# Agentes BDI bajo interacciones reguladas: un enfoque basado en flujos de trabajo


Bexy Alfonso Espinosa

Departamento de Sistemas Informáticos y Computación

Universitat Politécnica de València

Dirigido por:
Vicente Juan Botti Navarro
Emilio Vivancos Rubio

*Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital*

Junio 2012

# Agradecimientos

Agradezco a mis directores Vicente Botti y Emilio Vivancos por orientarme y apoyarme para la realización de este trabajo, y también para mi preparación personal como investigadora. Agradezco también a mis compañeros de trabajo por su ayuda y apoyo incondicional.

# Índice general

# Índice de figuras

## ÍNDICE DE FIGURAS

# Capítulo 1

# Introducción

El área de los Sistemas Multi-agente es uno de los campos de investigación más activos y prometedores hoy día. Proporcionan adaptabilidad, escalabilidad, alta tolerancia a fallos, autonomía, versatilidad entre otras, siendo estas características muy apropiadas para un amplio espectro de aplicaciones. Los temas de investigación relacionados se centran en diversos aspectos, como es el caso de la definición de estándares, metodologías, lenguajes de programación, plataformas de comunicación, semánticas etc. En particular el estudio de plataformas de comunicación y de aspectos relacionados con las interacciones entre agentes cobra vital importancia al considerar las necesidades y características de este tipo de sistemas. Un agente en un entorno multi-agente necesita comunicarse con otros para lograr sus objetivos. Aunque estas interacciones pudieran componerse de una secuencia indeterminada y desordenada de mensajes y acciones, el regularles mediante el uso de patrones, precedencias y restricciones, facilita la programación de los agentes.

Las propuestas de lenguajes de comunicación más utilizados son KQML (Knowledge Query and Manipulation Language) de DARPA [16] y ACL (Agent Communication Language) de FIPA [20]. Ambos establecen la estructura de los mensajes a intercambiar, se basan en la teoría de actos de habla [7] y parten de la existencia de una ontología común entre los agentes. Este trabajo se ha basado en el segundo por el alto nivel de aceptación que presenta en la comunidad de programadores de agentes. Al emplear este lenguaje, los agentes intercambian mensajes cuya estructura está conformada por un conjunto de parámetros como es el caso de: una performativa (en la que se indica el tipo de acto de comunicación que tendrá lugar con el mensaje), el receptor y remitente, el contenido a enviar, la ontología que se emplea etc.

Otro concepto con el que se trata en este trabajo es el de "conversaciones de agentes". Éste es un término empleado para las posibles combinaciones y secuencia de mensajes a intercambiar entre dos o más agentes que participan en un sistema de agentes dado [24]. Dichas conversaciones pueden responder a uno o varios protocolos, y en este sentido la **Foundation for Intelligent Physical Agents** (FIPA), ha definido un conjunto de protocolos estándar que contienen patrones predefinidos de secuencias de mensajes válidos: los "Protocolos de interacción FIPA" [19]. Este grupo de protocolos comprende desde la forma más básica de interacción hasta otras más complejas.

Paralelamente, el desarrollo de lenguajes de alto nivel para la programación de agentes, también es un área de investigación creciente. Generalmente se basan en el modelo BDI de agentes (Beliefs, Desires, Intentions) que satisface los requerimientos de autonomía, proactividad, reactividad etc, de este tipo de sistemas. El presente trabajo permite el uso de Jason [9] como lenguaje para programar los agentes. Sus creadores son R. Bordini y J. Hubner, y es una extensión del lenguaje AgentSpeak(L) [28]. Está codificado en Java, siendo posible, mediante su extensión, escribir algunas partes del código también en Java a través de estructuras denominadas "acciones internas" (acciones diseñadas para ser ejecutadas "fuera de la mente del agente"). Estas acciones internas han sido empleadas para extender el lenguaje de manera que permita el uso de los protocolos de interacción estándar de FIPA de una manera estructurada y sencilla.

Algunas contribuciones [4, 8, 11], hacen uso de estos conceptos con el objetivo de facilitar el desarrollo de agentes y de modelar su interacción proporcionando la infraestructura necesaria para que estos puedan interactuar bajo ciertas condiciones. No obstante, suelen cubrir las distintas necesidades en esta área de manera parcial, siendo necesario a menudo la búsqueda de varias tecnologías para que, su conjunto, permitan obtener una solución. Partiendo de ello, el objetivo fundamental de este trabajo, es **ofrecer un mecanismo fácil e intuitivo, mediante el cual se puedan especificar los agentes de un sistema en términos de sus interacciones y las relaciones entre ellas aunando distintas técnicas y lenguajes**.

El trabajo está organizado en 6 capítulos donde el primero es la introducción al trabajo. El resto del documento está organizado como sigue. En el Capítulo 2 se realiza un análisis de algunas plataformas propuestas con soporte para las interacciones entre agentes. El Capítulo 3 abunda en el diseño realizado y directrices seguidas para permitir en agentes BDI programados con Jason y ejecutándose sobre la plataforma Magentix2, el empleo de los protocolos de

interacción de FIPA en sus conversaciones. En el Capítulo 4 se realiza una propuesta de una herramienta de modelado que permite representar las interacciones, condiciones y los elementos presentes en un sistema regulado, dándole una secuenciación a las acciones que tienen lugar; además se describen las consideraciones tenidas en cuenta en la generación de código fuente a partir del modelo. En el Capítulo 5 se describe cómo se ha aplicado la propuesta a un caso de estudio en el contexto de un mercado de agua. Finalmente, en el Capítulo 6, se ofrecen las conclusiones en las que se presenta la valoración del trabajo propuesto y el trabajo futuro a realizar.

# 1. INTRODUCCIÓN

# Capítulo 2

# Estado del arte y motivación

## 2.1. Introducción

En este capítulo se comenta el estudio realizado relativo a algunas herramientas y técnicas existentes que han sido desarrolladas para mejorar la especificación de las interacciones entre los agentes. Se realiza una comparación de las mismas, valorando cómo se podrían mejorar. Además se menciona cómo se integran, en estas herramientas de especificación de interacciones entre agentes, los lenguajes de programación de agentes BDI y finalmente se presenta la motivación del trabajo partiendo de las descripciones realizadas. Finalmente se ofrecen las conclusiones de este estudio.

## 2.2. Protocolos de interacción

El proceso de comprender un mensaje referido a una ontología particular puede requerir un razonamiento considerable por parte de los agentes, y por consiguiente puede afectar a su rendimiento. Sería necesario probablemente explorar un espacio de búsqueda de posibles respuestas para encontrar la apropiada. El empleo de protocolos de interacción contribuye a reducir este espacio de búsqueda, ya que a través es estos se puede acortar el número de posibles respuestas a un tipo de mensaje para un protocolo dado. Según las especificaciones de FIPA [19] un protocolo de interacción es un "patrón común de diálogo empleado para ejecutar alguna tarea útil; se emplea para facilitar la simplificación de la maquinaria computacional necesaria para apoyar una tarea de diálogo dada entre agentes; simplemente un patrón de diálogo". Las conversaciones en cambio las define como "una secuencia continua de actos comunicativos in-

tercambiados entre uno o más agentes relativos a algún tema actual del discurso; puede (quizás implícitamente) acumular contexto que es empleado para determinar el significado de mensajes posteriores en la conversación". Por tanto una conversación puede estar vinculada a uno o más protocolos de interacción.

FIPA ha identificado un conjunto de patrones que se suelen seguir en las conversaciones de agentes y los ha estandarizado a través de los protocolos de interacción [19], que pueden ser a su vez representados por máquinas de estado. En este conjunto se incluyen los protocolos FIPA: "request", "query", "request when", "contract net", "iterated contract net", "English auction", "Dutch auction", "brokering", "recruiting", "subscribe" y "propose". La definición de estos estándares ha permitido que las plataformas de sistemas multi-agente se extiendan para incluir estos protocolos como soporte a la comunicación [4, 8, 11, 27].

### 2.2.1. Plataformas que los soportan

### 2.2.2. Jade

Jade (Java Agent Development Framework) [1, 8], desarrollada a partir de 1998 por Telecom Italia, es probablemente la plataforma más popular y extendida en el área de los sistemas multi-agente. Ha sido desarrollada en el lenguaje de programación Java y cumple con los estándares de FIPA, soportando de esta manera su arquitectura abstracta, ACL como lenguaje de comunicación y también sus protocolos de interacción. Los agentes en Jade se estructuran como hilos de ejecución individuales con su propio comportamiento, que viven en los conocidos "contenedores" y que pueden o no residir en el mismo nodo. Una plataforma consiste en un contenedor principal y contenedores adicionales que sostienen los agentes. El contenedor principal tiene directorios de todos los contenedores en ejecución y de sus respectivos agentes.

En general la plataforma ofrece soporte para un conjunto de facilidades además de las necesarias en cualquier sistema multi-agente como por ejemplo para tolerancia a fallos, para agentes móviles, interacción con servicios web, etc. Además, como se ha mencionado anteriormente, es posible también en los agentes Jade, el empleo de los protocolos de interacción de FIPA. Esto se logra gracias a un conjunto de clases para manipular las conversaciones basadas en estos protocolos. Estas clases se encuentran en el paquete *jade.proto*, y proporcionan métodos "callback" para ser empleados por los programadores; en estos se debe especificar la lógica necesaria para resolver un problema específico en los pasos del protocolo. Estas clases se dividen en dos grupos dependiendo del rol que desempeñe el agente que emplee la clase:

*iniciador* o *de respuesta*. De esta forma las interacciones deben ser especificadas en lenguaje Java en su totalidad, debiéndose preparar los mensajes a enviar en las llamadas a los métodos "callback".

### 2.2.3. Jadex

Jadex [11, 26] es una plataforma multi-agente que sigue el modelo BDI de agentes. Permite implementar agentes inteligentes en XML y Java que pueden desplegarse en distintos tipos de middleware como por ejemplo Jade o cualquier otra plataforma gracias a los "adapters". Un agente Jadex se compone de un fichero de definición de agente escrito en XML y de las clases Java que lo implementan; éste puede seguir los estándares de FIPA siempre y cuando la plataforma en la que vive lo permita (como es el caso de Jade). Jadex también proporciona la "Capacidad de protocolos de interacción" con soporte incorporado para la mayoría de los protocolos de interacción FIPA. En Jadex los distintos pasos de los protocolos se han analizado y llevado a la especificación ampliada de protocolos orientados a objetivos [1]. Esta descripción de protocolos divide cada rol en dos capas: la del dominio y la del protocolo, que en su conjunto encapsulan toda la funcionalidad de un rol.

Esta capacidad de protocolos contiene la funcionalidad de la interacción tanto del lado del iniciador como del participante. Para utilizar esta funcionalidad es necesario incluir la capacidad del protocolo dentro del fichero de definición del agente o ADF (Agent Definition File) por sus siglas en inglés. Del lado del iniciador la ejecución de un protocolo comienza cuando se crea una instancia del tipo de objetivo apropiado (por ejemplo en el protocolo request sería "rp_initiate"). Dentro de los planes se puede iniciar un protocolo creando una instancia del tipo de objetivo iniciado, especificando los valores para los parámetros necesarios y enviándolo. Cuando el protocolo finaliza, los resultados pueden ser extraídos directamente de los parámetros de salida del objetivo.

En el caso del participante la capacidad de protocolos depende también de emplear los objetivos como receptor y además de la configuración de las creencias. Para ello es necesario determinar exactamente qué tipos de mensaje se aceptarán del iniciador. Por ejemplo para el protocolo FIPA Request existen tres objetivos para los participantes: "rp_receiver_interaction", "rp_decide_request" and "rp_execute_request". En el caso de la configuración de las creencias se ha definido además, para cada protocolo soportado, una creencia filtro; por ejemplo la

[1] Jadex User Guide. http://jadex.informatik.uni-hamburg.de/docs/jadex-0.96x/userguide

creencia “rp_filter se emplea para determinar qué mensaje request debe activar la capacidad de protocolo.

### 2.2.4. Opal

Opal [27] es una plataforma de agentes multi-nivel que brinda soporte para organizaciones de agentes por medio de un módulo de gestión de conversaciones. Se basa en la idea de concebir los agentes al mínimo nivel de operación pero aún con las características necesarias para que se siga considerando agente para propósitos de modelado y diseño. De esta forma en Opal se define un agente como una entidad persistente que se implementa en un sistema multi-agente o también un actor que juega uno o más roles en una sociedad de agentes; y por otro lado también se define el concepto de “Micro-agente” como un tipo particular de agente que representa el mínimo y más primitivo nivel de de instanciación de un agente. Estos micro-agentes emplean una forma de comunicación más simple que los agentes de más alto nivel como por ejemplo los definidos por FIPA y poseen una flexibilidad más limitada. Se comportan de manera predecible y estrictamente definida y no ejecutan razonamiento en tiempo de ejecución. Los agentes por otra parte pueden estar compuestos por cualquier número de otros agentes o micro-agentes y los micro-agentes no primitivos se componen solamente de micro-agentes. Se plantea que la ventaja clave de esta arquitectura es que para los sistemas de menor escala los micro-agentes superan radicalmente en rendimiento a los agentes de grano más grueso en tiempo de ejecución.

Por otra parte en Opal existen los roles de micro-agente representando agentes FIPA que puede contener una variedad de roles de sub-agente. Un agente FIPA en Opal emplea un rol micro-agente como *controlador de las conversaciones* en las que el agente está involucrado. Este controlador requiere a su vez que algún micro-agente exista para jugar el rol de *distribuidor de mensajes*. Otro agente FIPA también puede requerir el micro-agente con el rol *Belief-Desire-Intention* para permitirle que se pueda desarrollar usando el modelo BDI.

Relativo a los protocolos de interacción, en Opal se implementan numerosos protocolos de interacción FIPA. Para controlar la expedición y cambio de los estados de las conversaciones dadas las propiedades de los mensajes recibidos y enviados, existe una entidad denominada *Controlador de interacción*. Esta entidad es responsable de iniciar, monitorizar y controlar las instanciaciones de los roles existentes para cada uno de los protocolos FIPA (por ejemplo: FIPARequestTracker o FIPAQueryTracker). Estos roles son especializaciones a su vez de un rol abstracto básico que es “Interaction traker”. Otro componente especial de la infraestructura de Opal es el *Gestor de conversaciones*. Este gestor está a cargo de aspectos como el registro

de protocolos de interacción individuales y sus violaciones, registrando su contexto; se hace cargo además de los tiempos de espera y respuestas tardías y de la localización de recursos para tareas más importantes etc. En Opal las conversaciones se descomponen en tres capas: *protocolo* (plantilla de secuencia esperada de actos comunicativos organizados en roles), *conversación* (instancia particular de uno o varios protocolos) y *política* (reglas y especificaciones de interacción que guían un camino o trayectoria en el espacio de una conversación). De esta manera cada protocolo define un espacio de posibles secuencias de actos comunicativos y cada conversación sigue una trayectoria en este espacio donde la política guía una conversación particular.

### 2.2.5. Magentix2

Magentix2 [2, 4] es una plataforma concebida para sistemas multi-agente abiertos. Además de brindar soporte para cuestiones básicas necesarias en un sistema multi-agente como la ejecución del ciclo de vida de un agente y la comunicación, también ofrece servicios y herramientas que permiten una gestión optimizada y segura del sistema. Por ejemplo brinda soporte para interacciones flexibles a través de protocolos y conversaciones entre agentes y organizaciones de agentes. También ofrece un servicio de trazas que permite a los agentes publicar o suscribirse a determinados eventos; un API de argumentación que permite a los agentes establecer diálogos de argumentación para llegara a acuerdos; y un modelo de seguridad que incorpora mecanismos de seguridad a bajo nivel y medidas de confianza. En Magentix2 los usuarios pueden gestionar aspectos relativos a organizaciones de manera sencilla a través de la plataforma THOMAS. Por último esta plataforma ofrece una integración con el lenguaje de programación de agentes Jason [9], lenguaje de programación lógica de alto nivel basado en el modelo BDI de agentes.

Como se ha mencionado Magentix2 ofrece la posibilidad del empleo de protocolos de interacción FIPA en las interacciones de los agentes. Esto se logra gracias a las Fábrica de Conversaciones [18], que es la herramienta encargada de la gestión de las conversaciones de agentes en la plataforma. Una fábrica de conversaciones permite mantener una interacción completa entre dos o más agentes de donde se tiene un *iniciador* (el que inicia la conversación) y uno o más *participantes* (el resto de los agentes). La plataforma posee plantillas de protocolos FIPA tanto para el rol de *iniciador* como el de *participante*, que pueden ser personalizadas en los pasos del protocolo donde sea necesario a través de métodos “callback”. Las dos estructuras principales que soportan las conversaciones como parte de la fábrica de conversaciones son *CProcessor* y

*CFactory*. La primera es la encargada de gestionar los mensajes enviados y recibidos en cada paso de la conversación realizando las acciones pertinentes para determinar el próximo paso en el protocolo; la segunda crea las conversaciones y los *CProcessor* que corresponden a un protocolo específico.

## 2.3. Análisis comparativo de plataformas

Tanto Jadex como Jade ofrecen clases Java para implementar los protocolos de interacción de FIPA, de manera que no se ofrece la posibilidad de emplear un lenguaje de programación de alto nivel específico para agentes directamente integrado con las herramientas para la gestión de las conversaciones. Esto no limita que estas plataformas sean seleccionadas para implementar agentes que requieran interactuar siguiendo protocolos de interacción, sin embargo, es más conveniente contar con un lenguaje de programación de agentes de alto nivel basado en la lógica cuando se requieren estructuras de razonamiento complejas que responden a la naturaleza inherente a los agentes de autonomía y proactividad.

Por otra parte tanto en Jadex como en Jade es necesario cumplir con la estructura rígida del protocolo en todo momento, lo que implica una falta de flexibilidad a la hora de entablar conversaciones. La plataforma Opal sin embargo, parece cubrir estos aspectos ya que con el empleo de micro-agentes se pueden implementar agentes de "grano grueso" que pueden seguir el modelo BDI, entablar conversaciones que responden a los estándares de protocolos de interacción de FIPA y gracias a la capa de "política" de su arquitectura las conversaciones pueden ser más flexibles y abiertas. No obstante el proyecto parece no haber alcanzado la madurez necesaria para ofrecer estas facilidades de manera pública en una herramienta que pueda estar disponible para su utilización y prueba.

Por último Magentix2 ofrece una serie de posibilidades apropiadas para los objetivos de nuestro trabajo, permitiendo el empleo de protocolos de interacción que siguen los estándares de FIPA ofreciendo, al mismo tiempo, mecanismos para que puedan ser modificados dinámicamente sin necesidad de reiniciar el sistema [18]. Otra de sus ventajas, como se explicará en el Capítulo 3, es que durante las conversaciones se almacena y añade la información requerida a los mensajes que serán enviados (Ej: iniciador, participantes, el estado de la conversación, etc.) de manera que no es necesario que el programador incluya esta información de forma explícita. Magentix2 permite además la programación de agentes en el lenguaje de alto nivel Jason [9], muy aceptado en la comunidad de programación de agentes por su flexibilidad y robustez, y

el empleo de mecanismos para conseguir una gestión optimizada y segura de sistemas multi-agente abiertos. Dadas estas condiciones, se ha seleccionado Magentix2 como la plataforma más apropiada para elaborar nuestra propuesta.

## 2.4. Motivación y objetivos

Dada la complejidad de todos los conceptos que pueden estar asociados a la interacción de agentes a través de conversaciones y protocolos de interacción, el objetivo principal de esta propuesta es ofrecer un mecanismo que permita al desarrollador realizar una especificación más simple y clara de los agentes. De esta forma se le alivia de los aspectos relacionados con subprocesos y sincronización subyacentes, así como de la preparación de información que no es del dominio o la gestión de tiempos de espera, entre otros. Estos aspectos serían gestionados de manera natural por la infraestructura del agente en lugar de hacerlo el programador. Así mismo se pretende, por consiguiente, mejorar el mecanismo de paso de mensajes poniendo atención al desempeño y la escalabilidad del sistema.

