Diagrama estructural de la arquitectura de un agente híbrido.

3.3 Arquitecturas híbridas

Como hemos podido comprobar, las arquitecturas deliberativas y reactivas presentan diferencias sustanciales. Aun así, en la literatura podemos encontrar un enfoque híbrido que combina aspectos de las dos arquitecturas para dar lugar a una arquitectura híbrida, como la que aparece en la figura 5. Como puede comprobarse, un agente híbrido típico dispone de componentes deliberativos que permiten realizar razonamientos complejos, realizar planes y tomar decisiones. Todo esto en combinación con componentes reactivos que permiten la reacción inmediata ante eventos para los cuales es necesario ese tipo de reacción.

El ejemplo paradigmático de arquitectura híbrida es InteRRaP de Muller [25]. Otros podrían ser las Touring Machines de Ferguson [8] y el sistema PRS (*Procedural Reasoning System*) [19]. En la figura 5 podíamos apreciar ya que los componentes se disponen en capas para su organización. Los componentes reactivos se encuentran conectados a los sensores y actuadores. Sin embargo los deliberativos están conectados a los reactivos. Pues bien, en InteRRaP esto se hace aun más patente al poder encontrarse tres capas: la de comportamiento, la de planificación y la de cooperación, citadas en orden de disposición. Cada una de ellas maneja un tipo de información, acorde al nivel de abstracción en el que se sitúan. Así, en la capa de comportamiento podemos encontrar una capacidad de decisión inmediata al trabajar con un modelo del mundo y estar en contacto sensores y actuadores. Por otro lado, en la de planificación podemos encontrar el comportamiento deliberativo que maneja conocimiento necesario para planificación. En la capa del último nivel encontramos el comportamiento que da sociabilidad al agente y que maneja conocimiento de su entorno social.

Las tecnologías que permitan trabajar con arquitecturas híbridas son raras. De hecho los autores no han encontrado una plataforma accesible que nos permita trabajar con agentes híbridos de forma explícita por lo volveremos a ellos en el resto del trabajo.

3.4 Construcción de agentes *a la* FIPA

En esta sección vamos a introducir un enfoque a la construcción de agentes basado en estándares, publicados por una organización internacional sin ánimo de lucro, dedicada a establecer un marco común y genérico tanto para la construcción de agentes inteligentes como de sistemas multi-agente. Con esto se consigue interoperabilidad entre sistemas de agentes construidos por diferentes equipos. Nos referimos a FIPA (*Foundation for Intelligent Physical Agents*)¹.

De entre los documentos más importantes que publica esta organización podemos encontrar el que define la “*Arquitectura Abstracta FIPA*” [16]. Su objetivo es, tal y como reza en el documento, “*el permitir la construcción de sistemas que se integren con su entorno de computación particular, mientras que interoperan con sistemas de agentes que residen en entornos heterogeneos, todo con el mínimo esfuerzo*”.

Para construir un agente que cumpla las especificaciones FIPA, podemos seguir dos enfoques distintos. Como primer enfoque podemos usar el de construir un agente FIPA desde cero, basándonos en las especificaciones pertinentes. Para ello necesitamos ser unos perfectos conocedores de los documentos FIPA implicados, y además disponer de unos recursos de programación nada despreciables. El enfoque más acertado trata de apoyarse en una plataforma que cumpla los estándares FIPA, para así construir el sistema que necesitamos. Con este enfoque cedemos libertad creadora a cambio de una mayor productividad. Un ejemplo de plataforma FIPA no comercial es JADE (*Java Agent DEvelopment Framework*) [2]. En realidad JADE consiste en un par de herramientas: una plataforma de agentes y un entorno de monitorización. La primera es imprescindible para ejecutar cualquier agente usando las *librerías* que provee la herramienta. El entorno de monitorización es una herramienta conveniente para un fácil seguimiento de la ejecución.

3.4.1 El agente básico

La manera más sencilla e inmediata de construir un agente JADE es heredando de una clase básica Java denominada `jade.core.Agent`. Ésta clase incluye los métodos

```
public void setup()
public void action()
```

que para el programador son los más importantes inicialmente. El primero ha de realizar todas las tareas necesarias para que el agente pueda comenzar su ejecución con todo lo necesario. El segundo es realmente el cuerpo de ejecución del agente. Hasta aquí, podríamos pensar que el modelo de agente que existe detrás de la plataforma JADE es simplista, pero no es así. El comportamiento del agente está determinado por el AFD que aparece en la figura 6 y que aparece también en el documento FIPA [10]. En la figura podemos ver que cuando el agente se crea pasa a un estado de iniciado. Posteriormente, cuando la plataforma invoca su método `action()` pasa a estar activo y de ahí puede pasar a suspendido, esperando un determinado evento ó en tránsito (moviéndose de un entorno de ejecución JADE a otro).

