Universitat Politècnica de València
Departament de Sistemes Informátics i Computació

Integración de agentes deliberativos en la plataforma SPADE: Desarrollo de pGomas

Autor: **Frayle Pérez, Sergio**

Tutor: **Julian Inglada, Vicente Javier**
Cotutor: **Palanca Camara, Javier**

julio de 2019

Resumen

Este trabajo aborda y analiza la importancia de incorporar un comportamiento deliberativo BDI a la plataforma de sistemas multiagente SPADE. Este complemento le da a SPADE una ganancia potencial, ya que la plataforma ahora estará preparada para la programación de agentes híbridos. Como caso de estudio se desarrolla una nueva versión de PGOMAS, en la que los agentes reactivos se reemplazan con los nuevos híbridos implementados. La intención es desarrollar un entorno de sistemas multiagente para la enseñanza del trabajo práctico en asignaturas de IA que resulte atractivo para los estudiantes.

Palabras clave: Sistemas MultiAgente, Agentes Inteligentes, SPADE, ASL, BDI, pGomas.

Abstract

This work addresses and discusses the importance of developing a deliberative BDI behaviour for the SPADE MultiAgent System Platform. This add-on gives SPADE a potential gain, as the platform will now be prepared for programming hybrid agents. A new version of PGOMAS is developed, in which the reactive agents are replaced with the new implemented hybrid ones. The aim is to develop a multiagent system environment for teaching practical work in AI subjects that results attractive to students.

Keywords: Multiagent Systems, Intelligent Agents, SPADE, ASL, BDI, pGomas.

Índice general

Índice de figuras

Capítulo 1

Introducción

La construcción de agentes autónomos con la capacidad de reaccionar en ambientes complejos es de gran interés para la comunidad científica de Inteligencia Artificial. En general, los distintos tipos de agentes se pueden agrupar en tres arquitecturas fundamentales: reactivas, deliberativas e híbridas [1]. Los agentes puramente reactivos no tienen una representación explícita del entorno ni de los otros agentes, por lo que viven en el ciclo percepción-acción. De manera general, se caracterizan por no tener como elemento central de razonamiento un modelo simbólico y por no utilizar razonamiento simbólico complejo [2]. En cuanto a las arquitecturas de los agentes deliberativos, como indica su nombre, se introduce una función deliberativa entre la percepción y la ejecución, que elige la acción correcta basándose en el razonamiento práctico. Finalmente, las arquitecturas híbridas pretenden combinar aspectos de ambos modelos, donde se precisa de una reflexión para un razonamiento a largo plazo y una urgencia de reacción para un comportamiento adecuado a la situación actual del agente. Se suelen estructurar mediante capas organizadas jerárquicamente, donde se aprecian tres niveles de abstracción:

Reactivo Nivel bajo. Se toman decisiones en base a los estímulos recibidos en tiempo real.

Conocimiento Nivel intermedio. Se olvida de los datos que recopila el agente y se centra en el conocimiento que posee del entorno adquirido por él mismo.

Social Nivel más alto. Maneja aspectos sociales del entorno, como información y serie de deseos e intenciones de otros agentes mediante un modelo social.

Motivación

Existen diversas plataformas de sistemas multiagente en la actualidad, entre las que destaca SPADE (Smart Python Agent Development Environment), que provee una interfaz sencilla para crear agentes [3]. Usa un modelo de comunicaciones basado en protocolos de mensajería instantánea que intercambia mensajes en tiempo real entre aplicaciones. Como ventajas sobre otras plataformas se puede mencionar su compatibilidad con lenguajes de contenido muy usados en la actualidad como FIPA y KQML, la posibilidad de usar notificación de presencia, el soporte de los protocolos de transporte HTTP y XMPP, la autenticación de usuario y contraseña para aumentar la seguridad así como una conexión cifrada a la plataforma para garantizar la confidencialidad y la movilidad de los agentes. Estas ventajas han hecho que tenga una gran aceptación en la actualidad y sea muy utilizada por distintos desarrolladores.

Pese a estas cualidades, SPADE tiene una limitación, y es que actualmente no da soporte para crear agentes deliberativos ni por supuesto híbridos. Hay un incentivo inicial y es dotar a esta plataforma de una arquitectura híbrida. Esto le daría a SPADE una mejora sustancial, que se traduce en el uso de la misma en nuevas aplicaciones con fines de investigación, docentes e industriales en las que hoy en día no es posible.