Para conseguir este propósito se proponen los siguientes objetivos específicos:

- Permitir en un lenguaje de alto nivel de programación de agentes, como es el caso de Jason, el empleo del los protocolos de interacción estándar de FIPA a través de la automatización, en la mayor medida posible de este proceso.
- Estudiar y adaptar la plataforma Magentix2, con soporte para protocolos de interacción, para ofrecer la funcionalidad anterior.
- Ofrecer un mecanismo de representación que permita establecer la interacciones que tendrán lugar en un sistema regulado y la relación entre ellas.
- Permitir la generación de código fuente a partir de dicha representación.
- Aplicar las estructuras diseñadas e implementadas a un caso de estudio.

## 2.5. Conclusiones

Existe una variedad de propuestas que intentan abordar el tema de las interacciones de agentes desde distintas perspectivas. En el presente capítulo se ha hecho una selección de las más representativas y se ha estudiado un conjunto de plataformas que brindan soporte para

protocolos de interacción, analizándolas tanto de manera global como desde el punto de vista particular de las interacciones. Se han analizado las facilidades que ofrecen y cuan adecuadas son para el propósito del presente trabajo y se ha justificado la selección de Magentix2 como plataforma más adecuada para desarrollar la propuesta.

# Capítulo 3

# Soporte para las interacciones de agentes BDI

## 3.1. Introducción

La plataforma Magentix2, seleccionada para implementar la propuesta, ofrece mecanismos para que los agentes puedan entablar conversaciones múltiples y simultáneas. Por otra parte el lenguaje Jason basado en el modelo BDI de agentes permite emplear estructuras de alto nivel a través de la programación lógica que facilitan la implementación de agentes con altas capacidades de razonamiento. En el presente capítulo se introduce una propuesta para integrar ambas tecnologías ofreciendo la posibilidad de crear agentes BDI conversacionales a nivel de la plataforma donde viven. Se detallan los aspectos fundamentales tanto de las tecnologías de soporte como de la propuesta y se ilustra la manera de hacer uso de ella a través de un ejemplo donde dos agentes mantienen una conversación haciendo uso del protocolo Request de FIPA.

## 3.2. Herramientas de soporte

### 3.2.1. Plataforma de Sistema Multi-Agente: Magentix2

Existen numerosas plataformas y herramientas que permiten especificar agentes comunicativos. Como ha sido mencionado anteriormente se ha seleccionado Magentix2 [4] para la presente propuesta por ser una plataforma que sigue los estándares FIPA con capacidad para ofrecer un conjunto de mecanismos útiles para la comunicación entre agentes así como herramientas para permitir la programación de agentes en un lenguaje de alto nivel basado en el

modelo BDI, entre otras facilidades.

Magentix2, emplea el middleware orientado a mensajes AMQP (Advanced Message Queuing Protocol) [1] para la comunicación. Este estándar facilita la interoperabilidad entre entidades heterogéneas, lo que convierte a la plataforma en un sistema abierto donde agentes heterogéneos pueden interactuar a través de mensajes FIPA-ACL. En específico Magentix2 emplea la implementación Apache Qpid [2] de AMQP, de manera que los agentes emplean las APIs del cliente de Qpid para conectarse a un broker Qpid y comunicarse con otros agentes en cualquier punto de internet. Con Magentix2 se ofrece una plataforma de soporte para la gestión de interacciones complejas entre agentes que pueden cambiar dinámicamente, lo que permite establecer conversaciones flexibles y abiertas.

Para facilitar el empleo de conversaciones, como ha sido mencionado en el capítulo anterior, la plataforma posee dos estructuras fundamentales encargadas de gestionarlas: el procesador de conversaciones *CProcessor* y la fábrica de conversaciones *CFactory*. Cada ejecución de un *CProcessor* se realiza en un hilo independiente. Tanto en el caso del rol *iniciador* como del *participante*, éste decide en cualquier punto de la conversación cuál es el próximo paso de acuerdo al protocolo de interacción que se está siguiendo. Además realiza las acciones y gestiona los mensajes enviados y recibidos en cada paso de la conversación. Los estados que representan las acciones que pueden ser realizadas pueden ser de los tipos [18]:

- *Begin*: Para iniciar una conversación. Siempre está presente.
- *Final*: Para finalizar una conversación. Siempre está presente.
- *Action*: Para representar cualquier acción no relacionada con algún acto de habla.
- *Send*: Para enviar un mensaje.
- *Wait*: En este tipo de estado la conversación se detiene hasta la recepción de un nuevo mensaje (lo que implica que el rol pase a un estado *Receive* siguiente) o hasta que haya transcurrido un tiempo de espera especificado previamente.
- *Receive*: Para recibir un mensaje. Debe estar precedido de un estado *Wait*.
- *Initiate*: Para iniciar una nueva subconversación.

[1] AMQP: http://www.amqp.org/
[2] Apache Qpid: http://qpid.apache.org/

- *Participate*: Es un tipo especial del estado *Receive*. El rol inicia una nueva subconversación al recibir el mensaje apropiado.

Además de estos estados existen otros tipos de estados para excepciones a los que se conecta todo estado normal de manera implícita. Estos son: *Cancel* (cuando se recibe un mensaje para finalizar la conversación de forma inesperada), *Not Accepted Message* (cuando desde el estado *Wait* se recibe un mensaje y no existe estado *Receive* apropiado para su procesamiento) y *Sending Errors* (trata la excepción si se produce un error al enviar un mensaje en el estado *Send*). En cada uno de estos estados el programador puede implementar un método con las acciones que se ejecutarán por el *CProcessor* cuando la conversación alcance ese estado. Estos métodos tienen parámetros y devuelven distintos valores en dependencia del tipo de estado. Las conversaciones asociadas al *CProcessor* pueden sufrir modificaciones en tiempo de ejecución que pueden afectar a los estados o las transiciones entre estos.

Por otra parte las *CFactories* tienen la finalidad de comenzar las conversaciones. Cuando comienza una conversación la *CFactory* inicia un *CProcessor* encargado de ejecutar la conversación. Existe una *CFactory* para el rol *iniciador* y una para el rol *participante*. En el primer caso la conversación comienza sin necesidad de eventos o estímulos externos; en el segundo caso en cambio la conversación se inicia cuando el agente recibe el mensaje apropiado.

Cada conversación posee un identificador único que es asignado por el *CProcessor* si el agente es *iniciador*. En el caso del *participante* el identificador estará contenido en el primer mensaje recibido de la conversación. Este mensaje puede implicar iniciar una conversación si no existe entre los identificadores de las conversaciones a las que el *participante* se ha unido. En caso contrario se procesa el mensaje de acuerdo a la conversación a la que pertenece. Este proceso es realizado por el *CProcessor* correspondiente.

Para el caso en el que ninguna *CFactory* pueda tratar el mensaje recibido, éste es asignado a una *CFactory* por defecto (*DefaultFactory*) la cual está a cargo de tratar los mensajes que no pueden ser tratados por ninguna otra fábrica.

Las conversaciones, además, se pueden anidar, existiendo para ello dos tipos de subprotocolo: síncrono (la conversación se detiene hasta que la hija finalice) y asíncrono (las conversaciones madre e hija siguen su ejecución de manera individual). En el primer caso a la nueva conversación le es asignado un identificador igual al de la que la creó; en el segundo caso se asigna un nuevo identificador [17].

### 3.2.2. Lenguaje de programación de agentes BDI: Jason

A través de Magentix2 es posible diseñar agentes con una alta capacidad de razonamiento que siguen el modelo BDI de agentes a través del uso de Jason[9] como lenguaje de programación de alto nivel. Jason es un intérprete basado en Java, para una versión extendida de AgentSpeak(L) [28], lenguaje orientado a agentes que contempla los aspectos más importantes de los sistemas de planificación reactiva. AgentSpeak(L) es uno de los mejores lenguajes basados en la arquitectura BDI. Es un lenguaje abstracto basado en la lógica que permite implementar agentes como sistemas de planificación reactiva. De manera que se ejecutan continuamente reaccionando a eventos y ejecutando planes para esos eventos. La naturaleza proactiva se logra a través de la noción de los objetivos, que no son más que estados deseados del mundo. A continuación se explicarán estos elementos en más detalle.

AgentSpeak(L) permite definir los agentes en términos de tres elementos principales: creencias (*Beliefs*), objetivos (*Goals*) y planes (*Plans*).

Las **creencias** representan la visión actual del agente del estado del mundo en el que está situado. Estas cambian de manera continúa durante la ejecución del agente y esto puede estar provocado por varios factores. En primer lugar, las creencias pueden cambiar después del que el agente haya "percibido" en el entorno (el mundo observable para él) algún hecho nuevo o distinto a los que ya tenía registrados; cuando otro agente le ha enviado alguna información a través de un mensaje; o cuando él explícitamente modifica su base de creencias como consecuencia de un razonamiento previo a través de "notas mentales". Las creencias se representan, en su forma más simple, a través de un conjunto de literales que expresan una propiedad de un objeto o individuo. Por ejemplo: `tall(john)` que expresa la propiedad *tall* del individuo *john*. Otro elemento importante que puede estar incluido en la base de creencias son las reglas. Estas permiten concluir nueva información basadas en otra ya conocida como cierta. Poseen una parte izquierda y una parte derecha o cuerpo separadas por el operador ":-". En la parte izquierda solo puede haber un literal que es la conclusión a ser obtenida si la parte derecha se satisface. En la parte derecha en cambio puede aparecer el mismo tipo de estructuras que aparecen en el contexto de un plan. Por ejemplo dada la regla:

```
fullname(Name,SurN,FullN) :- .concat(Name,`` '',SurN,FullN).
```

Esta permite, a través de la acción interna `.concat()` (se describen más adelante), y dados un nombre ( `Name` ) y un apellido ( `SurN` ), obtener la concatenación de ambos con un espacio

en blanco en medio. El resultado es almacenado en la variable `FullN` [1].

Los **objetivos** en cambio expresan la situación del mundo a la que el agente desea llegar. Existen dos tipos de objetivos que son:

- *Achievement Goals*: Expresan una situación a alcanzar y que el agente considerará verdadera si lo consigue. Se denotan con el operador `!` . Por ejemplo si queremos expresar el objetivo en el que un *switch* esté en estado encendido (*on*) se podría expresar de la forma `!state(switch,on)` .

- *Test goals*: Construcciones para extraer información de la base de creencias, o sea, para verificar si el agente cree un literal o conjunción de literales. Se denotan con el operador `?` . De manera que, por ejemplo, si se quisiera saber lo que el agente cree con respecto al estado del *switch* del ejemplo anterior, se pudiera expresar como: `?state(switch,S)` , donde ‘ `S` ’ se instanciaría con el estado *on* ó *off*.

Finalmente los **planes**, como se puede pensar intuitivamente, permiten alcanzar un objetivo a través de una secuencia de acciones. Esto ocurre a partir de *eventos* que se generan como consecuencia de la adición o eliminación de estos objetivos. Al generarse el evento, una secuencia de acciones que componen el plan es ejecutada y, si no ocurren fallos, el objetivo se alcanzaría. Básicamente los planes se componen de tres elementos: evento disparador, contexto y cuerpo y se denota de la forma:

```
evento_disparador : contexto <- cuerpo.
```

El *evento disparador* se puede producir por causas asociadas a cambios en las creencias o cambios en los objetivos. Estas producen cambios en el estado mental del agente y son las siguientes: adición ( `+!g` ) o eliminación ( `-!g` ) de un *achievement goal* ‘ `!g` ’, adición ( `+b` ) o eliminación ( `-b` ) de una creencia ‘ `b` ’ o adición ( `+?g` ) o eliminación ( `-?g` ) de un *test goal* ‘ `?g` ’. Una vez se ha generado un evento, el plan correspondiente pasa a ser “relevante”.

El *contexto* es el conjunto de condiciones que deben ser ciertas para que, dado un evento, el plan pueda pasar a ser ejecutado. Permite chequear la situación actual y determinar, entre las posibles alternativas, si es posible que un plan particular tenga éxito dada la última información que el agente posea. Este conjunto de condiciones debe existir o ser una consecuencia lógica de

[1] En Jason los términos en mayúsculas son variables.

la base de creencias del agente. El contexto no es una parte imprescindible del plan, de manera que, si no se ha definido, o si habiéndose definido se cumple, el plan pasa a ser "aplicable".

Como se ha mencionado el plan se compone además de un conjunto de acciones que se ejecutarán una vez que el plan sea "aplicable" y sea seleccionado para su ejecución. Estas acciones conforman el *cuerpo* del plan y pueden ser de distintos tipos:

- *Actions*: representan aquellas acciones que el agente puede realizar en su entorno. Simbólicamente se representan con un predicado con todos los términos instanciados.

- *Achievement goals*: son objetivos que se añaden para que, si existe un plan para alcanzar dicho objetivo, éste sea ejecutado. Esta acción puede ser exitosa (su plan correspondiente se ejecuta con éxito) o fallar (el plan correspondiente no finaliza su ejecución o no existe dicho plan). En caso de no tener éxito esto podría provocar el fallo del plan al que pertenece [1].

- *Test goals*: objetivos cuyo propósito fundamental es extraer información de la base de creencias; ya sea para comprobar si una creencia es cierta o para recuperar el valor de ciertos parámetros. Pueden implicar de igual forma la ejecución de un plan si existe alguno definido para ese objetivo. Provoca fallo si no existe plan o si el predicado no es una consecuencia lógica de la base de creencias. Aunque puedan parecer redundantes (este chequeo se puede realizar también en el contexto del plan), su utilidad radica en que permite conocer el estado de las creencias en un momento preciso del plan y no solo al inicio [2].

- *Mental Notes*: a través de este tipo de acciones es posible añadir información a la base de creencias que puede servir para "anotar" determinado resultado o dato para ser empleado en procesamientos posteriores.

- *Expressions*: de manera similar a Prolog, estas acciones incluyen expresiones relacionales o aritméticas.

[1] Existe una manera de añadir un objetivo sin tener que esperar por su consecución. Referirse a [9] para más información.

[2] La base de creencias cambia constantemente y la situación al comenzar la ejecución de un plan puede diferir de la que se presente durante dicha ejecución.

- *Internal actions*: estas acciones permiten al agente realizar algún procesamiento necesario, dentro del código del agente, accediendo al código base, que es en este caso Java. Todo el procesamiento correspondiente a la acción se realiza en un paso del ciclo de razonamiento del agente y puede ser empleada por los programadores para extender el lenguaje de programación con operaciones que no pueden ser realizadas de otra forma. Se distinguen de las acciones en el entorno porque tienen el carácter ‘ . ’ al inicio. Existen también acciones internas (AI) estándar predefinidas en Jason, útiles para realizar acciones genéricas como por ejemplo el envío de mensajes [1] ( `.send(mum,askIf,raining)` ) o imprimir un texto en pantalla ( `.print(``Mytext'')` ).

Cada agente Jason posee un ciclo de razonamiento que determina qué hacer y cómo hacerlo en cada momento. Este ciclo permite crear intenciones basadas en los planes que tiene el agente pendientes de ejecución. En cada ciclo de razonamiento básicamente el agente debe interactuar con el entorno, verificar la cola de los mensajes que se le han enviado, determinar si existen planes aplicables de acuerdo a los eventos que se hayan generado e intentar ejecutarlos. Tanto la percepción del entorno como la recepción de mensajes producen cambios en la base de creencias y generan eventos. A partir de los eventos se extrae un conjunto de planes “relevantes”, que pasan a ser “aplicables” cuando, al verificarse su contexto, éste es una consecuencia lógica de la base de creencias. Estos planes se convierten en intenciones que son posteriormente procesadas de acuerdo a un criterio de selección.

### 3.2.3. Integración de Jason en Magentix2

Jason permite el uso de distintas infraestructuras para la ejecución del sistema multi-agentes. De hecho, en su implementación por defecto, es posible emplear las infraestructuras *Centralised*, *Saci* y *Jade* pero también ofrece las herramientas necesarias para incorporar otras. Partiendo de ello este lenguaje se ha integrado a Magentix2 dotándolo de la capacidad para desarrollar agentes BDI. Para definir un agente Jason que se ejecute sobre Magentix2 es necesario en primer lugar contar con el código del agente Jason que regirá su comportamiento. En segundo lugar es necesario definir la “arquitectura del agente”. Esta arquitectura permitirá establecer la interacción del agente con el entorno y también las AI; además permite modificar algunos

[1] Esta es la estructura básica para la comunicación de agentes. En Jason se emplea KQML (Knowledge Query and Manipulation Language) para el intercambio de mensajes y las performativas que se emplean son similares. Estas son: *tell*, *untell*, *achieve*, *unachieve*, *askOne*, *askAll*, *tellHow*, *untellHow askHow*. Para una descripción más detallada referirse a [9].

comportamientos del agente como las acciones a realizar cuando se reciben mensajes o cuando se envían. Magentix2 proporciona una arquitectura por defecto que puede ser extendida en este sentido. Un agente Jason es también un agente Magentix2 y por tanto puede hacer uso de las herramientas de seguridad, trazas, protocolos de interacción y en general todas las posibilidades que éste ofrece [17].

## 3.3. Descripción del diseño para la incorporación de conversaciones en Jason

### 3.3.1. Directrices

Dadas las necesidades existentes de mejorar el proceso de comunicación entre agentes en un sistema multi-agente (SMA) el primer objetivo de esta propuesta es facilitar la tarea de la programación de esta parte del agente. Se pretende que los agentes no tengan prácticamente "conciencia" de que su razonamiento es parte de una conversación bajo un protocolo determinado, y que no necesiten saber cuál ha sido o cuál es el próximo paso en cualquier punto del protocolo. Dado este nivel de abstracción, el programador del agente solo necesita enfocarse en el razonamiento en lugar de emplear primitivas de comunicación específicas o los detalles de la especificación del protocolo para gestionarlo. Por otra parte, teniendo las ampliamente aceptadas especificaciones de protocolos de interacción de FIPA, se pretende integrarlas con un lenguaje de programación de alto nivel (Jason en este caso) para lograr este objetivo. Esta propuesta ha sido realizada por primera vez en [6] y el trabajo presente se extiende con más nivel de detalle.

Esta es una idea que puede ser extrapolada a otros lenguajes de programación o plataformas siguiendo el mismo proceso. Esto le evita al programador el tener que aprender nuevas construcciones para emplear los protocolos de interacción y convierte a la propuesta en lo suficientemente versátil para facilitar esta tarea independientemente de las herramientas, plataforma o lenguaje empleado.

### 3.3.2. Ideas básicas

Según las especificaciones de FIPA un protocolo de interacción puede verse como una secuencia de mensajes entre dos roles: el *iniciador* y el *participante*. El *iniciador* comienza la interacción notificando al *participante* con un mensaje acerca de sus intereses y el participante

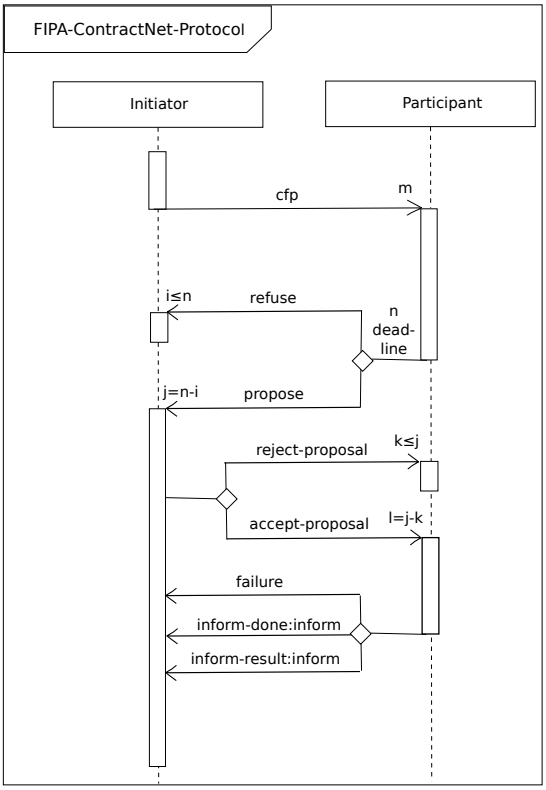


**Figura 3.1:** Protocolos de interacción FIPA Contract Net. [19]

responde a la solicitud de acuerdo a las características del protocolo. Ambos roles intercambian mensajes hasta el final de la interacción. Por ejemplo la figura 3.1 muestra la especificación del protocolo Contract Net de FIPA basada en una extensión de UML para representar protocolos de interacción [25].