Un ejemplo de código para crear un agente mínimo en JADE es el que podemos ver a continuación:

```
import jade.core.*;

public class HelloWorldAgent extends Agent
{
    public HelloWorldAgent()
    {}

    public void setup()
    {}
}
```

¹<http://www.fipa.org>

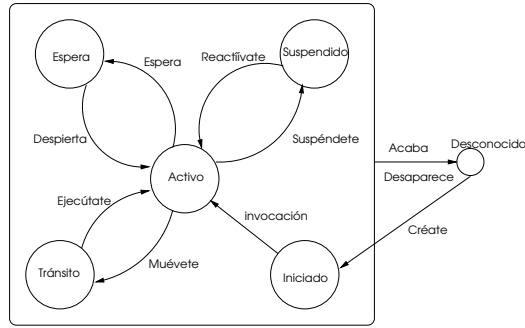


Figura 6: AFD que define el comportamiento de un agente JADE

```

public void action()
{
    System.out.println("Hello World");
}
}

```

3.4.2 Enviando y recibiendo mensajes

Hasta ahora no habíamos hablado del aspecto de comunicación agente-agente. Un agente JADE puede mandar mensajes del tipo `jade.lang.acl.ACLMessage`. Un mensaje perteneciente a esta clase puede ser transformado en un flujo de bits que representan la estructura de mensajes FIPA. Básicamente, un mensaje FIPA contiene los siguientes campos [9]:

- **performative**: indica el tipo de acto comunicativo del mensaje.
- **sender**: el emisor del mensaje.
- **receiver**: el receptor del mensaje.
- **reply-to**: el receptor de la réplica al mensaje.
- **content, language, encoding, ontology**: el contenido del mensaje, el lenguaje en el que va representado, la codificación y la ontología a la que pertenecen los datos del mismo, respectivamente.
- **protocol**: el protocolo de conversación subyacente entre los agentes implicados.
- **conversation-id**: identificador de la conversación particular a la que pertenece el mensaje.
- **reply-with**: identificador de mensaje que ha de llevar la réplica a este.
- **in-reply-to**: identifica la réplica con respecto al mensaje original.
- **reply-by**: dato de tiempo o fecha, dentro de la cual al agente emisor desearía recibir la réplica.

El siguiente fragmento de código, en el que aun no hemos tenido en cuenta los detalles relativos a la localización de otros agentes en el sistema, ilustra cómo crear y enviar de forma sencilla un mensaje en JADE:

```

ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
msg.setOntology('PC-ontology');
msg.setLanguage('Jess');
msg.addReceiver(pcseller);
msg.setContent('(pc-offer (mb 256) (processor celeron) (price ?p))');
send(msg);

```

Inicialmente se crea un mensaje con la intención de hacer una petición simple (`query-if`) [14]² a otro agente, a partir de la cual podrá respondernos que (a) no entiende el mensaje (`not-understood`), (b) rechaza realizar la petición (`refuse`), (c) ha habido un fallo al intentar elaborar la respuesta (`failure`) ó (d) la respuesta (`inform`).

Obviamente, un agente no solo debe ser capaz de enviar mensajes. También debe poder recibirlos en una forma adecuada. Todo agente tendrá unos cometidos determinados que, junto con el estado en el que estén las posibles conversaciones que mantenga en un momento determinado, van a condicionar los tipos de mensajes que está interesado en recibir en un momento determinado. Para ello, en JADE disponemos de un mecanismo interesante basado en plantillas de mensajes. Básicamente, se definen unas plantillas que especifican un conjunto de mensajes que el agente está interesado en recibir. El siguiente fragmento de código lo ilustra con un ejemplo:

```

MessageTemplate t1 = MessageTemplate.MatchPerformative(ACLMessage.QUERY_IF);
MessageTemplate t2 = MessageTemplate.and(t1, MessageTemplate.MatchOntology('PC-ontology'));
MessageTemplate t3 = MessageTemplate.and(t2, MessageTemplate.MatchLanguage('Jess'));
ACLMessage msg = blockingReceive(t3);

```

Este fragmento, que perfectamente podría aparecer en el cuerpo del método `action()` de un agente que eventualmente reciba el mensaje creado arriba, va creando plantillas que se van especializando sucesivamente hasta crear la más específica `t3`. Ésta incluye a todos los mensajes que tienen como performativa `QUERY_IF`, la ontología correspondiente es “PC-ontology” y el lenguaje que aparece en el contenido del mensaje es Jess. Un detalle interesante es la llamada `Agent.blockingReceive(MessageTemplate)`. Ésta causa que el flujo de ejecución del agente se detenga hasta que se reciba un mensaje que cumpla la plantilla pasada como argumento ³.

3.5 Construcción de sociedades de agentes

Llegados a este punto, conocemos varias maneras para realizar agentes individuales, siguiendo las dos corrientes principales: la deliberativa y la reactiva. Ahora bien, ¿cómo atacamos el problema de la programación de los mecanismos que hacen que los agentes cooperen entre sí para conseguir sus objetivos de una forma coordinada? Vamos a ver los enfoques seguidos en el editor de agentes Zeus (apartado 3.5.1), en la plataforma BDI JACK (apartado 3.5.2) y siguiendo el estándar FIPA (apartado 3.5.3).