Por otro lado, desarrollar habilidades prácticas en cursos de inteligencia artificial es esencial. Disponemos de varios entornos de simulación de sistemas multiagente que facilitan a los estudiantes dominar los conceptos y técnicas de manera amena. Uno de ellos es JGOMAS (JADE-based Game Oriented Multiagent System) [4], el cual ha resultado muy atractivo para los alumnos del grado de Informática de la UPV en los últimos años. Se basa en un juego de estrategia formado por dos equipos, donde cada uno tiene agentes que interpretan distintos roles, como pueden ser soldados y médicos. El objetivo de uno de los equipos, es capturar la bandera y llevarla a su base, el del otro, impedirlo. Cuenta con una interfaz gráfica que permite visualizar el desarrollo de la partida y el comportamiento de los agentes. Sin duda alguna, esta aproximación, con propósito instructivo, resulta sumamente interesante para los estudiantes.

Sin embargo, JGOMAS está desarrollado en JADE [5], lo cual a veces resulta una barrera difícil de pasar para los estudiantes a la hora de implementar algoritmos y técnicas de inteligencia artificial, como heurísticas que se concentren en encontrar la ruta más corta entre dos puntos, por mencionar alguna. Dada esta problemática se han planteado algunas soluciones, como la descrita en [6], donde se presenta una versión de JGOMAS desarrollada

en Python. Esta propuesta no se completó y por tanto no puede ser usada. Se quiere desarrollar una versión en Python, rediseñada completamente, que haga uso de la mejora que se propone darle a SPADE anteriormente comentada.

Objetivos

El objetivo general perseguido en este trabajo es integrar en SPADE una nueva arquitectura de agentes deliberativos que esté basada en BDI. Esta nueva arquitectura debe permitir desarrollar no solamente agentes deliberativos, sino también agentes híbridos con capacidad tanto reactiva como deliberativa. Además, se desarrollará un ejemplo complejo usando esta arquitectura, que consiste en implementar una versión de JGOMAS en Python que permita crear agentes donde la toma de decisiones sea posible mediante el empleo de abstracciones de más alto nivel como planes, objetivos y creencias.

Para cumplir el objetivo general se trazaron los siguientes objetivos específicos:

- Realizar una revisión bibliográfica del estado del arte acerca de agentes deliberativos e implementaciones disponibles.
- Diseño de un modelo de agente híbrido en SPADE.
- Implementar el modelo propuesto, que le permita a los agentes poder ser programados también desde un nivel de abstracción más elevado.
- Desarrollar una nueva versión del juego "Captura La Bandera" (JGOMAS) sobre SPADE haciendo uso de la nueva arquitectura.
- Proporcionar la infraestructura necesaria para poder disponer del entorno apropiado para el desarrollo de partidas.
- Desarrollar un ejemplo más complejo que haga uso de toda la potencia que brinda la arquitectura.
- Validar el desarrollo realizado.

Estructura del Documento

Este documento se encuentra dividido, para su mejor comprensión, en una Introducción y cinco capítulos más. El segundo capítulo se dedica a pre-

sentar una revisión bibliográfica del estado del arte. En el tercer capítulo se profundiza en cómo se desarrolló la arquitectura propuesta. El cuarto capítulo describe la nueva versión desarrollada de JGOMAS (pGomas). El quinto capítulo detalla una serie de resultados de experimentos que se realizaron. Por último, se agregan las conclusiones, trabajos futuros y la bibliografía consultada.

Capítulo 2

Estado del arte

En este capítulo se presenta una revisión bibliográfica del estado del arte de los principales componentes que se tuvieron en cuenta para realizar este trabajo.

2.1. Plataformas de sistemas multiagente

Como ya se comentó en el Capítulo 1, disponemos de una gran cantidad de plataformas de sistemas multiagente hoy en día [7]. A continuación se mencionan algunas de las más conocidas.

JADE [5] es una plataforma completamente implementada en Java que puede ser ejecutada de manera distribuida en distintas máquinas. La configuración se puede controlar a través de una GUI remota. Es la plataforma de agentes compatible con FIPA más popular en la comunidad académica e industrial [7]. Tiene soporte para HTTPS y brinda muy buena seguridad.

JACK [8] es un entorno multiplataforma para crear, ejecutar e integrar agentes múltiples de nivel comercial. Está construido sobre la lógica del modelo BDI. Está completamente desarrollado en Java, pero no es de código libre. Aún no es compatible con especificaciones FIPA. Es una plataforma con buena seguridad.

Agent Builder es una plataforma de agente que se puede utilizar en numerosas simulaciones. Es compatible con KQML y bastante fácil y simple de usar [7]. No presenta ningún tipo de seguridad. Requiere de una licencia de pago para su uso.

AgentScape [9] es una plataforma desarrollada para soportar el diseño y la implementación de sistemas de agentes distribuidos a gran escala. No es compatible con FIPA ni con KQML y tiene una interfaz gráfica pobre.

Tiene autenticación y la plataforma brinda buena seguridad.

MASON [10] es un kit de herramientas de simulación multiagente de eventos discretos desarrollado en Java. No es compatible con FIPA ni KQML. Solo está disponible para Windows. No presenta ningún tipo de seguirdad.