De acuerdo a [18], una conversación puede ser vista como un grafo dirigido compuesta de nodos y arcos; los nodos representan estados o pasos dentro de la conversación y los arcos las transiciones entre los estados. Partiendo de esto el objetivo sería integrar estos pasos de la comunicación en el ciclo de razonamiento del agente BDI. Primeramente es necesario conocer cuales son estos pasos del protocolo. La Figura 3.2 muestra un ejemplo de los pasos del protocolo Contract Net desde la perspectiva de *iniciador* de acuerdo con [19] visto como un grafo dirigido. En esta figura se observan algunos nodos resaltados. Estos representan aquellos pasos de la conversación en los que el agente puede realizar su razonamiento o tomar decisiones. A partir de aquí el próximo paso sería identificar, para cada rol en cada protocolo de

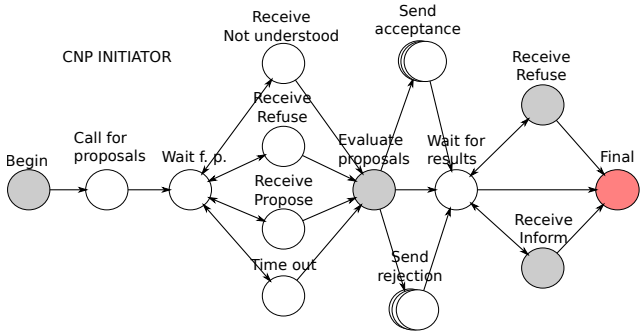


**Figura 3.2:** Pasos del protocolo Contract Net de FIPA para el rol *iniciador*.

comunicación, en qué pasos es necesario realizar un razonamiento o tomar una determinada decisión.

En cada paso un *Gestor de Conversaciones* (GC) indicará al agente que es necesario realizar algún razonamiento. La conversación se detendrá hasta que se proporcione algún resultado a través de una interacción entre el agente y el GC o hasta que transcurra un tiempo de espera determinado. Esto no afectará el ciclo de vida y de razonamiento del agente, que continuará su ejecución independientemente de la conversación. La información que se proporcione al agente en los diferentes pasos de la comunicación actuará como evento disparador para la ejecución de un plan o conjunto de acciones al mismo tiempo que proporcionará al agente los datos necesarios para que el agente razone sobre ellos. En aquellos pasos en los que no se realiza ningún razonamiento el GC seguirá con la secuencia de pasos indicada en la especificación del protocolo.

En el paso final del protocolo el GC finaliza la conversación realizando las tareas necesarias en este sentido. El agente es informado de la culminación de la conversación y de sus resultados. Por otra parte el agente puede finalizar la conversación en cualquier punto en caso de algún fallo o circunstancia específica y de la misma forma, si algo relacionado con la conversación falla internamente (por ejemplo un tiempo de respuesta excedido o algún error de comunicación), el GC finaliza la conversación informando al agente.

## 3.4. Entidad gestora de conversaciones

El soporte a los protocolos de interacción FIPA por la plataforma Magentix2 está dado principalmente por las *Fábricas de Conversaciones*[18], ofreciendo las herramientas necesa-

rias para ello y para poder modificarlos en tiempo de ejecución. La *Entidad Gestora de Conversaciones* o gestor de conversaciones (GC) en este caso es la interfaz entre las fábricas de la plataforma Magentix2 y el lenguaje de programación de agentes Jason, de manera que se puedan implementar los protocolos de interacción de FIPA de una manera más sencilla desde un nivel de abstracción más alto. En este caso el GC emplea las conversaciones asíncronas de la plataforma ya que cuando es necesario sincronizar una conversación esta tarea es delegada en el GC. De la misma forma no se emplean las conversaciones anidadas que proporciona la plataforma ya que esto se realiza a nivel del agente y es controlado internamente por el GC.

Como se ha explicado en la sección 3.3.2, cuando un razonamiento es necesario, la conversación se detiene a la espera de un evento para continuar (el agente proporciona resultados o transcurre un tiempo de espera). Para lograr esto se han incluido en el GC una serie de estructuras y además un conjunto de AI que permitirán realizar esta tarea.

En esencia las AI en Jason se implementan como clases de Java que deben estar incluidas en un paquete Java. Estas deben implementar la interfaz InternalAction, para la cual existe una implementación por defecto llamada DefaultInternalAction que simplifica la creación de nuevas AI. Otro aspecto importante al respecto es que cada agente crea una instancia de la clase correspondiente a la AI la primera vez que es ejecutada y reutiliza esta misma instancia en las subsecuentes ejecuciones de esta AI por el agente. Partiendo de estas características para cada protocolo de interacción se han definido dos AI: una para el rol *iniciador* y una para el rol *participante*. La primera vez que un rol ejecute un protocolo de interacción se creará una instancia de la AI correspondiente en dicho agente, y en subsecuentes ejecuciones se empleará esta misma instancia de la AI ya sea, para iniciar una nueva conversación con este protocolo o para continuar alguna conversación ya iniciada anteriormente.

Cada AI tendrá un registro de las conversaciones que ha creado el agente haciendo uso del protocolo correspondiente. Cuando el agente emplee la AI para iniciar una nueva conversación o proporcionar la información relacionada con uno de los pasos de la conversación, deberá suministrar un identificador para dicha conversación así como el paso de la conversación al que está asociada la información. Si es el paso inicial, y el agente es el *iniciador*, la AI registrará e iniciará una nueva conversación en la plataforma; en cambio, si es el *participante*, registrará y se unirá a dicha conversación como participante.

Cada conversación registrada por la AI tiene un identificador proporcionado por el agente y uno asignado por la plataforma. Así mismo tiene una serie de datos que representan el estado de la conversación en cada momento. Esto lo posibilita la clase Conversation que puede

ser extendida para representar una conversación de un protocolo determinado. De esta forma también existe una clase hija de Conversation para cada conversación cuyas instancias serán visibles tanto desde las AI como del *CProcessor* correspondiente en la plataforma. Otro aspecto importante a destacar es que la clase Conversation contiene también un semáforo que permitirá al GC pausar la conversación cuando se requiera un razonamiento por parte del agente o a la AI continuarla cuando el agente la haya invocado proporcionando la información necesaria. La manera en la que el agente es notificado acerca del paso sobre el que debe razonar y la información asociada para que lo haga es proporcionada por el GC accediendo a la instancia de la clase Conversation correspondiente y añadiendo esta información a la base de creencias del agente.

Al GC también lo conforman clases personalizadas para: i) especificar la arquitectura del agente Jason, ii) el agente Jason en la plataforma, iii) *CProcessor* de las conversaciones y iv) *CFactory* de las conversaciones. Todas ellas extienden de las clases ofrecidas por la plataforma para gestionar las conversaciones agregando los elementos necesarios para permitir el uso de la fábrica de conversaciones desde los agentes Jason.

En el acápite 3.5 se ofrece un resumen de los pasos necesarios para implementar protocolos de interacción en un agente Jason sobre Magentix2.

## 3.5. Creación de conversaciones en agentes Jason sobre Magentix2

De manera general para implementar una conversación sobre cualquier protocolo en un agente Jason que se ejecuta en la plataforma Magentix2, ya sea como *iniciador* o como *participante* es necesario tener en cuenta los siguientes pasos [6]:

1. Asegurarse de que el agente tenga acceso a, o esté en su base de creencias, la información requerida para iniciar la conversación. Por ejemplo, en el caso del *iniciador* de un protocolo Contract Net es necesario conocer el tiempo de espera para las respuestas de la otra parte y el nombre de los participantes con los que desea interactuar.

2. Implementar un plan que contenga la primera llamada a la AI correspondiente al protocolo que se va a emplear (en el caso del rol *iniciador*). Para el caso del *participante*, este debe tener un plan para unirse a la conversación cuando reciba la solicitud del *iniciador*.

3. Implementar un plan para cada paso del protocolo donde es necesario realizar un razonamiento o tomar una decisión: para ello se debe contar con las estructuras predefinidas en

forma de creencias que son añadidas al agente cuando se necesita este razonamiento, y que actúan como eventos disparadores para la ejecución de los planes correspondientes. La acción final para cada uno de estos planes debe ser siempre una llamada a la AI del protocolo, para el rol correspondiente en la conversación, con los parámetros apropiados. Esto permite que la conversación continúe al próximo paso[1]. La AI tiene la estructura: `.ia_<protocolo>_Initiator(...)` ó `.ia_<protocolo>_Participant(...)` en dependencia del rol y del protocolo. El número y tipo de los parámetros varía, pero por normal general, el primer parámetro siempre debe ser un texto que identifica el paso de la conversación y el último un valor de cadena, literal o numérico que representa el identificador de la conversación.

4. Implementar, de manera opcional, un plan para cada paso de razonamiento adicional; generalmente, este paso siempre resulta ser el paso final. De manera que, por ejemplo, si se realiza alguna gestión con los identificadores de las conversaciones, o es necesario realizar algún procesamiento al finalizar la conversación, este debe implementarse en este paso.

## 3.6. Ejemplo de un agente Jason Conversacional

Para ilustrar cómo se usan las conversaciones desde un agente Jason se tomará como referencia una adaptación al ejemplo del robot dispensador de cerveza propuesto en [9], que viene integrado en la distribución del paquete **Jason**. El ejemplo se basa en lo siguiente:

> "Un robot doméstico tiene como objetivo servir cerveza a su dueño. Su misión es recibir solicitudes de su dueño, ir a la nevera, tomar una cerveza y llevársela a su dueño. No obstante el robot debe preocuparse por el stock de cerveza (y eventualmente ordenar más cerveza al servicio de entrega a casa del mercado) y por algunas reglas incorporadas al robot por el departamento de salud (en este caso la regla define el límite diario del consumo de cerveza").

En nuestro ejemplo existen básicamente tres agentes: el robot, el dueño y el mercado. Existe además un agente adicional que actúa como "entorno" y realiza las acciones que se realizarían

[1] Actualmente el GC posee estructuras definidas para ser empleado por agentes Jason para los protocolos de FIPA: Contract Net, Query-If, Query-Ref, Request, Subscribe, Recruiting y Japanese Auction

en el entorno en el ejemplo original [1]. Las principales acciones que pueden realizar los agentes son las siguientes:

- **robot**
  - *at(robot,fridge)*: Para dirigirse a la nevera.
  - *at(robot,owner)*: Para dirigirse al dueño.
  - *open(fridge)*: Para abrir la nevera.
  - *get(beer)*: Para tomar la cerveza.
  - *close(fridge)*: Para cerrar la nevera.
  - *hand_in(beer)*: Para entregar la cerveza.
  - *order(beer)*: Para ordenar más cerveza al mercado.
- **dueño**
  - *get(beer)*: Para pedir al robot más cerveza.
  - *sip(beer)*: Para beber un sorbo de cerveza.
- **mercado**
  - *deliver(beer)*: Para entregar N unidades de cerveza.
  - *delivered(beer,N,OrderId)*: Para informar de la entrega realizada.

Tanto la acción *at(robot,fridge)* como *at(robot,owner)* se han simplificado de manera que solo implican cambiar la creencia del robot sobre la posición donde se encuentra. La solicitud de más cerveza al mercado por parte del robot se ha adaptado para nuestro ejemplo como una conversación bajo el protocolo Request de FIPA. De manera que se explicará cómo funciona el GC en el contexto del ejemplo.

Se partirá de que el objetivo inicial del dueño es obtener una cerveza. Se cuenta con un stock de dos cervezas en la nevera y cada vez que éste se termine se solicitará al mercado 5 unidades más. Cuando el robot recibe la solicitud del dueño éste realiza las acciones pertinentes para hacerle llegar la cerveza siempre y cuando no haya alcanzado el límite permitido de

[1] Jason ofrece la posibilidad de interactuar con un *entorno* a través de clases Java predefinidas para ello. En el ejemplo original este recurso se emplea para mostrar el estado de la ejecución de manera visual a través de una interfaz gráfica que se actualiza constantemente. En nuestro ejemplo se han representado estas funciones a través de un agente para mayor simplicidad.

cervezas consumidas establecido por el departamento de salud. El dueño bebe la cerveza (siempre realizará 10 sorbos por cada una) y al terminarse volverá a solicitar más cerveza al robot. Cuando el mercado reciba la solicitud por parte del robot para más cerveza, éste la procesará y realizará la entrega. A continuación se muestra el código para los agentes: dueño, robot y mercado. En el apéndice A.2 se muestra el código para el agente que simula el entorno y se explica a través de comentarios.

**Agente dueño**

```
1  /* Objetivos iniciales */
2
3  !get(beer).   // objetivo inicial: obtener cerveza
4
5  +!get(beer) : true
6     <- .send(robot, achieve, has(owner,beer)).
7
8  +has(owner,beer) : true
9     <- !drink(beer).
10 -has(owner,beer) : true
11    <- !get(beer).
12
13 // mientras tenga cerveza, bebo
14 +!drink(beer) : has(owner,beer)
15    <- !sip(beer);
16       .wait(300);
17       !drink(beer).
18 +!drink(beer) : not has(owner,beer)
19    <- true.
20
21 +msg(M)[source(Ag)] : true
22    <- .print("Message from ",Ag,": ",M);
23       -msg(M).
24
25 /*----------------- acciones con el entorno -------------------*/
26
27 +!sip(beer)[source(self)]
28    <-  .send(environmentAg,achieve,sip(beer)).
```

El agente dueño tiene un objetivo inicial que es obtener cerveza (línea 3). De manera que cuando comienza la ejecución se genera el evento `+!get(beer)`. Este evento hace que se

dispare el plan de las líneas 5-6. En este caso el agente solicitará al robot mediante un mensaje con la performativa *achieve* que consiga el objetivo `has(owner,beer)` . Una vez el robot ha logrado esto una percepción `has(owner,beer)` le es enviada al agente dueño por el entorno lo que disparará el plan de la línea 8-9. Este plan crea a su vez el objetivo `!drink(beer)` que conlleva a que el dueño comience a beber a través del plan recursivo de la línea 14-17. El ciclo termina cuando la creencia `has(owner,beer)` ya no esté, y esta se elimina cuando el agente entorno detecta que se terminaron los sorbos y se lo notifica al dueño (referirse a apéndice A.2). El plan que se ejecutaría en tal caso es el de las líneas 18-19, en cuyo contexto se verifica que no exista dicha creencia. Al mismo tiempo, la eliminación de la creencia `has(owner,beer)` genera un evento que dispara el plan de las líneas 10-11, el cual permite que el objetivo `get(beer)` para beber cerveza vuelva a ser añadido y comience el ciclo desde el inicio. El plan de las líneas 21-23 imprime un mensaje en pantalla cuando algún agente `Ag` le envía un mensaje con el formato `msg(M)` [1]. Al final del plan se elimina esta creencia ya que en otro caso no se generaría el evento `+msg(M)` si un mensaje idéntico llegase, porque la creencia ya estaría en la base de creencias. El plan de las líneas 27-28 permite al agente dueño notificar al entorno a través de un mensaje, que un sorbo de cerveza ha sido bebido.

**Agente robot**

```
1  /* Objetivos y reglas iniciales */
2
3  // inicialmente creo que hay cerveza en la nevera
4  available(beer,fridge).
5
6  // inicialmente mi posición es en la nevera
7  at(robot,fridge).
8
9  // mi dueño no debe consumir más de 10 cervezas al día
10 limit(beer,10).
11
12 too_much(B) :-
13    .date(YY,MM,DD) &
14    .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
15    limit(B,Limit) &
16    QtdB >= Limit.
```

[1] El término `source(Agente)` es una anotación del plan y permite saber quién ha generado la adición de este objetivo. Cuando el valor de `Agente` es "self" esto indica que el objetivo ha sido añadido por el propio agente. Para más información sobre anotaciones referirse a [9].

```
17
18
19  /* Planes */
20
21  +!has(owner,beer)
22     : available(beer,fridge)& not too_much(beer)
23     <- !at(robot,fridge);
24        !open(fridge);
25        !get(beer);
26        !close(fridge);
27        !at(robot,owner);
28        !hand_in(beer);
29        .date(YY,MM,DD); .time(HH,NN,SS);
30        +consumed(YY,MM,DD,HH,NN,SS,beer);
31        .count(consumed(YY,MM,DD,_,_,_,B),QtdB);
32        .print("Consumed: ",QtdB).
33
34  /*============== solicitud de cerveza al meracdo ===============*/
35  /* conversation plan: START ------------------------------ */
36  +!has(owner,beer)
37     :  not available(beer,fridge)
38     <-  .time(H,M,S);
39         .concat("",H,".",M,".",S,".deliverBeer",ConvID);
40         +conversationID(robot,frp,ConvID);
41         .send(supermarket,achieve,join(ConvID,Protocol));
42         .print(" * Starting and making proposal to supermarket * ");
43         TO = 2000;
44         .ia_fipa_request_Initiator("start", supermarket , TO,
45             "Fipa request conversation started",ConvID);
46         .ia_fipa_request_Initiator("request",beer,supermarket,
47             data(count(beer,5)), ConvID).
48  /*---------------------------------------------------------- */
49
50  /* conversation plan: TASK DONE -------------------------- */
51  +taskdonesuccessfully(P,Result,ConvID)
52     <-  .print(" * The task ",ConvID,
53            " was done successfully by agent ",P,". Result: ",
54            Result," * ");
55         +available(beer,fridge);
56         !has(owner,beer);
57         .ia_fipa_request_Initiator("taskdone",ConvID).
```

```
58 /*------------------------------------------------------- */
59
60 /* conversation plan: END ------------------------------- */
61 +conversationended(ConvID, Result)
62   :  conversationID(robot,frp,ConvID)
63    <-   -conversationID(robot,frp,ConvID).
64 /*-------------------------------------------------------*/
65 /*=======================================================*/
66
67 +!has(owner,beer)
68    :  too_much(beer) & limit(beer,L)
69    <- .concat("The Department of Health does not allow me to give
       you more than ", L, " beers a day! I am very sorry about that!
       ",M);
70       .send(owner,tell,msg(M)).
71
72 -!has(_,_)
73    :  true
74    <- .current_intention(I);
75       .print("Failed to achieve goal '!has(_,_)'. Current intention
          is: ",I).
76
77 +!at(robot,P) : at(robot,P) <- true.
78 +!at(robot,P) : not at(robot,P)
79   <- .print("Moving to ",P);
80      !move_towards(P).
81
82 // cuando la nevera está abierta se percibe el stock
83 // y se actualiza la creencia 'available'
84 +stock(beer,0)
85    :  available(beer,fridge)
86    <- -available(beer,fridge);
87       .print("The beer in stock has fnished.");
88       .abolish(stock(beer,_)).
89 +stock(beer,N)
90    :  N > 0 & not available(beer,fridge)
91    <- -+available(beer,fridge) ;
92       .abolish(stock(beer,_)).
93
94 +!move_towards(P)
95    <- -+at(robot,P).
```

```
96
97  /*---------------- Environment interactions ----------------------*/
98  +!open(fridge)
99     <- .send(environmentAg,achieve,open(fridge)).
100
101 +!get(beer)
102    <- .send(environmentAg,achieve,get(beer)).
103
104 +!close(fridge)
105    <- .send(environmentAg,achieve,close(fridge)).
106
107 +!hand_in(beer)
108    <- .send(environmentAg,achieve,hand_in(beer)).
```

El robot es el principal agente del ejemplo. Manteniendo la creencia `available(beer,fridge)` puede saber que aún hay cerveza disponible y cuando ya no esté deberá solicitar más al mercado. Inicialmente cree que hay cerveza (línea 4) y también cree que está posicionado en la nevera (línea 7). Además tiene especificado un límite de cervezas que el dueño puede beber indicado por el departamento de salud que es 10 cervezas en este caso (línea 10). Por otra parte la regla de las líneas 12-16 le permite al robot saber si la cantidad consumida en el mismo día (creencias con el formato `consumed(Año,Mes,Día,Hora,Minuto,Segundo,beer)` ) sobrepasa el límite permitido. En la regla se emplea la AI `.date` para conocer la fecha actual y `.count` para saber el número de creencias con un formato determinado. El término '_' permite especificar que en su lugar se espera cualquier valor, y en la variable `QtdB` se almacenaría esta cantidad consumida. La regla se cumpliría si `QtdB` es mayor que 10 en este caso.