3.5.1 Relaciones en Zeus

Dentro de Zeus, el editor BDI mencionado en el apartado 3.1, hay una fase de definición de cada agente correspondiente a su organización dentro del sistema en la que, además de indicarse sus habilidades, se indican sus agentes conocidos (i.e. *acquaintances*) [6]. Cada uno de sus conocidos se especifica junto con el tipo de relación que mantiene con él:

- Peer: relación sin ninguna suposición respecto al tipo.

²Existe un conjunto de interacciones típicas entre dos o más agentes, también estandarizado por FIPA, denominadas *Especificaciones de protocolos de interacción*.

³Esto no es del todo cierto. Existe la posibilidad de que el agente esté ejecutando varias tareas en forma concurrente. Así, si este es el caso, el resto de tareas concurrentes seguirán su ejecución normalmente.

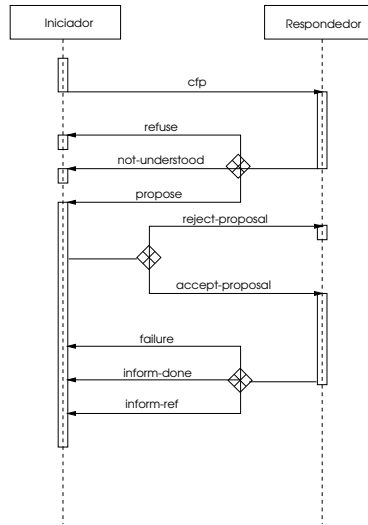


Figura 7: Diagrama de interacción que define el protocolo CNP.

- Superior: el conocido tiene mayor autoridad que el agente que se está definiendo, por lo que puede darle órdenes que éste tiene que obedecer.
- Subordinate: el contrario al anterior, siendo este agente capaz de dar órdenes que el conocido debe obedecer.
- Co-worker: el agente conocido y el que se define pertenecen a la misma comunidad de agentes. Esto implica que antes de demandar recursos a un agente *peer*, se le pedirán a agentes de la misma comunidad.

Posteriormente, dentro de la fase de definición de la coordinación, se indican los *protocolos de coordinación* y las *estrategias de interacción*.

En Zeus existe un único tipo de protocolo de coordinación hasta el momento, el *Contract Net Protocol* (CNP) [32]. En realidad, este es un protocolo genérico que permite negociar a varias entidades con dos roles diferentes, el de iniciador (un único agente) y el de pujador (tantos como estén interesados en la oferta del iniciador). Su funcionamiento se puede explicar resumidamente como sigue: una entidad necesita un determinado recurso, y prepara una petición del mismo indicando el tipo de recurso y la envía a quien pueda interesar. Las entidades que reciben la petición, determinan si pueden hacer una oferta del recurso solicitado y en qué condiciones. Preparan la oferta, dentro de un tiempo límite que especifica el iniciador de la negociación, y la envían. Una vez el iniciador recoge todas las ofertas, determina cual es la que le interesa y lo notifica a la entidad correspondiente, a partir de lo cual se pone a trabajar en la misma. Podemos ver una representación en forma de diagrama de interacción del protocolo en la figura 7 reproducida a partir de la que puede encontrarse en la especificación FIPA [11] en base a las performativas utilizadas. Pues bien, el indicar el protocolo de coordinación para un agente consiste en indicar si el mismo va a ser iniciador o *responder* (i.e. el agente que responde a la oferta), o las dos cosas.

Una vez que se especifica que el agente es capaz de emplear un protocolo de coordinación determinado, se ha de indicar cómo ha de emplearlo. En base a qué estrategia debe usarlo. La plataforma Zeus permite especificar un uso concreto de un protocolo dependiendo del tipo de hecho con el que se esté negociando. Además, es posible especificar con cuáles de los conocidos del agente en cuestión ha de usarse la estrategia o bien qué se ha de realizar con todos aquellos que presenten una relación de un tipo determinado (i.e.

Peer, Superior, Subordinate o Co-worker). Una vez hecho esto, se especifica la estrategia, que previamente tiene que estar programada en Java y añadida a Zeus.

Como puede verse, todo en Zeus se especifica a un nivel de abstracción bastante interesante para un programador que quiere iniciarse en la programación de MAS. Por otro lado, los agentes Zeus son, además, FIPA.

3.5.2 SimpleTeams en JACK

Una de las teorías que estudia los problemas derivados del trabajo en equipo mediante sociedades de agentes es la denominada de *TeamWork*. Esta teoría se crea a partir de un conjunto de agentes BDI e intenta modelar los mecanismos mediante los cuales un conjunto de agentes mantiene un conjunto de acciones que los llevan a conseguir objetivos individuales. La idea básica es la de intencion colectiva [5]. No es más que una propiedad que puede manifestar un equipo de agentes a través de la cual emerge el comportamiento en equipo.

Este modelo podemos verlo adoptado en JACK [3]. En esta infraestructura para el desarrollo y estudio de sistemas de información distribuidos basados en agentes BDI, podemos encontrar una extensión denominada *SimpleTeams* [24]. Esa extensión usa *Programación Orientada a Equipos* para especificar coaliciones o equipos, su funcionalidad, miembros, mecanismos de coordinación entre miembros y conocimiento compartido. Así, la declaración de un equipo se hace siguiendo la siguiente plantilla:

```
team IdEquipo extends Team {
    #performs role IdRole;
    #requires role IdRole referencia_role;
}
```

La declaración de un equipo es análoga a la de una clase en Java, aunque solamente basada en herencia a partir de el equipo básico `Team`. La línea de código correspondiente a la directiva `#performs` indica la funcionalidad de el equipo, a través de la indicación del rol que puede seguir (i.e. `RoleType`). A su vez, la línea de código de la directiva `#requires` se refiere a qué miembros va a tener el equipo. Estos se determinarán dinámicamente al especificarse simplemente qué funcionalidades (i.e. roles) se necesitan de los agentes/roles para pasar a formar parte del mismo. Dentro del código de la clase `team` se les referenciará mediante la referencia local `referencia_role`.

3.5.3 Coordinación en JADE

Hablar de coordinación en JADE es hablar de los mecanismos que establece FIPA, y la plataforma incorpora adaptados, a su conjunto de herramientas. La coordinación en JADE se sustenta, sobre todo, en dos elementos. El primero consiste en un servicio de directorio de agentes (*Agent Directory Services*) y un servicio de directorio de servicios (*Services Directory Services*). El segundo es el conjunto de los protocolos de coordinación que cada agente puede desempeñar.

Directorio de agentes y servicios

Existen dos situaciones diferentes en las cuales se puede necesitar obtener la localización de un agente. La primera se da cuando, sabiendo el nombre de un agente, no sabemos su localización en el sistema. La otra se presenta cuando un agente necesita invocar un servicio concreto en otro agente y, en principio, no sabemos qué otro agente puede ofrecérselo.

Para hacer frente a la primera situación, todo agente FIPA [16] debe darse de alta en un directorio junto con una dirección⁴ en donde poder localizarlo. Así, otro agente que desee contactar con él podrá, accediendo al directorio, obtener su localización.

⁴Obviamente, la dirección ha de ser de transporte y haciendo uso del servicio de directorio de agentes. Si se está utilizando una red IP, esta habrá de ser la correspondiente dirección IP ó DNS junto con el puerto en donde escucha.

De la misma forma, un agente que quiere ofrecer sus servicios ha de subscribirse al directorio de servicios mediante el servicio de directorio de servicios. Los datos de su subscripción incluirán un identificador único globalmente, el tipo de servicio dentro de una categoría y un conjunto de pares atributo-valor describiendo el acceso al servicio y otras características de interés.

El siguiente fragmento de código ilustra como un agente que ofrece un servicio de concatenación de cadenas de caracteres, se suscribe con una descripción del mismo en el facilitador de directorio.

```
public class ConcatAgent extends Agent
{
    ...

    public void setup()
    {
        ...
        ServiceDescription s = new ServiceDescription();
        s.setName('ConcatStrings');
        s.setType('Strings');
        Property p = new Property();
        p.setName('MaxLengthLeftString');
        p.setValue('25');
        s.addProperties(p);
        p.setName('MaxLengthRightString');
        p.setValue('25');
        s.addProperties(p);
        DFAgentDescription df = new DFAgentDescription();
        df.addServices(s);
        try{ DFService.register(this,df); }catch(FIPAException e){ e.printStackTrace(); }
        ...
    }
    ...
}
```

Obsérvese que el concepto de servicio es algo abstracto y que FIPA no trata de estandarizar los mecanismos para su invocación completa. La invocación de un servicio puede estar implícita en un acto comunicativo como por ejemplo en `query-if`. Es en el cuerpo ontológico utilizado en donde debería quedar definido plenamente cada uno de los servicios usados. Cuerpo ontológico que deberían conocer tanto el agente que ofrece el servicio como el que lo invoca.

Protocolos de negociación JADE

Una de las propiedades que distingue a los agentes de los objetos es que pueden realizar interacciones complejas, en lugar del simple paso de mensajes que realizan estos [22]. Estas interacciones complejas se denominan *protocolos de interacción* en el contexto de FIPA. El ejemplo típico es el CNP del que hemos hablado en la sección 3.5.1. Pero FIPA también ha estandarizado las subastas inglesa [13] y alemana [12].