La principal tarea del robot, que es alcanzarle cerveza al dueño, se lleva a cabo cuando el objetivo `has(owner,beer)` es agregado. Este objetivo se alcanza a través de los tres planes de las líneas 21-32, 36-47 y 67-70. El primer plan se dispara cuando existe cerveza disponible y no se ha bebido demasiado (ver contexto del plan). En este plan el robot realiza las acciones necesarias para alcanzarle cerveza al dueño, añadiendo subobjetivos que irán disparando los planes correspondientes. Primero el robot desea dirigirse a la nevera (línea 23) y lo consigue a través de los planes de las líneas 77 y 78-80. Si ya está en la nevera (creencia `at(robot,P)` donde `P=nevera` ) no realiza ninguna acción (línea 77); si no está se actualiza su posición (línea 95) a través del subjetivo `move_towards(P)` que dispara el plan de la línea 94. Luego de moverse a la nevera el robot debe abrir la nevera (subobjetivo `open(fridge)` de la línea 24); este subobjetivo dispara el plan de la línea 98-99 para notificar esta acción al entorno.

Seguidamente el robot toma la cerveza a través del plan de las líneas 101-102 que igualmente notifica esta acción al entorno. Este plan es disparado por el subobjetivo `get(beer)` de la línea 25. Como consecuencia de esta acción el entorno decrementará la disponibilidad y en caso de que esta llegue a cero le notificará al robot que no hay cerveza disponible (apéndice A.2). Luego se agrega el subobjetivo para cerrar la nevera `close(fridge)` que actúa como los anteriores a través del plan de las líneas 104-105. En este paso el entorno notifica al robot del stock existente a través de un mensaje con el contenido `stock(beer,AvB)` siendo `AvB` la cantidad restante. Los subobjetivos `at(robot,owner)` (línea 27) y `hand_in(beer)` (línea 28) actúan de manera similar a los anteriores a través de los planes de las líneas 77-80 y 107-108 y permiten que el robot se mueva hacia el dueño y le entregue la cerveza. Finalmente se añade una creencia asociada al consumo de una nueva cerveza (línea 30) en el día actual proporcionado por la AI `.date` y en la hora actual proporcionada por la AI `.time` (línea 29). Además se imprime un mensaje con la cantidad de cerveza que el dueño ha consumido en ese día (línea 32) obtenido a través de la AI `.count` (línea 31).

Las líneas de la 36 a la 63 corresponden a la estructura requerida para iniciar una conversación con el mercado a través del GC solicitando más cerveza. Esto será explicado más adelante.

El plan de las líneas 67-70 se dispara en caso de que el dueño haya consumido más cerveza de lo permitido (ver contexto). En ese caso se imprime un mensaje en pantalla y se le notifica esto al dueño mediante un mensaje con la performativa *tell*. Si durante la ejecución de alguno de los planes cuyo evento disparador es `has(owner,beer)` ocurre algún fallo, se dispararía el plan de contingencia de las líneas 72-75 (referirse a [9] para más información) que simplemente imprime la intención actual del agente, obtenida a partir de la AI `.current_intention` (línea 74), como un mensaje en pantalla (línea 75).

Como ha sido mencionado, cuando en el entorno se realizan las acciones para cerrar la nevera, el robot es notificado del 'stock' disponible de cerveza a través de un mensaje con el contenido `stock(beer,AvB)` . Este mensaje añade una creencia `stock(beer,AvB)` en la base de creencias del robot, y esto a su vez genera un evento que disparará el plan de las líneas 84-88 o 89-92 en dependencia de si el 'stock' es cero (primer plan) o si es mayor que cero (segundo plan). Si se ejecuta el primer plan, no quedaría cerveza disponible con lo que es necesario eliminar la creencia `available(beer,fridge)` (línea 86). Además en este plan

se eliminan las creencias del tipo `stock(beer,_)` mediante la AI `.abolish` [1]. Si se ejecuta el segundo se confirma que aún hay cerveza disponible (línea 91) y de igual forma se eliminan las creencias del tipo `stock(beer,_)` .

**Agente mercado**

```
1  /* Creencias iniciales */
2  // identificador de la última orden de compra: 1
3  last_order_id(1).
4  // productos que se venden, en este caso, solo cerveza
5  products([beer]).
6
7  /* Planes */
8
9  /*============ ANSWER ROBOT REQUEST FOR MORE BEER =============*/
10 /* conversation plan: START ------------------------------ */
11 +!join(ConvID,Protocol)[source(robot)]
12    <-   TO = 1000;
13         .ia_fipa_request_Participant("joinconversation",TO,ConvID).
14 /*-------------------------------------------------------------*/
15
16 /* conversation plan: AGREE ------------------------------ */
17 +request(Sender,Content,Data,ConvID)
18   : Sender=robot & last_order_id(N) & products(Prod) &
19     .member(Content,Prod)
20    <-  .print(" - I've received a request for doing: ",Content,
21           " and I'm AGREE. - ");
22        -+orderStatus(Content,agree);
23        .ia_fipa_request_Participant("agree",ConvID).
24 /*-------------------------------------------------------------*/
25
26 /* conversation plan: REFUSE ------------------------------ */
27 +request(Sender,Content,Data,ConvID)
28   : not Sender=robot & ast_order_id(N) & products(Prod) &
29     .member(Content,Prod)
30    <-  .print(" - I've received a request for doing: ",Content,
31           " and I REFUSE. - ");
32        -+orderStatus(Content,refuse);
```

[1] Puede darse el caso de que se añadan dos creencias `stock(beer,AvB)` donde el valor de AvB coincida, con lo que las creencias serían iguales. En ese caso no se dispararía este plan.

```
33          .ia_fipa_request_Participant("refuse",ConvID).
34  /*-------------------------------------------------------------*/
35
36  /* conversation plan: NOT UNDERSTOOD ---------------------- */
37  +request(Sender,Content,Data,ConvID)
38    : Sender=robot & last_order_id(N) & products(Prod) &
39      not .member(Content,Prod)
40     <-  .print(" - I've received a request for: ",Content,
41            " but I don't understand. - ");
42          -+orderStatus(Content,notunderstood);
43          .ia_fipa_request_Participant("notunderstood",ConvID).
44  /*-------------------------------------------------------------*/
45
46  /* conversation plan: FAILURE (DEFAULT REQUEST HANDLING) -- */
47  // si fallan los planes anteriores
48  +request(Sender,Content,Data,ConvID)
49     <-  .print(" - I've received a request for doing: ",Content,
50           " but I've FAILED. - ");
51          -+orderStatus(Content,failure);
52          .ia_fipa_request_Participant("failure",ConvID).
53  /*-------------------------------------------------------------*/
54
55  /* conversation plan: INFORM ------------------------------ */
56  +timetodotask(Content,Data,ConvID)
57    :  orderStatus(Content,agree) & last_order_id(N)
58     <-  if (Data=data(count(P,C)))
59          { Product = P ; Qtd = C; }
60          else{ Product = beer ; Qtd = 5;};
61          OrderId = N + 1;
62          -+last_order_id(OrderId);
63          !deliver(Product,Qtd);
64          Result = delivered(Product,Qtd,OrderId);
65          +taskResult(Content,ConvID,Result);
66          .print(" - Informing of task done - ");
67          .ia_fipa_request_Participant("inform",Content,
68                 Result,ConvID).
69  /*-------------------------------------------------------------*/
70
71  /* conversation plan: FAILURE ----------------------------- */
72  +timetodotask(Content,Data,ConvID)
73     <-  .print(" - I've failed doing the task: ",Content," - ");
```

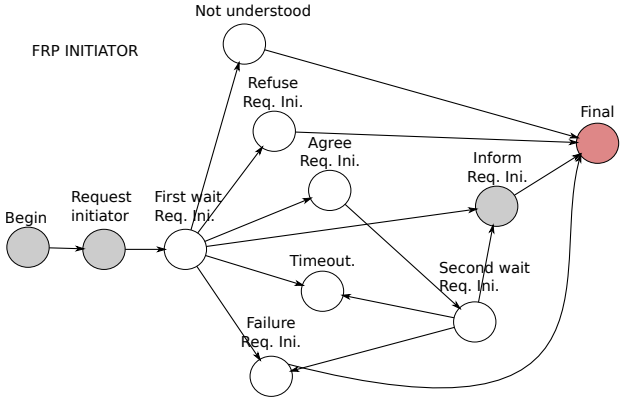


**Figura 3.3:** Pasos del protocolo Request de FIPA para el rol *iniciador*.

```
74          .ia_fipa_request_Participant("failure",Content,ConvID).
75  /*-------------------------------------------------------------*/
76  /*=============================================================*/
77
78  /*----------------- Environment interactions ---------------*/
79
80  +!deliver(Product,Qtd)
81     <-   .send(environmentAg,achieve,deliver(Product,Qtd)).
```

El agente mercado tiene una creencia inicial que representa el número de la última orden procesada (línea 3) y los productos que entrega (línea 5), que en este caso es solo cerveza. Además posee las estructuras necesarias para responder como participante en la conversación que sostiene con el robot cuando éste le solicita mas cerveza (líneas 11-74). A continuación se describirán los detalles de esta interacción.

### 3.6.1. Interacción robot-mercado a través del protocolo Request de FIPA

Como ha sido descrito, la conversación entre el robot y el mercado en la que se emplea el GC sigue el protocolo Request de FIPA y se inicia en el momento en el que el robot detecta que necesita reponer el stock de cerveza para su dueño. De manera que el robot juega un rol de *iniciador* en esta conversación y el mercado tiene el rol de *participante*. En este sentido las figuras 3.3 y 3.4 muestran los pasos correspondientes al protocolo Request para estos roles a través de grafos dirigidos. De manera genérica este protocolo se comporta en el GC de la siguiente forma.

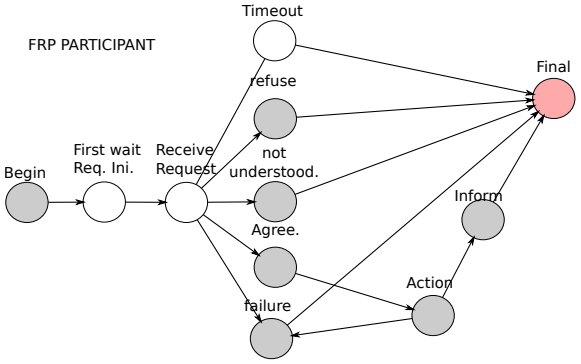


**Figura 3.4:** Pasos del protocolo Request de FIPA para el rol *participante*.

En el caso del *iniciador*, después del paso inicial *Begin*, éste realiza la solicitud al participante y pasa a un estado *Wait* en el que se queda esperando por la respuesta. De este estado se pasa ya sea a *Not understood* (el participante no ha comprendido la solicitud recibida), *Refuse* (rechaza la solicitud), *Agree* (está de acuerdo con la solicitud), *Failure* (se produce algún fallo en el participante tras recibir la solicitud) o *Timeout* (no se ha recibido respuesta del participante tras haber transcurrido un tiempo desde el envío de la solicitud). Si el participante está de acuerdo se pasaría a un segundo estado de espera del cual se sale si se recibe la información relacionada con el procesamiento de la solicitud o si se ha producido algún fallo en el participante o tras haber transcurrido un tiempo de espera; en cualquier otro caso la conversación terminaría. Como se observa en la figura 3.3, los estados *Begin*, *Request*, *Inform* y *Final* están señalados de un color (referirse a la sección 3.3.2). Esto significa que en esos estados se requiere algún procesamiento o información del agente Jason. En el caso del estado *Final*, se indica de un color diferente porque el razonamiento por parte del agente Jason es opcional.

El grafo de la figura 3.4 muestra los pasos del mismo protocolo para el rol *participante*. En este caso, una vez que el participante se une a la conversación, éste pasa a un estado *Wait* en el que espera por la solicitud. Una vez recibida pasa a uno de los cinco estados descritos para el rol *iniciador*, en este caso como remitente. En el caso de estar de acuerdo con la solicitud recibida pasa a un estado *Action* en el que se realizan las acciones correspondientes a la solicitud, y luego se pasa a *Inform* o *Failure* en dependencia de si se han realizado las acciones con éxito o no. De igual forma los estados destacados con un color implican un procesamiento por parte del agente Jason.

Volviendo al ejemplo, como se observa en el código del robot, existen tres planes para par-

ticipar en la conversación sobre el protocolo Request para solicitar cerveza al mercado (líneas 36-63). Estos planes corresponden a los pasos del protocolo *Begin*, *Request*, *Inform* y *Final*.

El primer plan (lineas 36-47), se dispara cuando es añadido el objetivo `has(owner,beer)` para llevarle cerveza al dueño cumpliéndose que no existe cerveza disponible y no se ha sobrepasado el límite permitido (ver contexto del plan). Este plan contiene las acciones necesarias para comenzar la interacción y realizar la solicitud al mercado (pasos del protocolo *Begin* y *Request*). Las dos primeras acciones del plan (líneas 38 y 39) permiten conformar un identificador para la conversación que será adjuntado en todos los planes para la conversación tanto en el *participante* como en el *iniciador*; la primera es la AI `.time` que permite recuperar la hora, minuto y segundo actuales en las variables `H`, `M` y `S`; la segunda es la AI `.concat` con la que se concatenan estos valores y una cadena separados por un punto en la variable `ConvID` [1] (en lo adelante se empleará esta variable para almacenar este valor). A continuación se añade una creencia en forma de literal que sirve para que el agente recuerde que está participando en una conversación y contiene el nombre del agente, un identificador para el protocolo a emplear y el identificador de la conversación (línea 40). En la línea 41 se envía una invitación al mercado para unirse a la conversación a través de un mensaje con con la performativa `achieve`, que permite generar un evento de adición de un *achievement goal* en el receptor del mensaje y ejecutar así un plan con este evento como disparador. Seguidamente se imprime un mensaje en pantalla (línea 42) y se almacena en la variable `TO` la cantidad de milisegundos deseados (2000 en este caso) para los tiempos de espera de las respuestas de la otra parte(línea 43). Finalmente en las líneas 44 y 46 se ejecutan las AI correspondientes para interactuar con el GC. La primera corresponde al paso *Begin* del protocolo y se identifica con la cadena "start"; se pasa como parámetro también en esta AI en ese orden: el agente *participante*, el tiempo de espera, un mensaje inicial y un identificador de conversación. El la segunda AI en cambio, correspondiente al paso *Request* del protocolo, se pasa la cadena "request" para identificar el paso, un identificador para la solicitud que se está realizando, el agente *participante*, la información asociada a la solicitud y el identificador de conversación en ese orden.

El segundo plan (líneas 51-57) tiene como objetivo realizar el procesamiento necesario tras recibir un *Inform* por parte del mercado. Esto implica que éste ha estado de acuerdo con la solicitud y ha realizado las acciones pertinentes para responder a esa solicitud. El GC añade la creencia `taskdonesuccessfully(P,Result,ConvID)` a la base de creencias del robot,

[1] Esta es una estructura abierta. Puede ser especificado cualquier valor que pueda identificar una conversación de otra.

donde `P` es el nombre del *participante* (mercado en este caso), y `Result` es el resultado del procesamiento del *participante*. En este plan básicamente se imprime un mensaje en pantalla (línea 52-54), se añade la creencia `available(beer,fridge)` (línea 55) para tener en cuenta que ya hay cerveza disponible tras la confirmación positiva del mercado, se añade el objetivo `has(owner,beer)` para que el robot vuelva a intentar darle cerveza al dueño (línea 56) y se invoca a la AI para el paso *Inform* (línea 57).

El último plan del robot para la conversación (líneas 61-63) corresponde al paso final del protocolo. Su especificación es opcional, pero se recomienda tenerlo en cuenta para realizar acciones de "limpieza" asociadas a la conversación. De igual forma que el anterior este plan se dispara tras la adición de la creencia *conversationended(ConvID, Result)* por el GC. En el contexto se verifica la existencia de la creencia que identifica a la conversación que es posteriormente eliminada en el cuerpo del plan (línea 63).

En el lado del del *participante* (en este caso el mercado) también se deben definir los planes necesarios para interactuar con el GC y poder recibir y proporcionar la información requerida cuando sea necesario. En las líneas 11-74 del código del agente mercado se definen los planes para los pasos del protocolo *Begin*, *Agree*, *Refuse*, *Not understood*, *Failure*, *Action* e *Inform*. El primer plan (línea 11-13) responde a la invitación enviada por el robot para unirse a la conversación. Para ello se ejecuta la AI correspondiente (línea 13).

Los planes de las líneas 17 a la 52 tienen como evento disparador la adición de la creencia `+request(Sender,Content,Data,ConvID)` por parte del GC. Estos planes son excluyente porque una vez que uno trata el evento ya no se ejecuta el resto. De manera que si se cumple que el `Sender` es el robot y que lo que se solicita está dentro de los productos que el mercado suele entregar (líneas 18-19) se ejecutará el plan *Agree*. En este caso la AI `.member` permite verificar si un valor se encuentra dentro de una lista de valores y en la variable `N` se almacena el valor de la última orden procesada. Por otra parte, si el `Sender` no es el robot se ejecutará el plan *Refuse*. Si no existe información relacionada con el producto que se solicita se ejecutará el plan *Not understood* y finalmente si no se ejecuta ninguno de los planes anteriores se ejecutará el plan *Failure* para darle continuidad a la conversación. En todos estos planes básicamente se actualiza el estado de una creencia `orderStatus(Content,Status)` según el plan que se haya ejecutado y se invoca a la AI correspondiente.

Una vez se ha tomado una decisión sobre la solicitud recibida la conversación continúa y en caso de que se haya estado de acuerdo el GC añade la creencia *timetodotask(Content,Data,ConvID)* al agente. Esto trae como consecuencia la ejecución de uno de dos posibles planes: *Inform* o

*Failure* (líneas 56-74). Si existe la creencia *orderStatus(Content,agree)* indicando que la solicitud ha sido aceptada se ejecuta el plan *Inform* de lo contrario se ejecuta el plan *Failure*. En el primero se instancian las variables `Product` y `Qdt` con el producto y la cantidad a entregar. Esta información se extrae de los datos proporcionados en la solicitud del robot si el literal cumple con el formato entendido por el mercado (líneas 58 y 59) o 5 unidades de cerveza como valor por defecto (línea 60); se incrementa y actualiza el valor de la última orden procesada (líneas 61 y 62) y se intenta entregar el producto a través de la adición del subobjetivo `deliver(Product,Qtd)` (línea 63). Este subobjetivo provoca que se dispare el plan de las líneas 80-81 que solicita esta acción al entorno (apéndice A.2). En la línea 65 del plan se agrega una creencia con la información del resultado del procesamiento de la orden para esa conversación, seguidamente se imprime un mensaje en pantalla y se invoca a la AI correspondiente al paso *Inform* del protocolo, pasando como parámetro además el identificador de la solicitud y el resultado de la misma (línea 67-68).

Si al agregarse la creencia *timetodotask(Content,Data,ConvID)* se comprueba que no existe *orderStatus(Content,agree)* se considera que ejecución de la acción ha fallado y se dispara el plan de las líneas 72-74. En este plan se imprime un mensaje y invoca a AI correspondiente al paso *Failure*.

La salida en pantalla de los mensajes que se imprimen durante la ejecución del ejemplo pueden ser consultada en el apéndice A.3. Como se observa, los mensajes correspondientes a la conversación se imprimen cuando se agota el stock inicial de dos cervezas y cuando se agota el stock de 5 cervezas que se repuso tras la entrega del mercado; en ese punto el dueño ha consumido ya 7 cervezas. Llegadas las 10 cervezas límite el robot informa de que no le está permitido entregar más cerveza.

En general el número y tipo de los parámetros de las AI varían, ya sea entre los pasos del protocolo como entre los protocolos, pero tienen en común un identificador del paso como parámetro inicial y un identificador de la conversación como parámetro final. También como norma general se debe llamar a la AI correspondiente al protocolo siempre al final del plan en la que se encuentran, ya que esta llamada puede ocasionar posiblemente la ejecución de planes correspondientes a otros pasos de la conversación. Las estructuras que son realmente dependientes del dominio son las que se empleen en el contexto de los planes, que determinan qué plan se debe ejecutar cuando se tenga más de una alternativa para continuar la conversación y también las acciones que se deban realizar como parte de un razonamiento más complejo en algún paso específico de la conversación.

## 3.7. Conclusiones

En este capítulo se ha hecho una descripción detallada tanto de las herramientas que se han empleado para la propuesta como de la manera en que ha sido concebida e implementada. Se ha indicado cómo es posible, a través de un gestor de conversaciones, mantener varias conversaciones simultáneas por parte de un agente ya sea empleando el mismo protocolo o distintos protocolos. Se ha mostrado a través de un ejemplo, como de una manera sencilla se intenta abstraer al programador de las complejidades intrínsecas de las interacciones concurrentes manteniendo una comunicación indirecta. Como se ha observado, esta propuesta presenta numerosas estructuras que son estándares para cualquier dominio y que pueden ser definidas fácilmente de manera automática. Por esta razón se ha trabajado en una herramienta de modelado que permite definir, de manera visual estas interacciones aplicado a un entorno regulado de agentes conversacionales, que permite generar una plantilla para el código. Se comentarán los detalles de la herramienta en el capítulo 4.

# Capítulo 4

# Herramienta de modelado

## 4.1. Introducción

En la Ingeniería Dirigida por Modelos o Model Driven Engineering (MDE) por sus siglas en inglés, las aplicaciones son definidas por modelos lo que permite la generación automática de código haciendo uso de un nivel de abstracción mayor. Esto permite a los programadores aumentar su productividad si existen partes del código que se repiten constantemente en distintos momentos y esta es una tarea que puede ser ser automatizada. Tal es el caso de las interacciones entre agentes que se han abordado a lo largo de este trabajo: presentan estructuras repetitivas que abren una ventana para su posible automatización. En el presente capítulo se presenta una extensión al *Gestor de Conversaciones* propuesto en el capítulo 3 a través de una herramienta de modelado basada en *workflows* que permite representar interacciones y procesos en una manera gráfica amigable. La herramienta se implementa como un plug-in para el entorno de desarrollo integrado de Eclipse. Puede ser empleada en contextos donde varios agentes interactúan para intercambiar bienes o servicios en un entorno regulado. El empleo de este tipo de herramientas supone innumerables ventajas en el desarrollo de agentes, que van desde: hacer la tarea más fácil para los desarrolladores, eliminar las posibilidades de errores y ofrecer mayor modularidad hasta un mejor control y gestión de las interacciones.

## 4.2. Motivación

Dado que la implementación de protocolos de interacción es una tarea bastante apropiada a ser automatizada (ya que existen estructuras que pueden ser reutilizadas independientemente

del dominio) han surgido una serie de herramientas que permiten tanto el modelado gráfico de las interacciones como la generación del código de los agentes a partir de modelos [14, 15, 22].

En [22] se realiza una propuesta tipo "metodología" que toma la descripción de un protocolo y genera la descripción del comportamiento correspondiente de los agentes. Esta descripción se realiza a través de un lenguaje de modelado independiente de la plataforma para sistemas multi-agente denominado DSML4MAS (Platform Independent Modelling Language for MAS); a través de esta herramienta se puede obtener código tanto para agentes JACK [1] como para agentes Jade.

En cambio en [14] se propone la metodología "Hermes" para diseñar las interacciones entre agentes en términos de "objetivos de interacción". Se explica cómo los diseños obtenidos a partir de esta metodología se pueden implementar mapeando los artefactos de diseño a colecciones de planes. Estos planes pueden ser de los tipos:

- Coordinación: se derivan de los objetivos de interacción para coordinar los agentes en la interacción).
- Ejecución: se derivan directamente de las acciones y son pasos para completar un objetivo de interacció n.
- De interfaz: para transformar los mensajes entre agentes en objetivos y eventos para el procesamiento intra-agente.

La implementación en este caso requiere de plataformas de agentes cuyo comportamiento es definido a través de planes y objetivos como es el caso de JACK, Jadex, JAM [2], y Jason. Concretamente para este trabajo se ha seleccionado la plataforma Jadex.

Por otra parte en [15] se ofrece una herramienta gráfica para cubrir la brecha entre el análisis y el diseño de sistemas multi-agente integrando el diseño y la implementación de conversaciones de agentes a través de un plug-in de Eclipse [3] que genera código en Jade.

Existen además propuestas que se basan en la metáfora de flujos de trabajo (workflows) para contextos BPM (Business Process Management), donde un workflow representa un proceso de negocio. Tal es el caso de Wade (Workflows and Agents Development Environment) [12]. Esta plataforma está basada en Jade y ofrece formalismos basados en lenguaje Java. Además de definir agentes de acuerdo a workflows, esta ofrece una arquitectura, componentes adicionales

[1] http://www.agent-software.com.au/products/jack
[2] http://www.marcush.net/IRS/irs_downloads.html

y mecanismos que facilitan la administración de aplicaciones distribuidas basadas en Wade en términos de configuración, activación y monitorización. Wade permite coordinar los sistemas existentes y también se ofrece como marco de desarrollo de software para crear aplicaciones nuevas que impliquen la ejecución de tareas largas y complejas. Junto a Wade se ofrece un entorno de desarrollo denominado WOLF (WOrkflow LiFe cycle management environment) [13] que facilita la creación de aplicaciones basadas en Wade de manera gráfica a través de un plug-in de Eclipse.

En las propuestas analizadas se han realizados algunas observaciones. En algunos casos el enfoque es hacia la generación del código correspondiente a las interacciones independientemente del contexto en el que éstas se producen, pudiéndose obtener código para distintas plataformas pero considerándose únicamente las interacciones. En otros casos también se carece de mecanismos para la gestión de conversaciones a nivel de la lógica interna del sistema y también para la generación de código de agentes BDI. En otro caso no se ofrecen mecanismos para la especificación de sistema de manera amigable a través de algún entorno gráfico.

Atendiendo a lo anteriormente expuesto y adoptando la metáfora de workflows empleada en [12], en este trabajo se ha elaborado una herramienta de modelado que permite la especificación de manera gráfica de un sistema regulado por determinadas condiciones y donde participan un conjunto de roles. La herramienta permite además la especificación de interacciones, basadas en los protocolos de FIPA, que serán transformadas en código Jason de acuerdo a la estructura presentada en el capítulo 3.

## 4.3. Herramientas utilizadas

La herramienta de modelado que se ha diseñado ha sido concebida como un plug-in para Eclipse. Se ha escogido esta variante por ser un IDE (Integrated Development Environment) que proporciona gran flexibilidad y por su amplia aceptación en la comunidad de programadores. Además, al ser un IDE para desarrollo en Java, es perfectamente compatible tanto con la plataforma de sistemas multi-agente como con los agentes Jason que están también basados en Java. Eclipse permite la programación de agentes Jason a ser ejecutados sobre la plataforma Magentix2, de manera que es posible la creación de un modelo, la generación de código para los agentes y su reutilización en el proyecto para el sistema multi-agentes. Puede ser extendido modularmente por los denominados plug-ins, que proporcionan funcionalidad a otros plug-ins

a través de los puntos de extensión. Eclipse además proporciona una sólida base con el EMF [1] (Eclipse Modelling Framework) y el GMF [2] (Graphical Modelling Framework).

El EMF es un *framework* de modelado y generación de código que ofrece un conjunto de herramientas que soportan MDE de manera primaria en Java. Permite la definición de un modelo en XML Schema, UML o Java anotado para luego ser importados a EMF. El modelo obtenido es una representación de alto nivel que que establece la equivalencia entre estos formatos. Desde la especificación de este meta-modelo, EMF ofrece herramientas y soporte para obtener un conjunto de clases Java para el modelo, de clases para la visualización y edición y un editor básico. Este meta-modelo describe las características de los elementos que componen el modelo en sí a través de la definición de clases, atributos, métodos, relaciones de agregación y herencia, tipos de datos etc.

El GMF en cambio ofrece una serie de componentes generativos e infraestructuras para tiempo de ejecución para el desarrollo de editores gráficos basado en EMF y GEF [3] (Graphical Editing Framework).

Eclipse también ofrece herramientas para generar código ejecutable a partir de los modelos a través de M2T [4] (Model To text). Con M2T es posible generar artefactos textuales desde los modelos partiendo de la definición de un modelo concreto y del meta-modelo a partir del cual ha sido creado. Para ello es necesario principalmente crear un generador de código y una plantilla de transformación (haciendo uso del lenguaje Xpand para controlar la generación de salida).

## 4.4. Diseño de la herramienta de modelado

El modelo propuesto está diseñado básicamente para representar las condiciones y características de las acciones que puede realizar un agente en el sistema. Para conseguir esto se tiene en cuenta que existe un agente especial o "agente regulador" que mantendrá conversaciones con el resto de los agentes ofreciéndoles información del dominio siempre y cuando se satisfagan las condiciones necesarias. Las conversaciones que mantengan los agentes fuera de este contexto no se contemplan en esta versión de la propuesta, pero se pretende incluir utilidades en este sentido en versiones futuras. Estas decisiones de diseño han sido tomadas a partir

[1] http://www.eclipse.org/modeling/emf/
[2] http://www.eclipse.org/modeling/gmp/
[3] http://www.eclipse.org/gef/
[4] http://www.eclipse.org/modeling/m2t/

del modelo de negociación presentado en [5], que fundamenta su estructura en interacciones del tipo de las comentadas en el capítulo 3 de este trabajo.

### 4.4.1. Modelo de Negociación basado en interacciones

El modelo propuesto [5] ofrece estructuras para la negociación a ser empleadas en un entorno donde existen agentes que poseen determinados bienes o servicios, que están interesados en negociar con ellos y que juegan determinados roles en esta negociación. Los roles que se incluyen en esta propuesta pueden ser agrupados en *participantes* y *staff*. Dentro del primer grupo se encuentran: el rol *guest (g)* que representa cualquier agente interesado en entrar en el sistema, y una vez en el sistema el *guest* pasa a ser un *participante (p)*, que luego será instanciado en los roles *black (b)* o *wait (w)* en dependencia de la parte que representa en la negociación. En el segundo grupo se encuentran los roles *mediator (m)*, *Negociation Table Manager (ntm)* y *Legal Authority (la)* que se encargan de tareas reguladoras específicas, en este caso, la gestión de los protocolos de negociación de manera global, el control de las negociaciones de manera individual y las tareas asociadas a la adopción de los acuerdos respectivamente.

La especificación del modelo se realiza fundamentalmente a través de tres elementos: interacciones (proceso atómico o diálogo entre los agentes), *workflows* (modelos de interacción complejos y prescripciones procedurales) y transiciones (permiten la navegación de los participantes entre las interacciones). Con estos elementos se han definido las estructuras que componen el modelo que son las siguientes (figura 4.1):

- **Admission**: Interacción para realizar las acciones de verificación y registro de posibles participantes en una negociación. Los invitados ofrecen su información individual.
- **Negotiation Hall**: Grupo de interacciones que permiten a los participantes estar al tanto de las negociaciones activas, de las que han finalizado y de sus correspondientes acuerdos. En este *workflow* también es posible crear nuevas mesas de negociación y realizar tareas asociadas a acciones críticas.
- **Negotiation Table**: En este *workflow* los participantes ejecutan las negociaciones haciendo uso de protocolos de negociación estándar como por ejemplo subastas.
- **Agreement Enactment**: Interacción que permite la firma de los acuerdos y la gestión de quejas por parte de los participantes que pueden llegar a modificar estos acuerdos.

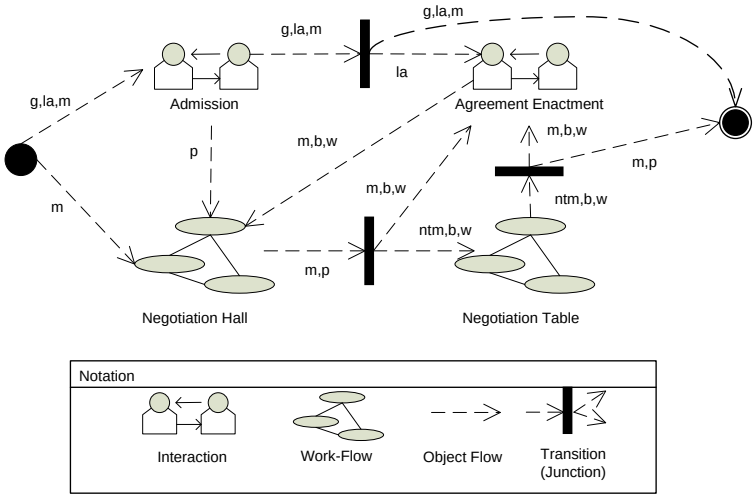


**Figura 4.1:** Modelo de Negociación basado en interacciones propuesto.

Este modelo propuesto ha sido aplicado en un caso de estudio de un mercado de agua. Este caso de estudio será utilizado más adelante para la aplicación de la propuesta.

### 4.4.2. Descripción del meta-modelo para la herramienta de modelado

Como se ha descrito anteriormente en la sección 4.4.1 existen determinados roles que regulan las acciones que van realizando los agentes en el sistema para poder entablar negociaciones y obtener beneficios de estas. Partiendo de esto, la herramienta de modelado propuesta puede ser empleada para contextos en el que se siga este tipo de modelo de negociación. Actualmente existe solo un "agente regulador" que auna los roles en el grupo *staff* del modelo de negociación, pero se pretende ampliar la propuesta para que existan agentes reguladores con responsabilidades más específicas.

Los elementos que conforman el modelo a generar se presentan con más detalle en [5]. Se puede tener una mejor comprensión de cuáles son estos elementos y sus características si se observa el meta-modelo (Ecore) de la figura 4.2 concebido para este propósito y obtenido a partir del EMF de Eclipse. Uno de los elementos principales es el representado por la clase *Action* que es una clase abstracta diseñada para representar las acciones del sistema. Esta clase posee una descripción y un identificador que permite hacer referencia a ella en todo el código. Las clases del meta-modelo consideradas como acciones (a través de una relación de herencia) son:

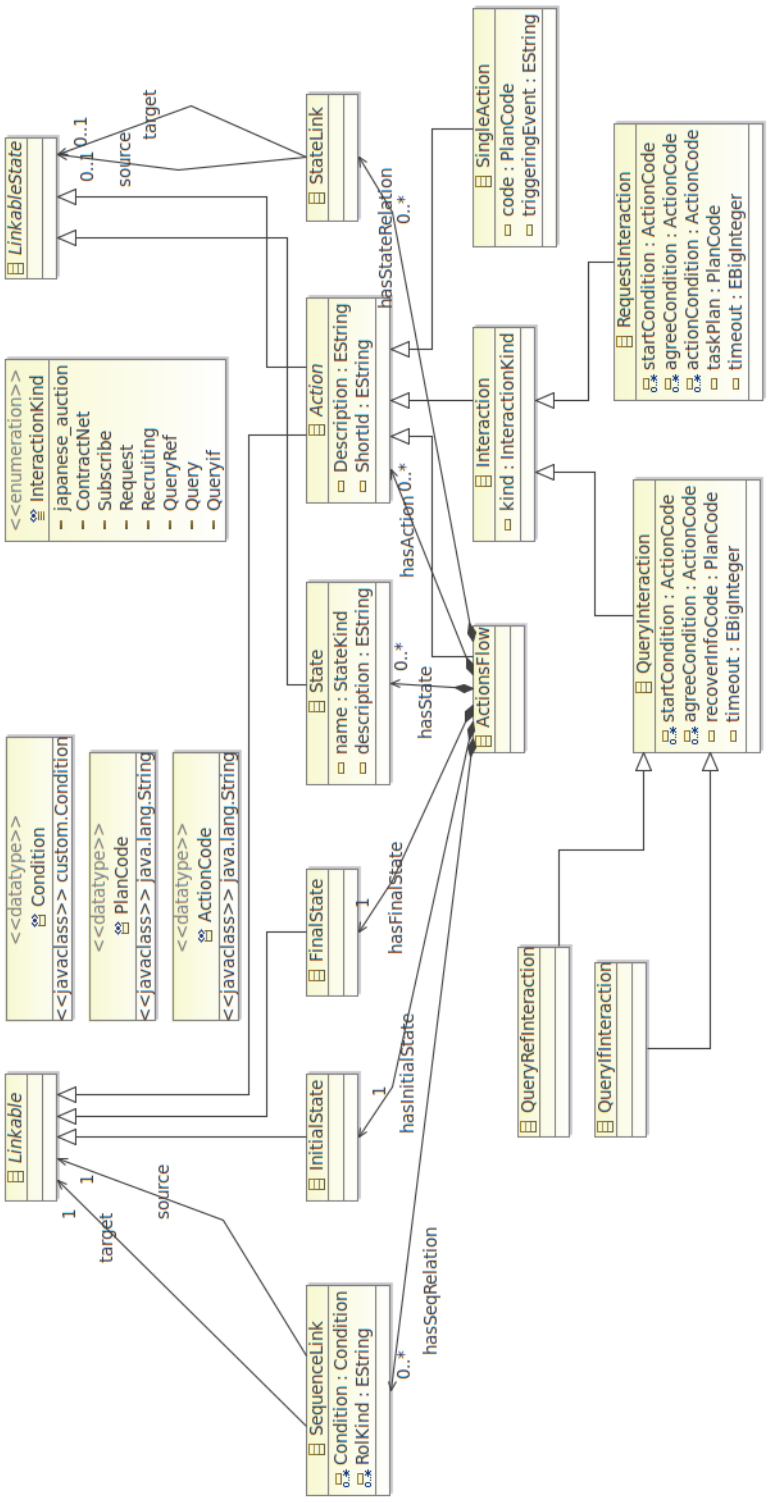


**Figura 4.2:** Ecore de la herramienta de modelado obtenido a través de EMF.

- *SingleAction*: Diseñada para especificar operaciones en un trozo código embebido en la definición del sistema. En la transformación a código este tipo de acción se convierte en un plan Jason, de manera que los atributos: *code* y *triggeringEvent* corresponden a las acciones del plan y a la descripción del evento disparador del plan respectivamente.

- *ActionsFlow*: Clase que representa un *workflow* en el sistema. A través de la relación *has-Action* se observa como un *ActionsFlow* puede contener, además de todos los elementos del diagrama, todo tipo de acciones incluyéndose a sí mismo (definición recursiva). De esta forma un *workflow* puede contener otros *workflows*. La inclusión de un *ActionsFlow* en el modelo implica la generación de un nuevo modelo.

- *Interaction*: Representa las interacciones presentes en el sistema. Es una clase abstracta de la que extienden todos los tipos de interacciones que puedan ser representadas en el modelo. El tipo de dato *InteractionKind* del meta-modelo es el empleado para determinar el tipo de la interacción a través del atributo *InteractionKind*.