Las interacciones básicas FIPA pertenecen a la familia denominada *achieve rational effect* que se refiere a la asimilación por parte del agente receptor de la interacción, de un efecto racional deseado por el agente que inicia la interacción. Protocolos FIPA de este tipo son, por ejemplo, FIPA-query, FIPA-propose, FIPA-brokering, etc. Su funcionamiento básico es similar en todos. El agente iniciador envía un mensaje. Posteriormente el receptor (*responder*) responde bien con un `not-understood`, bien con un `refuse` ó `agree` con lo que proporciona una respuesta inicial a la petición. Una vez que el receptor ha llevado a cabo la acción, envía un último mensaje al iniciador del tipo `inform` ó `failure` para notificar el resultado. Los diseñadores de JADE han incluido dos clases `jade.proto.AchieveREInitiator` y

`jade.proto.AchieveREResponder` que asisten en la programación de interacciones basadas en estos protocolos básicos. Usando el ejemplo de código que aparece en el manual (ver página 37 de [2]) podemos ver la programación del código relativo a un agente iniciando una interacción del tipo FIPA-request [15].

```

ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
request.setProtocol(FIPAProtocolNames.FIPA_REQUEST);
request.addReceiver(new AID('receiver', AID.ISLOCALNAME));
addBehaviour(new AchieveREInitiator(myAgent, request) {
    protected void handleInform(ACLMessage inform) {
        System.out.println("Protocol finished. Rational Effect achieved. Received " + inform);
    }
});

```

En este fragmento de código introducimos un elemento nuevo, el `behaviour` de un agente JADE. Inicialmente creamos el mensaje e indicamos que el protocolo va a ser un FIPA-request. Le añadimos el receptor y añadimos al agente un comportamiento `AchieveREInitiator`. Un comportamiento es una abstracción usada para modelar el flujo de ejecución de un fragmento de código. Así, podemos tener comportamientos simples (i.e. `SimpleBehaviour`) que se refieren a un código que se ejecuta secuencialmente o comportamientos en paralelo (i.e. `ParallelBehaviour`) que a su vez está compuesto de dos o más comportamientos que se ejecutan en paralelo, etc. En particular, el comportamiento `AchieveREInitiator` es del tipo `FSMBehaviour`, cuyo flujo de ejecución está modelado por un AFD definido por el programador de tal manera que cada estado se refiere a un comportamiento embebido dentro del comportamiento compuesto. El AFD, en este caso, se refiere al protocolo desde el punto de vista del agente que inicia la conversación. Podríamos haber definido nuestro propio comportamiento como una subclase de `AchieveREInitiator` redefiniendo después los métodos `handleXXX` en donde `XXX` se refiere a las performativas que pueden recibirse en la parte del agente que inicia la conversación. Sin embargo, en este ejemplo evitamos el tener que crear una clase nueva, creando el nuevo comportamiento de forma implícita y redefiniendo directamente el método `handleInform()`. Por supuesto, el propio comportamiento es el que se encarga de redirigir los mensajes a los métodos adecuados. El programador únicamente debe encargarse de reprogramar cada uno de los manejadores de acuerdo con sus necesidades.

El siguiente fragmento de código haría las veces de *responder* de la interacción FIPA-request:

```

MessageTemplate mt = AchieveREResponder.createMessageTemplate(FIPAProtocolNames.FIPA_REQUEST);
myAgent.addBehaviour(new AchieveREResponder(myAgent, mt) {
    protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage response)
    {
        System.out.println("Responder has received the following message: " + request);
        ACLMessage informDone = request.createReply();
        informDone.setPerformative(ACLMessage.INFORM);
        informDone.setContent("inform done");
    }
});

```

En este caso, no hemos de crear ningún mensaje inicialmente sino una plantilla (ver apartado 3.4.2) con la que se seleccionan los mensajes de interés, en este caso los que se refieren a una interacción del tipo FIPA-request y que, además, solo puede recibir el *responder* de la interacción. Esta funcionalidad nos la proporciona un método estático de la clase `AchieveREResponder`. Esta clase funciona como su análoga en el *initiator*. Se define mediante un AFD que gobierna la transición entre sus diferentes comportamientos embebidos dentro de uno del tipo `FSMBehaviour`. El comportamiento que se va a encargar del manejo de los mensajes entrantes únicamente redefine el método `prepareResultNotification()`. Recuerdese que podemos responder al `request` con un `not-understood`, un `refuse` ó un `agree` si es que accedíamos a realizar la petición. Sin embargo, si el *initiator* recibe un `inform` significa tanto que accedemos a realizar la petición como que el resultado va en el propio mensaje, y por lo tanto es redundante el `agree`.

El otro tipo de interacción que viene asistido por este tipo de clases es el del CNP, y funciona en una forma muy similar a las anteriores.

4 Herramientas para la implementación

En este apartado reseñaremos qué tipos de herramientas son necesarias no ya para la programación sino para la depuración y mantenimiento de las aplicaciones de agentes de entre las que podemos encontrar.

4.1 IDEs (*Integrated Development Enviroment*)

Un entorno de desarrollo integrado es una herramienta que permite la implementación de proyectos para lenguajes de alto nivel, como por ejemplo VisualC++ de Microsoft o Forte para Java. Estas aplicaciones incluyen potentes editores de texto, gestores de proyectos para el manejo de un gran número de ficheros de código, compiladores para el lenguaje, entornos de depuración y ejecución para la realización de todo tipo de pruebas del software.