Como se ha visto en el capítulo 3 los elementos que varían en la utilización de un protocolo para un propósito u otro están dados por: el contexto en el que se ejecuta cada paso del protocolo, que determina cuál es la alternativa a seleccionar para continuar la conversación y por el conjunto de acciones a ser ejecutadas como parte de un razonamiento. Por el momento la herramienta admite solo interacciones basadas en los protocolos Request, Query-If y Query-Ref de FIPA, pero se pretende extenderla para incluir todos los protocolos que soporte la plataforma, una vez comprobada la validez y utilidad de nuestro modelo. Las clases del meta-modelo que representan este tipo de interacciones y los atributos que permiten especificar estos elementos son:

- *RequestInteraction*: Interacción según el protocolo Request de FIPA. Esta posee los atributos: *startCondition*, *agreeCondition* y *actionCondition* para las condiciones de los pasos del protocolo *Begin*, *Agree* y *Action* respectivamente. También posee el atributo *taskPlan* para especificar el conjunto de acciones a ser realizadas en el paso *Action* y el atributo *timeout* para definir el tiempo de espera para las respuestas del otro agente.

- *QueryInteraction*: Interacción según el protocolo Query de FIPA. De esta clase extienden *QueryRefInteraction* y *QueryIfInteraction* que son los que realmente se utilizarán en la especificación del modelo. Posee los atributos para los pasos del protocolo *Begin* y

*Agree*, así como las acciones a realizar en el paso *Action* del protocolo para ofrecer la información solicitada, especificadas en el atributo *recoverInfoCode*. También se define en esta interacción el atributo *timeout* para esperar por las respuestas del otro agente.

Además de las clases para representar las posibles acciones que se puedan realizar el meta-modelo también comprende las clases:

- *InitialState*: Para definir el punto de partida por el que el agente entra al sistema.
- *FinalState*: Para definir el punto por el que el agente sale del sistema.
- *State*: Se asocia a las acciones. Puede representar el estado en el que tiene que estar el agente para ejecutar la acción o el estado resultante al que pasa el agente después de haber ejecutado la acción, dependiendo de si la relación está orientada hacia la acción (el estado es un requisito) o hacia el estado (el estado es una consecuencia).
- *StateLink*: Permite establecer la relación *State-Action* y *Action-State*.
- *SequenceLink*: Permite establecer la relación *Action-Action*, *InitialState-Action* y *Action-FinalState*. De esta forma establece una relación de precedencia entre las acciones. Este tipo de relación puede tener asociado roles y condiciones. Estos son los roles que debe tener el agente para ejecutar la acción que tiene la relación como consecuente y las condiciones que se deben cumplir. El tipo de dato *Condition* del meta-modelo es el empleado para estas condiciones.

## 4.5. Implementación y generación de código

Como se ha mencionado GMF ofrece una serie de componentes que permiten la generación de las estructuras necesarias para la herramienta de modelado partiendo de un conjunto de especificaciones realizadas a través del EMF y el GEF. Como resultado de su empleo y adaptación al propósito de la herramienta se ha obtenido un plug-in para eclipse que ofrece un entorno gráfico para el modelado. A través de una paleta de componentes es posible seleccionar el elemento a incluir en el modelo (ver figura 4.3). Una vez incluidos los elementos en el modelo, se pueden editar sus propiedades para especificar los valores que deben tener. Por ejemplo en la figura 4.4 se muestra un sencillo escenario en el que el “Agente regulador” es una especie de tablón informativo con productos en el que los agentes pueden consultar los

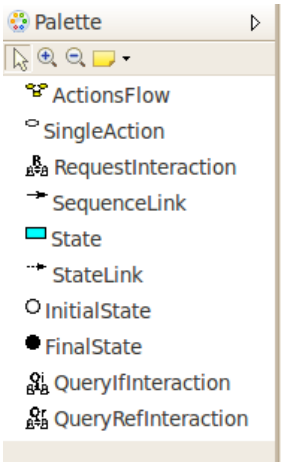


**Figura 4.3:** Paleta de componentes de la herramienta de modelado.

precios. El modelo consta de una *RequestInteraction* para responder a la solicitud del precio y una *SingleAction* para registrar al usuario como posible comprador. Además existe un estado al que pasa el agente de la *SingleAction* que es en este caso el de "Posible comprador". Para iniciar la interacción el agente *iniciador* debe tener el rol de *guest* así como para realizar su registro como posible comprador. Además para que la interacción tenga lugar es necesario que se cumplan dos condiciones: que en la solicitud del agente exista una variable instanciada con el objeto del que se desea saber el precio y que en el "Agente regulador" exista un registro del precio del objeto. Las propiedades de los elementos se pueden editar a través de la vista "Properties" ofrecida por Eclipse, y en caso necesario se crea una interfaz adecuada que permite su especificación de una manera más amigable. Por ejemplo en la figura 4.4 se muestra la edición de la propiedad "Task Plan" del *RequestInteraction*, donde se especifican las acciones a realizar en el paso *Action* del protocolo de interacción en el lado del *Participante*.

Partiendo de esto el siguiente paso sería la generación de las plantillas de código. En este sentido, para esta versión de la herramienta, el código que se genera corresponde al "agente regulador". El "agente regulador" actúa siempre como participante en las posibles conversaciones que los agentes de la plataforma deseen iniciar para obtener información del sistema a través de él. El programador solo tendrá que añadir el código correspondiente a los agentes como "iniciadores" en las interacciones. El algoritmo 1 es el algoritmo general para la generación de las plantillas de código.

Para ofrecer una mejor organización y modularidad del código, el algoritmo genera varios

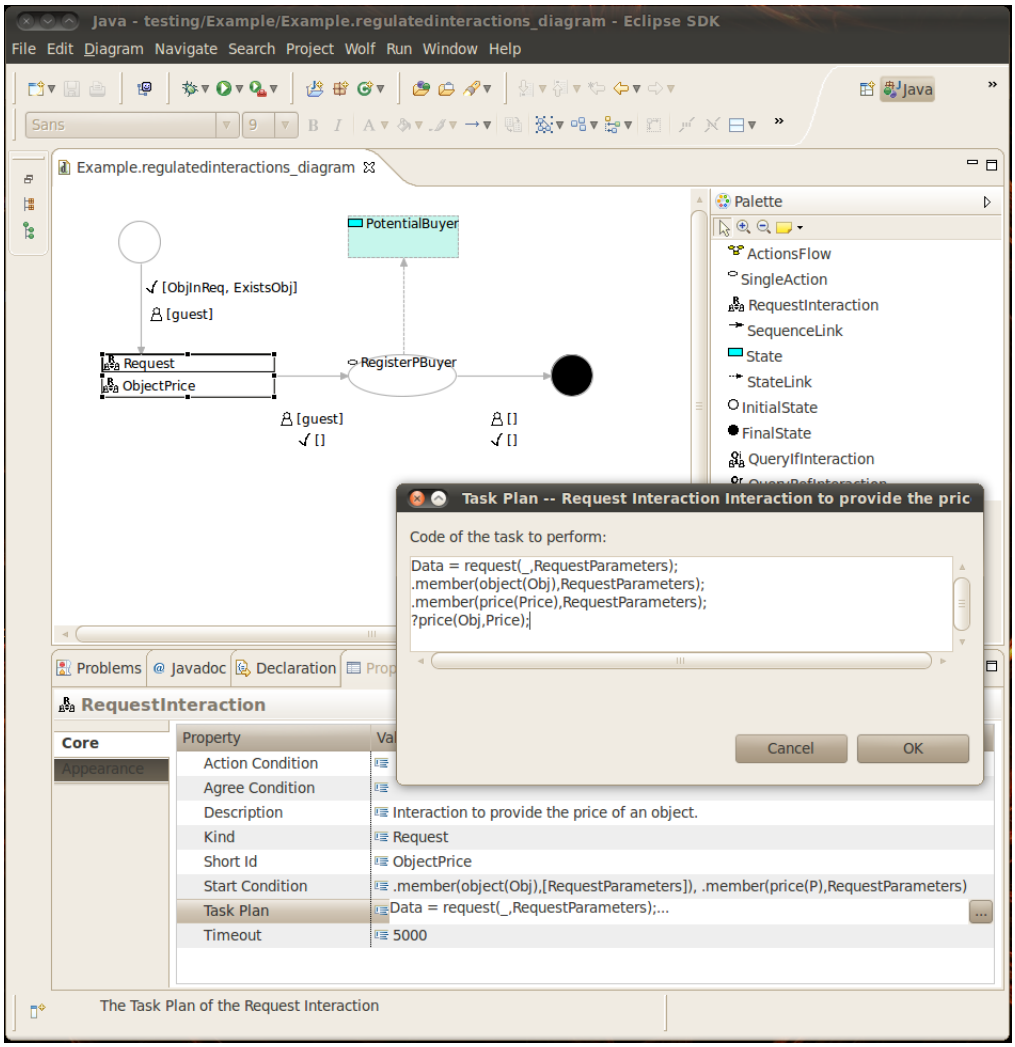


**Figura 4.4:** Diagrama ejemplo realizado con la herramienta de modelado.

**Algoritmo 1** Pasos para la generación de código a partir del modelo

**Entrada:** Modelo definido $M$

1: Crear creencias iniciales en fichero "beliefs_builder.asl"
2: **para todo** $ActionsFlow$ ($AF$) del modelo **hacer**
3: Crear un fichero independiente que lleve como nombre el identificador de $AF$
4: Incluir nombres de ficheros correspondientes a cada $ActionsFlow$ del modelo
5: **fin para**
6: **para todo** $SingleAction$ inmediatamente después del estado inicial **hacer**
7: Agregar un objetivo inicial
8: **fin para**
9: **para todo** $Interaction$ ($I$) **hacer**
10: **para todo** $StateLink$ que entra a $I$ **hacer**
11: Agregar esta condición en el plan inicial de la interacción $I$
12: **fin para**
13: Generar plan para crear identificador de conversación $ConvID$
14: Generar los planes necesarios para la $Interaction$
15: Incluir en plan inicial las condiciones y roles de los $SequenceLinks$ entrantes a $I$
16: **para todo** $StateLink$ que sale de $I$ **hacer**
17: Agregar creencia con el estado correspondiente para el agente en el último plan
18: **fin para**
19: **para todo** $SingleAction$ inmediatamente después de $I$ **hacer**
20: Agregar un objetivo en el último plan a ejecutarse
21: **fin para**
22: **fin para**

23: **para todo** $SingleAction$ ($SA$)del modelo $M$ **hacer**
24: Crear plan correspondiente
25: **para todo** $StateLink$ que entra a $SA$ **hacer**
26: Agregar esta condición en el contexto del plan para $SA$
27: **fin para**
28: Incluir para plan de $SA$ las condiciones y roles de los $SequenceLinks$ entrantes a $SA$

29: **para todo** $StateLink$ que sale de $SA$ **hacer**
30: Agregar creencia con el estado correspondiente para el agente
31: **fin para**
32: **para todo** $SingleAction$ inmediatamente después de $SA$ **hacer**
33: Agregar un objetivo en el último plan
34: **fin para**
35: **fin para**
36: **para todo** $ActionsFlow$ ($AF$) del modelo $M$ **hacer**
37: Crear objetivo y plan inicial para condiciones y roles de $SequenceLink$ entrantes
38: Repetir los pasos del 2 al 39 haciendo $M = AF$
39: **fin para**

ficheros con extensión ".asl"[1], aunque se pudo haber concebido también para incluirlo todo en un mismo fichero. Básicamente se crean tantos ficheros como *ActionsFlows* existan en el modelo y además un fichero para las creencias y reglas empleadas durante toda la ejecución de los agentes. En la línea 1 del algoritmo se realiza esta acción. A continuación, en el ciclo la línea 2 se crean los ficheros correspondientes a cada *ActionsFlow* y se generan los *include* correspondientes en el fichero actual. Luego, para cada *SingleAction* que esté inmediatamente después del paso inicial se debe especificar un objetivo a ser añadido al inicio de la ejecución (línea 6), de manera que los planes correspondientes se ejecutarán desde el comienzo.

De la línea 9 a la 22 se genera el código para cada interacción del modelo. Para ello se verifican aquellos *Estados* que el agente debe tener para poder interactuar (estados entrantes), y se pone esta condición en el plan inicial de la interacción (líneas 10-12); se genera un plan que, teniendo en cuenta varios parámetros, obtendrá un identificador único para la conversación (línea 13); se generan los planes necesarios de acuerdo al tipo de la interacción (línea 14). En

[1] ".asl" es la extensión que se emplea para los ficheros que contienen el código de los agentes Jason. El código de un agente puede estar distribuido en varios ficheros ".asl" siempre y cuando se incluyan unos dentro de otros mediante la directiva *include* de Jason. Para más información referirse a [9]

el contexto del primer plan generado se incluirán las condiciones y roles de los *SequenceLink* entrantes (línea 15). Para cada estado que sale de la interacción se actualizan el registro de las creencias correspondientes para el agente con el que se interactúa (líneas 16-18). Finalmente para cada *SingleAction* inmediatamente después de la interacción se agrega un objetivo que conlleva a la ejecución del plan correspondiente (líneas 19-21).

A continuación para cada *SingleAction* del modelo se crea el plan correspondiente (línea 24) y de igual forma se consideran: los estados entrantes (líneas 25-27) y las condiciones y roles de los *SequenceLink* entrantes (línea 28) para añadirlos al contexto del plan; así como los estados salientes (líneas 29-31) y las *SingleAction* inmediatamente después de esta para actualizar el estado del agente y añadir los objetivos correspondientes a las *SingleAction* siguientes respectivamente (líneas 32-34).

Finalmente para cada *ActionsFlow* del modelo se crea un plan donde se verifican las condiciones y roles de los *SequenceLink* entrantes al *ActionsFlow* y se añade un objetivo para la ejecución del plan (línea 37) y se repiten los mismos pasos que los realizados en el modelo padre (línea 38).

Todos los modelos deben tener un estado inicial y uno final, de otra forma pudieran quedar inconsistencias en el código generado. El razonamiento individual de los agentes iniciadores, así como la especificación de la contrapartida para las interacciones con el agente regulador deben ser proporcionadas y deben seguir la estructura requerida en dependencia de las interacciones a realizar. Además se debe emplear una ontología común que permita una comprensión común del dominio [1]. El código generado por la herramienta para el ejemplo de la figura 4.4 se puede consultar en el apéndice A.1 del trabajo.

## 4.6. Conclusiones

En este capítulo se ha presentado una primera versión para una herramienta de modelado que ayuda en la definición de los procesos que tienen lugar en un entorno con ciertas características. Uno de los principales aspectos que se consideran son los factores que hacen que transitar de un proceso a otro requiera de ciertas comprobaciones. Se toman en cuenta en este sentido principalmente los roles que debe jugar el agente, los estados en los que se debe

[1] Hasta el momento no es posible incluir una ontología en el modelo pero es una tarea prioritaria para futuras versiones.

encontrar y las condiciones que se deben cumplir para realizar una acción u otra. Con estos elementos la herramienta permite la generación automática de plantillas de código para el "Agente regulador" encargado de las comprobaciones del sistema y con el cual los agentes iniciarán conversaciones.

# Capítulo 5

# Aplicación a un caso de estudio: mWater

## 5.1. Introducción

En este capítulo se comentarán los detalles tanto del empleo de la herramienta de modelado como del uso del "Gestor de conversaciones" en un prototipo para un caso de estudio enmarcado en un mercado de agua. El caso de estudio "mWater" presenta las características requeridas para la aplicación de la propuesta, al existir un conjunto de entidades o personas con el interés de negociar sus derechos de agua y de llegar a acuerdos, propiciando así un uso más eficiente de los recursos hídricos. Se presentará tanto el diseño realizado como las estructuras empleadas en la implementación.

## 5.2. mWater

### 5.2.1. Descripción del Caso de Estudio

En este caso de estudio se aborda una problemática de interés en numerosos países hoy día: la escasez de agua y el aprovechamiento de los recursos hídricos. Actualmente existe poco balance entre los tipos de uso del agua (por ejemplo el consumo humano, uso en la industria, la navegación etc), lo que conduce a una necesidad urgente de mejorar este proceso . Para ello es necesario considerar aspectos económicos, de medio ambiente y sociales determinados por condiciones físicas como las lluvias, distribución de la población, uso de la tierra, actividades económicas y además, principalmente por la demanda de agua. Una forma de medir el éxito

de una política de gestión de agua es observando su uso actual. Un diseñador de políticas en este sentido tiene como objetivo crear leyes de agua apropiadas para regular las acciones de los usuarios y darles la posibilidad de intercambiar recursos.

Un mercado de agua se comporta como un mercado tradicional de bienes en el que los derechos de agua deben intercambiarse según bases del día a día, cumpliéndose con determinadas normas preestablecidas y a cambio de alguna compensación. Estos mercados permiten dar respuesta a cambios rápidos en la oferta y la demanda de manera que pueden contribuir a un aumento de la inversión y empleo partiendo de que los usuarios poseen acceso asegurado a suministros de agua. En estas negociaciones puede que se desee fomentar salidas que se asocien a expectativas de precio, y además a otros factores como la garantía de bienes públicos en un entorno saludable, el equilibrio entre distintas partes interesadas como granjeros, municipios, la empresa energética etc. [10]

Partiendo de esto se puede intuir que el diseño de políticas en este sentido es una tarea difícil. Surge la necesidad entonces de que los diseñadores de políticas cuenten con medios y metodologías que les permitan predecir y visualizar las consecuencias potenciales de la inclusión de nuevas regulaciones y de ajustarlas antes de su aplicación en un contexto real para evitar comportamientos no deseados. Experiencias internacionales en California (USA), Chile, Australia y México has demostrado que estos mercados de agua mejoran la eficiencia económica del uso del agua y estimulan la inversión [21, 23, 29, 30].

Como se ha mencionado en el capítulo 4, este caso de estudio ha sido empleado para la aplicación del Modelo de Negociación (MN) para una infraestructura basada en Sistemas Multiagente que proponemos en [5]. En este capítulo se explica en más detalle esta implementación haciendo uso de las herramientas y técnicas descritas a lo largo del trabajo.

### 5.2.2. Diseño realizado

Las estructuras empleadas para el diseño del prototipo se resumen en un conjunto de roles e interacciones descritas en la sección 4.4.1. Por simplicidad del diseño se ha considerado agrupar los roles en *buyer* y *seller* (correspondientes a *black* y *white* en el grupo de roles participantes) y *staff* (grupo de roles staff), de manera que, el modelo obtenido a través de la herramienta de modelado generará el código para el rol *staff*, mientras que el código para el razonamiento de los agentes *buyer* y *seller* debe ser ofrecido por las entidades participantes. No obstante se han incluido las estructuras necesarias para interactuar con el agente "staff" y solicitarle información en un conjunto de plantillas a ser empleadas por estas entidades, de manera

que solo sea necesario implementar los planes en los que se toman decisiones dependientes del dominio. Por otra parte en el prototipo se incluyen las siguientes acciones agrupadas por la estructura del MN a la que pertenecen:

- **Admission**

    1. *Acreditación*: Corresponde a la interacción *Admission* del MN. A través de esta acción los usuarios podrán entrar al mercado.

- **Negociation Hall**

    1. *Mesas de negociación*: Acción para saber cuáles son las mesas de negociación actualmente abiertas en el mercado y a cuáles de ellas el agente ha sido invitado.
    2. *Unirse a mesa de negociación*: Permite al agente solicitar entrar a una mesa de negociación a negociar.
    3. *Crear nueva mesa de negociación*: Permite al agente solicitar crear una nueva mesa de negociación.
    4. *Invitar participantes*: Para invitar a posible participantes de la mesa de negociación.

- **Negociation Table**

    1. *Negociar*: Para establecer negociaciones con otros agentes (subasta japonesa en este caso), pero nuevos protocolos pueden ser fácilmente incorporados.

- **Agreement Enactment**

    1. *Registrar Acuerdo de Transferencia*: Para registrar los datos del acuerdo de negociación así como de las participaciones si éste ha sido exitoso.

Estas acciones resumen a gran escala lo necesario en el mercado de agua. Existen otras acciones que también forman parte de este tipo de mercados como es el caso de las validaciones de los acuerdos para verificar si los acuerdos están acordes con las convenciones normativas del plan hidrológico y de los procesos de quejas en los que se pueden generar modificaciones a los acuerdos ya realizados, pero se salen del alcance de este trabajo. No obstante este prototipo sienta las bases para un sistema con mayores prestaciones y funcionalidades.

### 5.2.3. Modelo para mWater

Partiendo de las funcionalidades comentadas en la sección 5.2.2 se ha procedido, mediante la herramienta de modelado, a la creación del modelo que permitirá obtener las plantillas de código para el agente *staff* en este caso. El modelo realizado se muestra en la figura 5.1. El primer proceso una vez es iniciado el prototipo es la ejecución de una *SingleAction* denominada *createConfiguration*; en esta acción se crea una nueva configuración. Esta configuración permite recuperar, para una ejecución realizada con unos parámetros determinados, todas las operaciones realizadas y resultados obtenidos. Estos parámetros son un requisito necesario para crear la configuración que estará asociada a todos los procesos a realizar con lo que se especifica en el *SequenceLink* desde el estado inicial a *createConfiguration*. Seguidamente el agente puede acreditarse ya sea como *buyer* o como *seller*. Para ello iniciará una interacción con el *staff* del tipo Request en la que éste realizará una serie de comprobaciones en los pasos correspondientes del protocolo como *participante*. Estas comprobaciones se especifican en las propiedades de la interacción. Como resultado de esta interacción el agente que solicita acreditarse pasara a los estados “accredited” y “inTradingHall” también representados en la figura. Como se observa, este último estado es un requerimiento para iniciar la interacción para obtener la lista de mesas de negociación activas (*getOpenNTTList*), para unirse a una mesa de negociación (*joinNTT*) y para crear una nueva mesa de negociación (*createNTT*). En todas las interacciones el agente que inicia puede participar tanto como *buyer* o como *seller*. Un vez el agente conoce las mesas de negociación abiertas puede decidir unirse a alguna a la que haya sido invitado o crear una nueva. Si se une a alguna mesa pasará a tener el estado *InTable*. Posteriormente el *staff* pasará a verificar si están dadas las condiciones para iniciar una negociación (*VerifyAuctionCondition*); esta acción se ejecutará cada vez que se cree una nueva mesa o se una un participante a un mesa existente realizándose las acciones necesarias para notificar al dueño de la mesa que la negociación debe comenzar. Una vez finalizada una negociación el *seller* debe informar al *staff* tanto de los datos asociados al acuerdo y al derecho de agua como de las participaciones. Las acciones para registrar los datos del acuerdo y de las participaciones se realizan mediante las *SingleAction registerTransferAgreement* y *registerParticipations* respectivamente.

Como se observa en la figura 5.1 la acción *createNTT* es de tipo *ActionsFlow*. Esto significa que esta acción contiene una secuencia de acciones similar a las que presenta el modelo en la que está contenida y que estas acciones se definen a su vez en un nuevo modelo (ver sec-

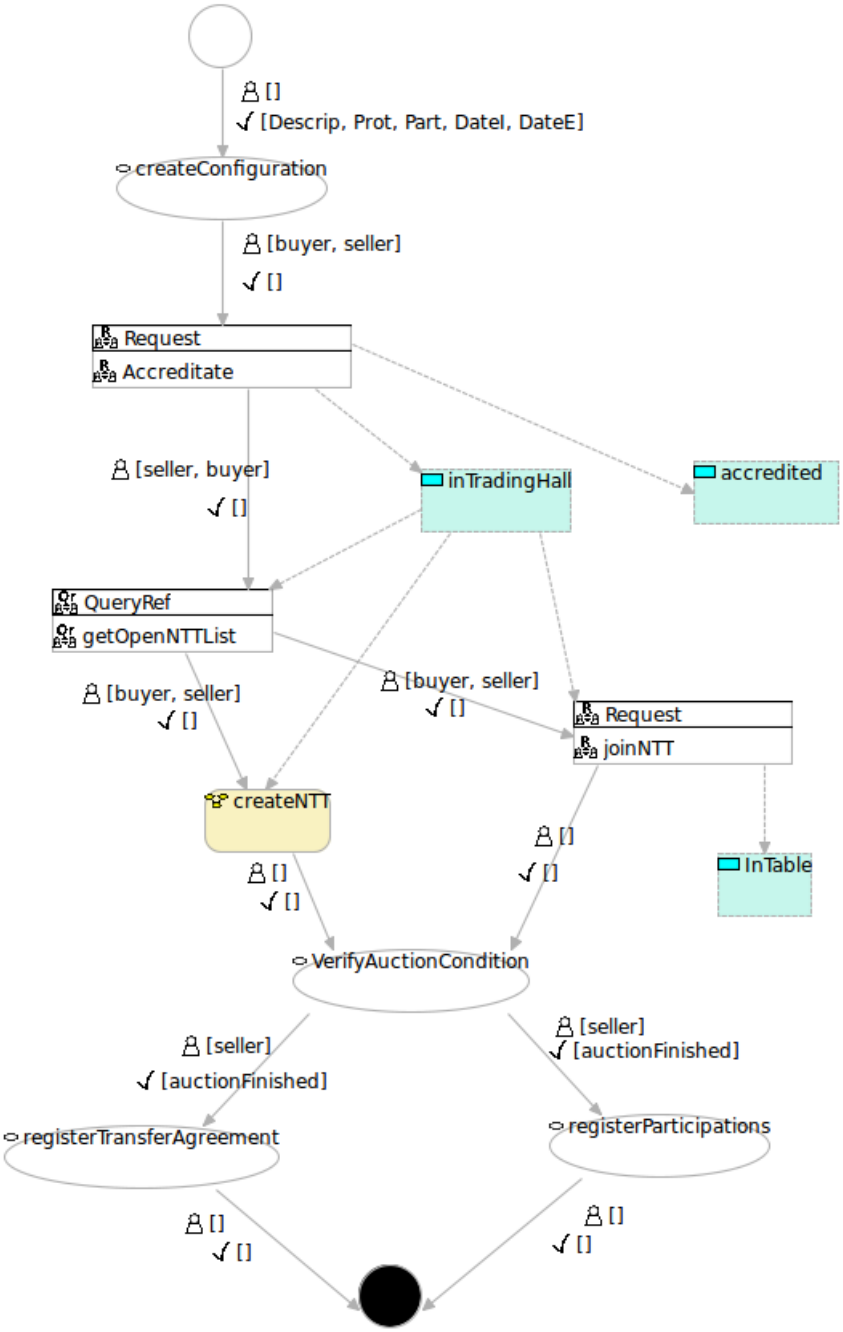


**Figura 5.1:** Modelo para la generación de código para el prototipo mWater.

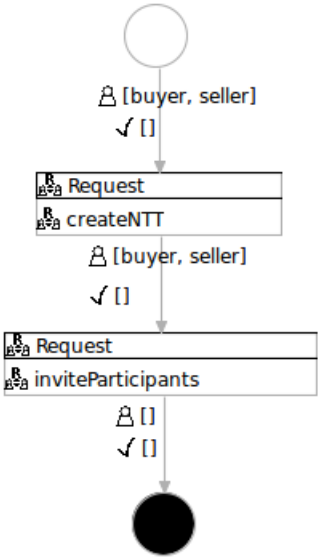


**Figura 5.2:** Modelo para el *ActionsFlow*: *createNTT* que permite crear una nueva mesa de negociación.

ción 4.4.2). El modelo correspondiente se muestra en la figura 5.2. En este modelo se incluyen básicamente dos interacciones: una para crear la mesa de negociación y otra para invitar a lo participantes especificados.

En el modelo no se contemplan las posibles negociaciones que puedan llevar a cabo los agentes porque esto debe estar implementado en razonamiento particular de cada agente. No obstante se ofrecen plantillas que pueden ser empleadas y adaptadas con este propósito.

### 5.2.4. Implementación

Este caso de estudio ha sido desarrollado en conjunto en un equipo de trabajo con responsabilidades distintas para cada miembro. Otros miembros realizaron el diseño y especificación de la base da datos a través de gestor de base de datos MySQL [1]. Por otra parte ha sido desarrollada, también por otros miembros del equipo, una aplicación que permite, a partir de los datos en la base de datos, obtener estadísticas y mostrarlas de gráficamente de una manera amigable dada una configuración particular del sistema [5]. Como parte del prototipo ha sido necesario la creación de una interfaz, a través de estructuras proporcionadas por Jason, que permitiera leer y almacenar las creencias del agente en la base datos.

[1] http://www.mysql.com/

El prototipo también cuenta con una GUI (Graphical User Interface) que permite la interacción humano-agente, de tal manera que un humano puede interactuar con otros humanos o agentes automáticos en tiempo de ejecución, realizando cambios en el sistema e incluso negociaciones. Esta interfaz permite estudiar distintas situaciones y comportamientos para ver su influencia en la evolución del mercado. Para este propósito se creó un sitio Web con PHP como lenguaje de interpretación para facilitar la comunicación entre la Web y el sistema multi-agente. Además se empleó una aplicación interfaz, también creada en el grupo de trabajo, que facilita el paso de información de la Web al sistema multi-agente y viceversa. En este sentido el humano tendrá una representación en el sistema a través de un agente que, conoce las estructuras para interactuar y realizar las acciones necesarias pero no razona sobre ningún proceso ya que las decisiones las toma el humano y se las trasmite. La figura 5.3 muestra una página del sitio en la que el humano participa en una negociación bajo una Subasta Japonesa con dos agentes automáticos del sistema.

Por último el código para los agentes ha sido extendido a partir de las plantillas generadas por la herramienta de modelado, y también se ha implementado el comportamiento de los agentes participantes tanto en su rol de *seller* como *buyer*.

## 5.3. Conclusiones

En el presente capítulo se ha presentado un prototipo que ha servido como base de pruebas para el empleo de la herramienta de modelado y de las técnicas para las conversaciones propuestas en los capítulos 3 y 4. El prototipo es una base para el sistema multi-agente que asistirá a los diseñadores de políticas en el contexto de mercados de agua, ofreciendo soporte para el control del comportamiento del mercado y la toma de decisiones. Tras el desarrollo del sistema multi-agente a través de estas herramientas se han podido ejecutar múltiples conversaciones concurrentes sobre la plataforma de manera satisfactoria y se ha comprobado cuánto se ahorra en la implementación, en términos de esfuerzo y tiempo de desarrollo. En este sentido el programador solo debe enfocarse en la estructura a nivel global del sistema y en el razonamiento a realizar por parte de los agentes en puntos concretos del código, abstrayéndose de factores como tiempos de espera, control de conversaciones concurrentes, de excepciones asociadas a las interacciones, etc. La herramienta de modelado además permite ofrecer modularidad, organización y facilita las comprobaciones necesarias a nivel de código.

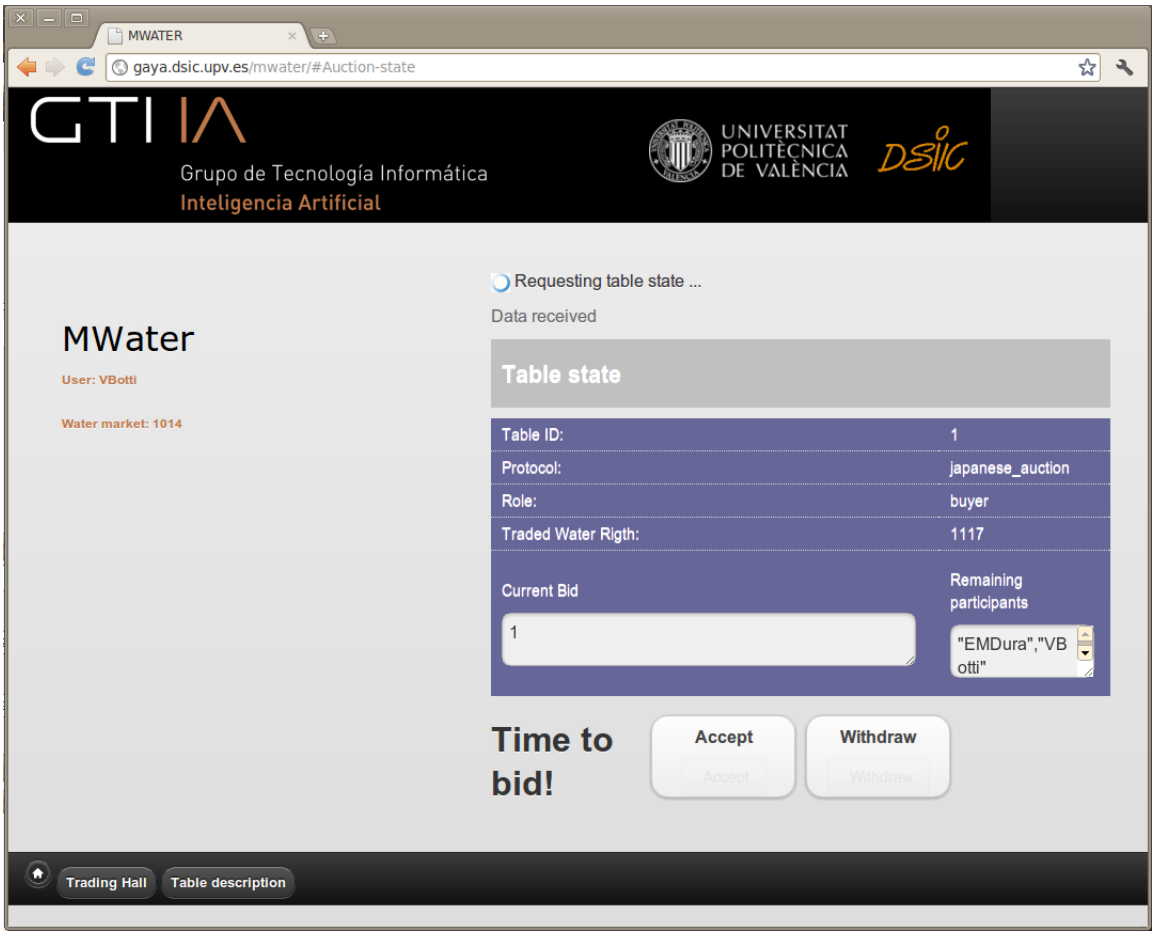


**Figura 5.3:** GUI para la interacción humano-agente. El usuario participa en una subasta japonesa con otros agentes y/o humanos.

# Capítulo 6

# Conclusiones y trabajo futuro

## 6.1. Conclusiones generales

Los sistemas multi-agentes prometen ser capaces de hacer frente a diversos dominios de aplicaciones complejas, pero la realidad es que estos en sí mismos tienden a ser complejos. Se han dedicado esfuerzos de investigación a distintos niveles intentando cubrir necesidades tanto a nivel de plataformas, de los agentes de manera individual como de metodologías y herramientas de diseño. Para el caso particular de las interacciones entre agentes han sido elaboradas distintas propuestas que ofrecen mecanismos para soportarlas y facilitarlas, aunque generalmente se suelen cubrir aspectos de manera parcial, siendo necesario a menudo la búsqueda de varias tecnologías para que, su conjunto, permitan obtener una solución. El trabajo aquí presentado se relaciona con varios de estos niveles intentando cubrir la brecha que pueda existir entre ellos y ofreciendo un conjunto de herramientas que permiten implementar las conversaciones entre agentes de una manera más sencilla, organizada, robusta y regulada.

En primer lugar se ha creado un gestor de conversaciones que permite a los programadores emplear protocolos de interacción para la comunicación entre agentes, desde un modelo BDI (agentes Jason) y sobre una plataforma con soporte para protocolos de interacción (Magentix2). La posibilidad de crear acciones internas ofrecida por Jason ha facilitado el intercambio de información asociada a la conversación entre el gestor de conversaciones y el agente. Para ello es necesario definir un conjunto de planes e invocar a determinadas acciones internas en esos planes dependiendo del protocolo de interacción que se esté empleando y de cada paso en particular del protocolo. Pudiera pensarse que el empleo del gestor de conversaciones, lejos de hacer el código más simple, hace que éste se haga más extenso dado por la necesidad de

incorporar al código determinadas estructuras para poder hacer uso del mismo. Esto quizás sea cierto si se trata con ejemplos muy sencillos en los que prácticamente no existan conversaciones concurrentes y se deban realizar pocas verificaciones; sin embargo, si por el contrario nos enfrentamos a sistemas más complejos, en los que se ejecuten numerosas conversaciones al mismo tiempo y se requiera un mayor número de verificaciones, este tipo de estructuras facilita el trabajo.

Los planes para hacer uso del gestor de conversaciones presentan numerosas estructuras que pueden ser reutilizadas independientemente del dominio, lo que posibilita que su especificación pueda ser una tarea fácilmente automatizada. Esto unido al hecho de que las interacciones y posibles acciones que pueda realizar un agente suelen ocurrir siguiendo un determinado flujo de trabajo en el que intervienen un conjunto de condiciones, ha motivado a la creación de una herramienta de modelado basado en *workflows* para la generación de código. El código generado es compatible con el gestor de conversaciones también propuesto en este trabajo. En el modelo pueden ser especificadas las condiciones, roles o estados que debe tener un agente para realizar una acción o interacción así como los estados a los que puede pasar. También se pueden especificar, ya a nivel de interacción, las condiciones para pasar de un estado a otro cuando se está teniendo la conversación, así como las acciones a realizar en los pasos donde sea necesaria su especificación, que son en definitiva los aspectos que pueden cambiar de un dominio a otro.

Tras el desarrollo del prototipo para el caso de estudio “mWater” haciendo uso de la herramienta de modelado, se ha experimentado un ahorro en términos de esfuerzo y tiempo de desarrollo en primer lugar si se compara con el esfuerzo que se ha requerido en la implementación de este mismo caso de estudio haciendo uso de otras herramientas, y en segundo lugar por las ventajas que ofrece una herramienta visual con una interfaz amigable que ofrece generación de plantillas de código donde solo es necesario realizar pequeñas modificaciones. Por otra parte, la herramienta de modelado además permite ofrecer modularidad, organización y algunas comprobaciones necesarias a nivel de código. Además con el uso del gestor de conversaciones la atención se debe enfocar solo a la estructura a nivel global del sistema y al razonamiento a realizar por parte de los agentes en puntos concretos del código, de donde los factores como tiempos de espera, control de conversaciones concurrentes, de excepciones asociadas a las interacciones, etc, se dejan a cargo de la plataforma.

## 6.2. Trabajo futuro

Los protocolos de interacción predefinidos son suficientes para algunos dominios y problemas, pero a veces se requiere de mayor flexibilidad ya que los sistemas multi-agentes presentan una naturaleza inherentemente dinámica. En este sentido la plataforma Magentix2 soporta la modificación de los protocolos de interacción de manera dinámica pero se necesita extender la propuesta realizada en este trabajo para que los agentes Jason puedan hacer uso de esta funcionalidad y poder crear conversaciones más abiertas. Por otra parte, también ofrecería mayor flexibilidad el poder modificar o definir nuevas interacciones desde cero, con lo que, incorporar esta posibilidad en la herramienta de modelado es también una tarea prevista.

Con el empleo de la herramienta de modelado a su vez, como ha sido comentado, se asume que los agentes que iniciarán las interacciones, presentan la estructura requerida para poder conversar con el "agente regulador". Sería útil que la herramienta de modelado pudiera generar plantillas para ser empleadas por estos agentes también, lo que se convierte, de igual forma en un propósito a corto plazo. Así mismo se prevé analizar cómo es el comportamiento del sistema si, en lugar de tener un único "agente regulador", se pudiera desglosar en varios roles con funciones más especificas de acuerdo a lo propuesto en [5].