Respecto a JACK, esta plataforma incluye su propio IDE que incluye un editor de texto adecuado a las características propias del lenguaje Java extendido usado para la programación de agentes. Además, incluye un compilador para este lenguaje que primero transforma a Java el código y posteriormente una el compilador para Java puro produciendo así los *bytecodes* que ya pueden ser interpretados. En el conjunto de ventanas de compilación del IDE puede encontrarse adjunto un entorno de ejecución de la aplicación JACK ó bien cualquier otra aplicación Java. No existe posibilidad de depurar código con este IDE aunque sería posible, una vez se tiene todo el código Java, utilizar cualquier otro para este propósito.

Zeus es un editor BDI y por lo tanto, en ningún momento se tiene la necesidad de manejar código directamente, aunque realmente el editor BDI esté trabajando en la trastienda con un código Java que responde a un modelo de agentes FIPA. Por lo tanto esto va a condicionar decisivamente el que no exista un compilador ó depurador de código directamente accesible al programador.

El caso de JADE distinto al de los dos anteriores. Con JADE debemos programar los agentes usando código 100% Java. Por lo tanto, a la hora de la programación podemos usar cualquier IDE Java que exista en el mercado (o simplemente un editor de texto para proyectos pequeños). Con esto podremos editar, compilar, depurar y ejecutar agentes JADE. Pero siempre trabajando en el nivel de abstracción correspondiente al código Java.

4.2 Depuración y Mantenimiento de código

A poco que el lector haya experimentado un proceso de depuración de código admitirá que es un proceso complejo. Si se tiene en cuenta que un MAS es un sistema de software distribuido, la depuración en este tipo de sistemas se hace mucho más difícil y compleja. Los primeros en señalar que este tipo de herramientas resultas imprescindibles en la programación de un MAS fueron Liedekerke y Avouris [33]. La principal razón que argumentan para esa complejidad es que, en tiempo de ejecución, al desarrollador le resulta imposible tener una imagen global correcta y exacta de lo que está ocurriendo en el sistema. Más aun, debido a que existen diferentes fuentes de eventos que monitorizar, éstos solo pueden ser correctos en un conjunto si llevan una marca de tiempo obtenida de una fuente global.

Los creadores de Zeus [7] dividen los errores de un MAS en estructurales y funcionales. De entre los errores funcionales podemos encontrarnos algunos como relaciones entre agentes erroneas, recursos no especificados, granularidad en las unidades de tiempo en las que se especifica la duración de las tareas, etc. Los funcionales están relacionados con la lógica de las actividades en las que los agentes se ven inmersos. Incluso, existe la posibilidad de que aun siendo los agentes tanto estructural como funcionalmente correctos, el sistema multi-agente no funcione como se esperaba, dando lugar a errores de coordinación. Por lo tanto, a la hora de depurar este tipo de sistemas, hay que actuar a esos tres niveles.

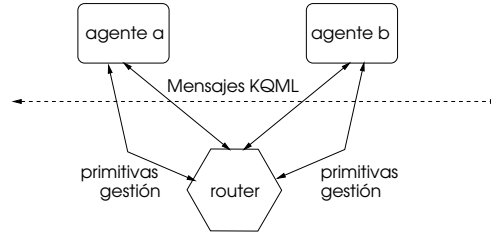


Figura 8: Diagrama estructural de la arquitectura basada en router de mensajes de JATLite.

Para el desarrollo y la depuración de programas no es solamente útil el prestar atención a los mensajes de error que se producen a partir de un paso de mensaje simple. También es útil el habilitar algún mecanismo que sea capaz de almacenar, de algún modo, todas las interacciones que se llevan a cabo entre dos o más entidades de interés. En el contexto de un MAS, si se tiene la posibilidad de almacenar todas las interacciones realizadas en, por ejemplo, una subasta tendremos capacidad no solo para detectar fallos en la programación, sino para verificar la integridad de los protocolos y mejorar estrategias de negociación para que nuestros agentes sean aun más eficientes. Para ello debemos colocar un elemento software intermediario a través del cual pasen todos los mensajes emitidos en el sistema y que, posteriormente, pueda mostrárnolos adecuadamente según nuestras necesidades.

Un ejemplo de este enfoque es el de JATLite [23], un conjunto de herramientas que proporcionan una infraestructura para el desarrollo de agentes con dos características principales: heterogeneidad en su construcción e independencia de localización física del agente. Para todo esto usa un intermediario denominado *router* de mensajes de agentes. El esquema estructural podemos verlo en la figura 8. En ella se aprecia que todos los mensajes pasan a través del router que, además de redirigirlos al destinatario correspondiente, si este no está conectado en ese momento, hace uso de sus buffers para mantenerlo allí hasta que sea posible enviarlo.

En FIPA se sigue un enfoque parecido dentro de la arquitectura. En la arquitectura abstracta de un sistema FIPA aparece el servicio de transporte de mensajes (i.e. `message-transport-service`) [16] como un elemento obligatorio de toda instanciación de la arquitectura abstracta y a través del cual han de enviarse todos los mensajes entre agentes. Esto hace que cualquier plataforma de agentes que sea FIPA integre un *router de mensajes* de forma natural y obligatoria. A continuación mencionamos los casos particulares de Zeus y JADE.