Como trabajo para extender la herramienta de modelado, y dotarla de un mayor número de funcionalidades y robustez también se prevé agregarle la posibilidad de emplear el resto de los protocolos definidos en las especificaciones de FIPA y de emplear ontologías en las que se definan los términos específicos de cada dominio.

# 6. CONCLUSIONES Y TRABAJO FUTURO

# Apéndices

# Apéndice A

# Código

## A.1. Código generado por la herramienta de modelado para el modelo ejemplo 4.4

A continuación se muestra el código generado para el "Agente regulador" y además el fichero para las creencias de soporte "beliefs_builder.asl" a emplear durante la ejecución.

**regulatorAgent.asl**

```
{include ("beliefs_builder.asl")}

/* Initial objectives */



/* Plans */

+?objectPriceConvID(Sender,[Parameters],ConvID)
<-      //Getting the list of ground values of the literals in the
    parameters
        !listFromOneArgLiterals([Parameters],ValuesList);
        .date(Y,Mo,D);
        .time(H,Mi,Seg);
        !concatListElements(ValuesList,ValuesString);
        .concat(ValuesString,".",Sender,".",Y,".",Mo,".",D,".",H,"."
            ,Mi,".",Seg,ID);
        ConvID = conversation(objectPrice,Sender,ID).

```

```
18 /* ------------------------------------------------------------ */
19
20 /* -> FIPA REQUEST objectPrice - */
21 /*      Interaction to provide the price of an object.  */
22
23 @join_request_fail_objectPrice
24 +?join(ConvID,frp,request("objectPrice","Interaction to provide the
       price of an object.",[RequestParameters]),AgentName,Rol):
25         objectPriceConv(ConvID,[RequestParameters])
26         &(not .member(Rol,[guest])
27         |(not .member(object(Obj),RequestParameters))
28         |(not price(Obj,_)))
29 <-
30         .fail.
31
32 @join_request_objectPrice
33 +?join(ConvID,frp,request("objectPrice","Interaction to provide the
       price of an object.",[RequestParameters]),AgentName,Rol):
34         ConvID = conversation(objectPrice,ID)&
35         objectPriceTimeOut(TO)&.member(object(Obj),[
              RequestParameters])
36         &.member(price(P),RequestParameters)
37         &(.member(Rol,[guest])
38         |(.member(object(Obj),RequestParameters))
39         |(price(Obj,_)))
40 <-
41         +objectPriceConv(ConvID,[RequestParameters]);
42         .ia_fipa_request_Participant("joinconversation",TO,ConvID).
43
44
45 @not_understood_objectPrice
46 +request(Sender,Content,Data,ConvID):
47         strName(Sender,StrSender)&
48         objectPriceConv(ConvID,[RequestParameters])&
49         possibleactions(L)& not .member(Content,L)&Content=
              objectPrice
50 <-
51         .print("- I've received a request for doing: objectPrice but
               I don't understand. It is not inside the actions I can
              do.");
52         -+taskStatus(Content,notunderstood,StrSender,ConvID);
```

```
53          .ia_fipa_request_Participant("notunderstood",ConvID).
54
55  @refuse_objectPrice
56  +request(Sender,Content,Data,ConvID):
57          strName(Sender,StrSender)&
58          objectPriceConv(ConvID,[RequestParameters])&
59          Data=request("objectPrice",[RequestParameters])&
60          Content=objectPrice
61  <-
62          .print("- I've received a request for doing: objectPrice but
                I refuse.");
63          -taskStatus(Content,_,StrSender,ConvID);
64          +taskStatus(Content,refuse,StrSender,ConvID);
65          .ia_fipa_request_Participant("refuse",ConvID).
66
67  @agree_objectPrice
68  +request(Sender,Content,Data,ConvID):
69          strName(Sender,StrSender)&
70          objectPriceConv(ConvID,[RequestParameters])&
71          Data=request("objectPrice",[RequestParameters])&
72          Content=objectPrice
73
74  <-
75          //.print("- I've received a request for doing: objectPrice
                and I'm agree.");
76          -taskStatus(Content,_,StrSender,ConvID);
77          +taskStatus(Content,agree,StrSender,ConvID);
78          .ia_fipa_request_Participant("agree",ConvID).
79
80  @do_task_inform_objectPrice
81  +timetodotask(Content,Data,ConvID):
82          objectPriceConv(ConvID,[RequestParameters])&
83          Content=objectPrice&
84          taskStatus(Content,agree,StrSender,ConvID)
85  <-
86          !doTask(Content,StrSender,Data,ConvID);
87          ?taskResult(Content,ConvID,StrSender,Result);
88          -taskStatus(Content,agree,StrSender,ConvID);
89          .ia_fipa_request_Participant("inform",Content,Result,ConvID)
                .
90
```

```
91
92  @do_task_objectPrice
93  +!doTask(Content,Data,ConvID):
94          objectPriceConv(ConvID,[RequestParameters])&
95          Content=objectPrice&
96          Data=request(objectPrice,[RequestParameters])&
97          taskStatus(Content,agree,StrSender,ConvID)
98          &.member(object(Obj),[RequestParameters])
99          &.member(price(P),RequestParameters)
100 <-
101         //{User code. The result must be stored in a 'Result'
                variable.}
102
103         Data = request(_,RequestParameters);
104         .member(object(Obj),RequestParameters);
105         .member(price(Price),RequestParameters);
106         ?price(Obj,Price);
107         Result = price(Obj,Price).
108
109         +taskResult(Content,ConvID,StrSender,Result).
110
111 @do_task_failure_objectPrice
112 -!doTask(Content,Data,ConvID):
113         objectPriceConv(ConvID,[RequestParameters])&
114         Content=objectPrice
115 <-
116         .ia_fipa_request_Participant("failure",Content,ConvID).
117
118         @conversation_ended_objectPrice
119         +conversationended(ConvID,Result):
120         objectPriceConv(ConvID,[RequestParameters])&
121         Content=objectPrice&
122         taskResult(Content,ConvID,Sender,Result)
123 <-
124         -taskResult(Content,ConvID,Sender,Result);
125         registerPossibleBuyer(AgentName).
126
127
128         /* - FIPA REQUEST objectPrice <- */
129
130
```

```
@single_action_RegisterPBuyer
+!registerPossibleBuyer(AgentName):
        (.member(Rol,[guest]))
<-
        if (possibleBuyers(PB))
        {.concat(PB,AgentName,NewPB);
        -possibleBuyers(PB);
        +possibleBuyers(NewPB);}
        else{ +possibleBuyers([AgentName]); }
                .abolish(state(Sender,_));
                +state(Sender,PotentialBuyer).
```

**beliefs_builder.asl**

```
/* GENERAL RULES */

strName(Name,StrName):- .string(Name)&StrName=Name.
strName(Name,StrName):- not .string(Name)&.concat("",Name,StrName).
possibleactions(objectPrice).

/* GENERAL PLANS */

//Plan for getting a list of the terms given a list of one argument
   literals
+!listFromOneArgLiterals([Lit|LiteralsListTail],Result)
<-      Lit=..[A,[B],C];
        if (not .ground(B)){Elem=[];}else{Elem=B;}
        !concatElems(Elem,LiteralsListTail,Result).
        +!listFromOneArgLiterals([],[]).
        +!concatElems(Elem,Tail,Out)
<-
        !listFromOneArgLiterals(Tail,Result);
        if (Elem\==[]){.concat([Elem],Result,Out);}else{.concat(Elem
           ,Result,Out);}.
        +!concatElems([],"").

//Plan for getting a string by concatenating the elements of a list
+!concatListElements(L,Out)
<-      .reverse(L,List);
        !concatListElem(List,Out).

```

```
+!concatListElem([Head|Tail],Out)
<-      !concatListElem(Tail,Result);
        if (.ground(Head)){.concat(Result,Head,Out);}else{.concat(
            Result,"",Out);}.
        +!concatListElem([],"").
```

## A.2. Código para el agente simulador del entorno del acápite 3.6

A continuación se muestra el código correspondiente al agente simula al entorno en el ejemplo de la sección 3.6. Éste posee una serie de planes para dar respuesta a solicitudes del robot, el dueño y el mercado. En los comentarios se describe la funcionalidad de cada uno.

**environmentAg.asl**

```
/* Creencias iniciales */

//Estado del dueño
//----------------
//Cantidad de sorbos inicial. Como no tiene cerveza inicialmente
    este valor es cero.
sipCount(0).
//Para controlar si el robot está llevando una cerveza consigo
carryingBeer(false).

//Estado de la nevera
//-------------------
//Inicialemente la nevera está cerrada
fridgeOpen(false).
//Hay dos cervezas disponibles
available(beer,2).

/* Planes */

//Para registrar que el dueño ha bebido un sorbo.
//Si se agota el número de sorbos se le notifica tanto al dueño como
    al robot.
+!sip(beer)[source(owner)]
   <-  ?sipCount(Sc);
            if (Sc > 0) {
                 NewValue = Sc - 1;
                 -+sipCount(NewValue);
```

```
26                  .print("Taking sip: ",NewValue);
27              }else{
28                  .send(robot,untell,has(owner,beer));
29                  .send(owner,untell,has(owner,beer));
30                  .print("No more beer!");
31              }.
32
33  //Cuando se actualiza el número de sorbos, si este valor
34  //es positivo se notifica al dueño y al robot que ya el dueño tiene
        cerveza.
35  +sipCount(C)
36     <-  if (C > 0) {
37                  .send(owner,tell,has(owner,beer));
38                  .send(robot,tell,has(owner,beer));
39              }.
40
41  //Para actualizar el estado de la nevera a 'abierta'
42  +!open(fridge)[source(S)]:available(beer,AvB)
43     <-  .print(S, " is opening the fridge...");
44         -+fridgeOpen(true).
45
46  //Para tomar una cerveza. Debe haber cerveza disponible, la nevera
47  //debe estar abierta y el robot no debe llevar una cerveza consigo.
48  //Si se agota la cerveza se notifica al robot y se actualizan las
49  //creencias necesarias.
50  @pgetbeer[atomic]
51  +!get(beer)[source(S)]
52    : available(beer,AvB) & fridgeOpen(true) &
53      AvB>0  &  carryingBeer(false)
54     <-  NewValue = AvB - 1 ;
55         if (NewValue = 0)
56          {.send(robot,untell,available(beer,fridge));};
57         -+available(beer,NewValue);
58         -+carryingBeer(true).
59
60  //Para actualizar el estado de la nevera a 'cerrada'.
61  //Se envía el stock restante al robot.
62  +!close(fridge)[source(S)]
63     <-   .print(S, " is closing the fridge...");
64          -+fridgeOpen(false);
65          ?available(beer,AvB);
66          .send(robot,tell,stock(beer,AvB)).
```

```
67
68  //Para entregar la cerveza al dueño. Se reinicia la cantidad de
        sorbos a 10
69  //y se actualiza que el robot ya no lleva cerveza consigo.
70  +!hand_in(beer)
71     <-  if (carryingBeer(true)){
72         .print("Beer given in hand to owner. 10 sips left.");
73                 -+sipCount(10);
74                 -+carryingBeer(false);
75             }.
76
77  //Para actualizar la cantidad de productos disponibles tras
78  //una entrega realizada por el mercado.
79  +!deliver(Product,Qtd)
80     <-  -+available(Product,Qtd);
81         .print(Qtd," unities of ",Product," delivered!").
```

## A.3. Salida por consola para el ejemplo de acápite 3.6

En esta sección se muestra la salida en pantalla generada tras la ejecución del ejemplo propuesto en la sección 3.6.

```
1  [environmentAg] robot is opening the fridge...
2  [robot] Moving to owner
3  [robot] Consumed: 1
4  [environmentAg] robot is closing the fridge...
5  [environmentAg] Beer given in hand to owner. 10 sips left.
6  [environmentAg] Taking sip: 9
7  [environmentAg] Taking sip: 8
8  [environmentAg] Taking sip: 7
9  [environmentAg] Taking sip: 6
10 [environmentAg] Taking sip: 5
11 [environmentAg] Taking sip: 4
12 [environmentAg] Taking sip: 3
13 [environmentAg] Taking sip: 2
14 [environmentAg] Taking sip: 1
15 [environmentAg] Taking sip: 0
16 [environmentAg] No more beer!
17 [robot] Moving to fridge
18 [environmentAg] robot is opening the fridge...
19 [robot] Moving to owner
20 [robot] Consumed: 2
21 [environmentAg] robot is closing the fridge...
22 [robot] The beer in stock has fnished.
```

```
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
[robot]  * Starting and making proposal to supermarket *
[supermarket]  - I've received a request for doing: beer and I'm AGREE. -
[supermarket]  - Informing of task done -
[robot]  * The task 3.2.32.deliverBeer was done successfully by agent supermarket.
    Result: delivered(beer,5,2) *
[environmentAg] 5 unities of beer delivered!
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 3
[environmentAg] robot is closing the fridge...
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 4
[environmentAg] robot is closing the fridge...
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
```

```
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 5
[environmentAg] robot is closing the fridge...
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 6
[environmentAg] robot is closing the fridge...
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 7
[environmentAg] robot is closing the fridge...
[robot] The beer in stock has fnished.
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
```

```
[robot]  * Starting and making proposal to supermarket *
[supermarket]  - I've received a request for doing: beer and I'm AGREE. -
[supermarket]  - Informing of task done -
[environmentAg] 5 unities of beer delivered!
[robot]  * The task 3.2.54.deliverBeer was done successfully by agent supermarket.
    Result: delivered(beer,5,3) *
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 8
[environmentAg] robot is closing the fridge...
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 9
[environmentAg] robot is closing the fridge...
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
[environmentAg] Taking sip: 3
[environmentAg] Taking sip: 2
[environmentAg] Taking sip: 1
[environmentAg] Taking sip: 0
[environmentAg] No more beer!
[robot] Moving to fridge
[environmentAg] robot is opening the fridge...
[robot] Moving to owner
[robot] Consumed: 10
[environmentAg] robot is closing the fridge...
[environmentAg] Beer given in hand to owner. 10 sips left.
[environmentAg] Taking sip: 9
[environmentAg] Taking sip: 8
[environmentAg] Taking sip: 7
[environmentAg] Taking sip: 6
[environmentAg] Taking sip: 5
[environmentAg] Taking sip: 4
```

```
177 [environmentAg] Taking sip: 3
178 [environmentAg] Taking sip: 2
179 [environmentAg] Taking sip: 1
180 [environmentAg] Taking sip: 0
181 [environmentAg] No more beer!
182 [owner] Message from robot: The Department of Health does not allow me to give you
        more than 10 beers a day! I am very sorry about that!
```

# Bibliografía


[1] Jade. `http://jade.tilab.com`.

[2] Magentix 2, An Open Multi-agent Systems Platform. `http://www.gti-ia.upv.es/sma/tools/magentix2/index.php`.

[3] Eclipse.org consortium. Eclipse Homepage, 2012.

[4] J. M. Alberola, J. M. Such, A. Espinosa, V. Botti, and A. García-Fornes. Magentix: a Multiagent Platform Integrated in Linux. In *EUMAS*, pages 1–10, 2008.

[5] B. Alfonso, V. Botti, A. Garrido, and A. Giret. A mas-Based Infrastructure for Negotiation and its Application to a Water-Right Market. In *Third International Workshop on Infrastructures and Tools for Multiagent Systems*, Valencia, 2012.

[6] B. Alfonso, E. Vivancos, V. Botti, and A. García-Fornes. Integrating Jason in a Multiagent Platform with Support for Interaction Protocols. In *Proceedings of the compilation of the co-located workshop on AGERE!'11*, SPLASH '11 Workshops, pages 221–226, New York, NY, USA, 2011. ACM.

[7] J. L. Austin. *How to do Things With Words*. Harvard University Press, Cambridge, Mass., 1975.

[8] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley and Sons, 2007.

[9] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in Agent Speak Usign Jason*. John Wiley & Sons, 2007.

[10] V. Botti, A. Garrido, A. Giret, and P. Noriega. *The Role of MAS as a Decision Support Tool in a Water-Rights Market*, volume 7068, pages 35–49. Springer, Berlin / Heidelberg, 2012. springerlink:10.1007/978-3-642-27216-5_4.

[11] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI Agent System Combining Middleware and Reasoning. In M. C. M. K. R. Unland, editor, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhäuser-Verlag, 9 2005.

[12] G. Caire, D. Gotta, and M. Banzi. WADE: A Software Platform to Develop Mission Critical Applications Exploiting Agents and Workflows. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 29–36, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[13] G. Caire, M. Porta, E. Quarantotto, and G. Sacchi. Wolf - An Eclipse Plug-In for wade. In *Proceedings of the 2008 IEEE 17th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '08, pages 26–32, Washington, DC, USA, 2008. IEEE Computer Society.

[14] C. Cheong and M. Winikoff. Hermes: Implementing Goal-Oriented Agent Interactions. In *In Proceedings of the Third international Workshop on Programming Multi-Agent Systems (ProMAS)*, 2005.

[15] M. Dinkloh and J. Nimis. A Tool for Integrated Design and Implementation of Conversations in Multi-Agent Systems. In *In Proc. of the 1st International Workshop on Programming Multiagent Systems: languages, frameworks, techniques and tools, ProMAS-03, held with AAMAS-03*, pages 84–98. Springer. To, 2003.

[16] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *CIKM '94 Proceedings of the Third International Conference on Information and Knowledge Management*.

[17] R. L. Fogués. Magentix2: Una Nueva Plataforma para Sistemas Multiagente Abiertos. Master's thesis, Universitat politécnica de València, 2010.

[18] R. L. Fogués, J. M. Alberola, J. M. Such, A. Espinosa, and A. García-Fornes. Towards Dynamic Agent Interaction Support in Open Multiagent Systems. In *Proceedings of the 13th International Conference of the Catalan Association for Artificial Intelligence*, volume 220, pages 89–98. IOS Press, 2010.

[19] Foundation for Intelligent Physical Agents. *FIPA XC00025E: FIPA Interaction Protocol Library Specification*.

[20] Foundation for Intelligent Physical Agents. *FIPA XC00061D: FIPA ACL Message Structure Specification*.

[21] A. Garrido and J. Calatrava. Water Markets and Customary Allocation Rules: Explaining Some of the Difficulties of Designing Formal Trading Rules. *Journal of Economic Issues*, XL 1:27–24, 2006.

[22] C. Hahn, I. Zinnikus, S. Warwas, and K. Fischer. From Agent Interaction Protocols to Executable Code: A Model-Driven Approach. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1199–1200, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[23] J. Honey-Roses. Assessing the Potential of Water Trading in Spain. in: Enr 319 advanced international environmental economics, 2007.

[24] J. Mcginnis and D. Robertson. Dynamic and Distributed Interaction Protocols. In *In Proceedings of the AISB 2004 Convention*, pages 45–54, 2004.

[25] J. J. Odell, H. V. D. Parunak, and B. Bauer. Representing Agent Interaction Protocols in uml. In *IN OMG DOCUMENT AD/99-12-01. INTELLICORP INC*, pages 121–140. Springer-Verlag, 2001.

[26] A. Pokahr, L. Braubach, A. Walczak, and W. Lamersdorf. *Developing Multi-Agent Systems with JADE*, chapter Jadex - Engineering Goal-Oriented Agents, pages 254–258. Wiley and Sons, 2007.

[27] M. Purvis, S. Cranefield, M. Nowostawski, and D. Carter. Opal: A Multi-level Infrastructure for Agent-Oriented Software Development. University of Otago, Dunedin, New Zealand, Information Science Discussion Paper Series, ISSN 1172-6024, 2002.

[28] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In R. van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[29] M. W. Rosegrant and R. Gazmuri S. Reforming Water Allocation Policy Through Markets in Tradable Water Rights: Lessons From Chile, Mexico, and California. EPTD discussion papers 6, International Food Policy Research Institute (IFPRI), 1994.

[30] M. Thobani. Formal Water Markets: Why, When, and How to Introduce Tradable Water Rights. *World Bank Research Observer*, 12(2):161–79, 1997.