En Zeus existe una herramienta denominada el *Society Viewer* (SW) que presenta dos funciones principales para el programador a la hora de depurar el comportamiento de sus agentes: (a) la organización jerárquica de los agentes que están activos en el sistema en un momento determinado y (b) el intercambio de mensajes que se realiza entre ellos. Básicamente, consiste en una interface gráfica en donde aparecen representados todos los agentes activos, y las relaciones entre ellos codificadas con colores diferentes para cada tipo de relación. Los mensajes que se envíen se verán en forma de animación, o posteriormente en una bitácora que los mostrará adecuadamente.

JADE presenta un entorno de ejecución en el cual se puede disponer de un agente *sniffer* que absorbe todos los mensajes y los muestra. Con esto tenemos la capacidad de seguir las conversaciones y detectar anomalías en las mismas.

4.3 Otros elementos en depuración de código

Una actividad típica cuando se está depurando código de un lenguaje de alto nivel es la no solo acceder al contenido de las variables que estén activas en un momento determinado sino el poder cambiar sus valores para así poder comprobar el efecto de los cambios en la ejecución.

Esta es, precisamente, una propiedad que distingue a Zeus. Para ello disponemos de la *Control Tool*. Esto nos va a permitir visualizar, añadir, modificar y borrar, de cualquier agente que esté activo en ese momento, sus objetivos, recursos, especificación de tareas, *acquittances* y estrategias de coordinación. El resto de plataformas no presentan esa facilidad en tiempo de ejecución.

En [27] se sugiere la posibilidad de que los sistemas multi-agente basados en el intercambio de mensajes asíncronos, dependientes de la interacción con humanos favorezcan un esquema de desarrollo y depuración colaborativo. Esto es así ya que gracias al intercambio de mensajes tenemos dos elementos principales:

- Se puede programar en forma *colaborativa* al nivel de lenguaje común (i.e. performativas, semántica y lenguaje de contenido).
- Usando un mecanismo de enrutamiento de mensajes del tipo *store-and-forward* es posible obtener una traza de cualquier interacción, y por lo tanto realizar una depuración más efectiva.

Se propone un esquema de programación descentralizado por agentes en el cual cada uno de ellos se realiza en el lenguaje de programación deseado. La programación colaborativa comienza cuando se ponen los agentes a interactuar a través del ACL y el lenguaje de contenido. Esto funciona al estar todos los programadores en concordancia con respecto a la definición de los mensajes comunes. Los mensajes han de ser interpretados por los parsers en los correspondientes mensajes. Si un parser en un agente determinado no está actualizado, pueden darse las siguientes situaciones (a) se sigue considerando un mensaje como válido cuando se ha eliminado, (b) no se ha tomado en cuenta un nuevo tipo de mensaje añadido y (c) no se ha modificado la definición de un mensaje que ya existía. En cualquiera de los tres casos, cuando un agente recibe un mensaje que para él es erróneo, devolverá como respuesta un mensaje del tipo ERROR, incluyendo la cadena que no se pudo analizar. En esta forma, podemos obtener información, a través de las cadenas generadas en destino, de lo que está pasando internamente y obrar en consecuencia.

5 Conclusiones

Como hemos podido comprobar, la programación de agentes puede resultar en un número grande y heterogeneo de alternativas tecnológicas. Si dejamos a un lado la posibilidad de construir agentes sin utilizar herramientas diseñadas para tal efecto, hemos podido varios enfoques diferentes, dependiendo del modelo de agente subyacente y la tecnología usada.

Con respecto a los agentes reactivos, el modelo es tan simple que quizás no es necesario más que una herramienta que nos permita trabajar con sistemas de producción y permitir así el razonamiento, y un lenguaje de programación potente en cuanto a sus capacidades. Como ejemplo el binomio Jess + Java. Naturalmente, si es necesaria la interacción con otros agentes tendremos que usar alguna librería para manejo de ACLs.

En el campo de los agentes BDI se dispone de más variedad. Lógico, al ser el modelo BDI más rico y complejo que el modelo reactivo de agente. Por un lado podemos tener editores BDI que esconden casi todos los detalles de programación al desarrollador, dedicándose este únicamente a la definición de los agentes, sus relaciones y capacidades de negociación. Por otro lado tenemos lenguajes extendidos con directivas, que permiten, haciendo uso de un modelo BDI muy completo y potente la programación de sistemas de agentes de una forma efectiva aunque bastante compleja. Esto implica, además, una curva de aprendizaje bastante pronunciada. Y por último, una plataforma de agentes, asentada sobre un estándar muy extendido que posibilita la interoperabilidad entre plataformas distintas, y aplicaciones producidas por distintos equipos de desarrolladores aunque, de nuevo, bastante compleja.

Por otro lado, parece que la depuración y el mantenimiento de sistemas multi-agente es un tema que aun está bastante poco desarrollado, aunque la mayoría de herramientas ofrecen elementos que asisten en la tarea. Principalmente, la posibilidad de seguir las interacciones entre agentes.

Referencias

- [1] Agent Oriented Software Pty. Ltd. *JACK Intelligent Agents User Guide*, 2002.
- [2] Fabio Bellifemine, Giovanni Caire, and Giovanni Rimassa. *JADE Programmers Guide*. TILab and University of Parma, 2002. JADE 2.6.
- [3] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java. *AgentLink News Letter, January 1999*. White paper, <http://www.agent-software.com.au.>, 1999.
- [4] Paolo Busetta, Ralph Ronnquist, Andrew Hodgson, and Andrew Lucas. Using java for artificial intelligence and intelligent agent systems. Technical Report Technical Report 4, Agent Oriented Software, October 1999.
- [5] P.R. Cohen and H.J. Levesque. Teamwork. *Nous, Special Issue on Cognitive Science and Artificial Intelligence*, 4(25):487–512, 1991.
- [6] Jaron Collins, Divine Ndumu, and Christopher van Buskirk. *The Application Realisation Guide. Zeus Methodology Documentation Part III. The Zeus Agent Building Toolkit*, release 1.04 edition, July 2000.
- [7] Lyndon C. Lee Divine T. Ñdumu, Hyacinth S. Ñwana and Jaron C. Collis. Visualising and debugging distributed multi-agent systems. In *3rd Int. Conference on Autonomous Agents*, Seattle, May 1999.
- [8] I. A. Ferguson. Touringmachines: Autonomous agents with attitudes. *IEEE Computer*, 25(5):51–55, 1992.
- [9] Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Technical report, FIPA, August 2001.
- [10] Foundation for Intelligent Physical Agents. FIPA agent management specification. Technical report, FIPA, October 2001.
- [11] Foundation for Intelligent Physical Agents. FIPA contract net interaction protocol specification. Technical report, FIPA, August 2001.
- [12] Foundation for Intelligent Physical Agents. FIPA dutch auction interaction protocol specification. Technical report, FIPA, August 2001.
- [13] Foundation for Intelligent Physical Agents. FIPA english auction interaction protocol specification. Technical report, FIPA, August 2001.
- [14] Foundation for Intelligent Physical Agents. FIPA query interaction protocol specification. Technical report, FIPA, August 2001.
- [15] Foundation for Intelligent Physical Agents. FIPA request interaction protocol specification. Technical report, FIPA, August 2001.
- [16] Foundation for Intelligent Physical Agents. FIPA abstract architecture specification. Technical report, FIPA, February 2002.
- [17] Ernest J. Friedman-Hill. *Jess, The Expert System Shell for the Java Platform*. Distributed Computing Systems. Sandia National Laboratories, version 6.1a3 edition, May 2002.

- [18] L. Gasser, C. Braganza, and N. Hermann. Mace: A flexible testbed for distributed ai research. In Pitman M. Huhns, Ed., editor, *Distributed Artificial Intelligence*, pages 119–152. London and Morgan Kaufmann, San Mateo, CA, 1987.
- [19] M.P. Georgeff and A.L. Lansky. Reactive reasoning and planning. In *Readings in Plannings*, page 729734. 1990.
- [20] Michael N. Huhns and Munindar P. Singh. Agents are everywhere! *IEEE INTERNET COMPUTING*, 1(1), Jan/Feb 1987.
- [21] Carlos Iglesias, Mercedes Garrijo, and José Gonzalez. A survey of agent-oriented methodologies. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.
- [22] N. R. Jennings and M. Wooldridge. Agent-oriented software engineering. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2001.
- [23] H. Jeon, C. Petrie, and M. R. Cutkosky. Jatlite: A java agent infrastructure with message routing. *IEEE Internet Computing*, Mar/Apr 2000.
- [24] Agent Oriented Software Pty. Ltd. *SimpleTeam Technical Brief*, August 2002.
- [25] J. Muller and M. Pischel. The Agent Architecture InterRaP: Concept and application. Technical Report RR-93-26, 1993.
- [26] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.
- [27] Charles J. Petrie. Agent-Based Software Engineering. In Jeffrey Bradshaw and Geoff Arnold, editors, *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000)*, Manchester, UK, 2000. The Practical Application Company Ltd.
- [28] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.
- [29] Reticular Systems Inc. *AgentBuilder User’s Guide. An Integrated Toolkit for Constructing Intelligent Software Agents.*, version 1.3. rev. 0. edition, April 2000.
- [30] Pierre-Michel Ricordel and Yves Demazeau. From analysis to deployment: a multi-agent platform survey. In Andrea Ominici, Robert Tolksdorf, and Franco Zambonelli, editors, *Working notes of the Engineering Engineering Societies in the Agent’s World*. 2000.
- [31] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
- [32] Reid R. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 357–366. Morgan Kaufmann Publishers, Los Altos, CA, 1988.
- [33] Marc H. Van Liederke and Nicholas M. Avouris. Debugging multi-agent systems. *Information and Software Technology*, 37(2):103–112, 1995.
- [34] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 2(10):115–152, 1995.