SPADE es una plataforma que brinda varias ventajas para programar agentes, algunas de ellas mencionas en el Capítulo 1. Puede ejecutarse de manera distribuida en distintas máquinas, incluso si tienen distinto sistema operativo, ya que está implementada en Python. Es de código libre y uso gratuito. Se desarrolló en el grupo de investigación donde se realiza este trabajo. En la siguiente Sección se detalla más acerca de la misma.

2.1.1. SPADE

SPADE es una plataforma de sistemas multiagente muy usada en la actualidad. Brinda varias ventajas ante otras, como ya se ha visto, lo que ha hecho que tenga una gran aceptación a nivel mundial. En esta Sección se profundizará en los agentes que se pueden crear con ella y en el funcionamiento de la misma.

El código ejecutado por cada agente SPADE puede estructurarse en comportamientos, por lo que es posible implementar diferentes aspectos de la lógica del agente por separado [11]. Un comportamiento es un proceso que un agente puede ejecutar.

El modelo de agente está básicamente compuesto por un mecanismo de conexión a la plataforma, un *dispatcher* de mensajes y un conjunto de diferentes comportamientos a los que el *dispatcher* entrega los mensajes [12]. Para poder conectarse al servidor XMPP, cada agente debe tener un identificador (JID [1]) y una contraseña. El *dispatcher* de mensajes actúa como un cartero: cuando llega un mensaje para el agente, lo coloca en el "buzón" correcto. Cuando el agente necesita enviar un mensaje, el *dispatcher* de mensajes lo pone en el flujo de comunicación.

Un agente puede tener asociado varios comportamientos de manera simultánea. SPADE proporciona algunos tipos de comportamientos predefinidos como *Cyclic,One-Shot,Periodic,Time-Out* y *Finite State Machine*. Todos estos heredan de *Cyclic* como se muestra en la Figura 2.1. Los comportamientos *Cyclic* y *Periodic* son útiles para realizar tareas repetitivas. Los comportamientos *One-Shot* y *Time-Out* pueden usarse para realizar tareas ocasionales. La máquina de estados finitos permite que se construyan comportamientos más complejos.

[1] Jabber Identification

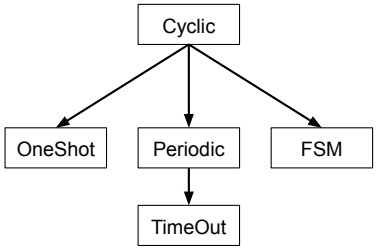


Figura 2.1: Comportamientos básicos de un agente SPADE.

A partir de estos comportamientos se pueden desarrollar otros. Cada uno tiene una plantilla, que puede contener los mismos atributos que un mensaje de SPADE. Cuando el agente recibe un mensaje, el *dispatcher* de mensajes mira la plantilla de este, y lo redirige a la cola del comportamiento que espera este tipo de mensajes. Estos atributos son:

to: el JID del receptor del mensaje.

sender: el JID del emisor del mensaje.

body: el cuerpo del mensaje.

thread: el id del hilo de la conversación.

metadata: un diccionario de cadenas para definir los metadatos del mensajes. Aquí se pueden incluir entre otras, performativas y ontologías.

Una de las características más diferenciadoras de los agentes de SPADE es su capacidad para mantener una lista de contactos (amigos) y recibir notificaciones en tiempo real sobre sus contactos. Esta es una característica heredada de la tecnología de mensajería instantánea que usa la plataforma. Además, cada agente de SPADE tiene una propiedad para gestionar su presencia. Este componente administra la notificación de presencia de un agente, y tiene tres atributos:

State El estado de un mensaje de presencia muestra si el agente está disponible o no. Esto significa que el agente está conectado a un servidor XMPP o no. Adicionalmente, existen varios grados de disponibilidad, entre ellos ocupado, lejos e interesado en recibir mensajes.

Status Se utiliza para explicar con lenguaje natural el estado actual de presencia del agente, como por ejemplo "Trabajando...".

Priority Dado que un agente (y de hecho cualquier usuario XMPP) puede tener múltiples conexiones a un servidor XMPP, se puede establecer la prioridad de cada una de esas conexiones para fijar el nivel de cada una.

La Figura 2.2 muestra gráficamente el manejo de la notificación de presencia en SPADE. Así por ejemplo, el agente "Jean" en un momento dado no está disponible debido a que no se encuentra conectado al servidor XMPP, y el resto de los agentes, que tienen a "Jean" en su lista de contactos, lo pueden saber. Conocer esto antes de contactar a un agente es muy útil, pues tendríamos la información de si va o no a recibir mensajes en tiempo real. Se han usado distintos símbolos para señalar los distintos estados de la disponibilidad que puede tener un agente. Pudiéramos decir que "Billy" y "Jane" se encuentran ocupados y no podríamos hablar con ellos justo en este momento, y "Scott" está disponible para recibir mensajes.

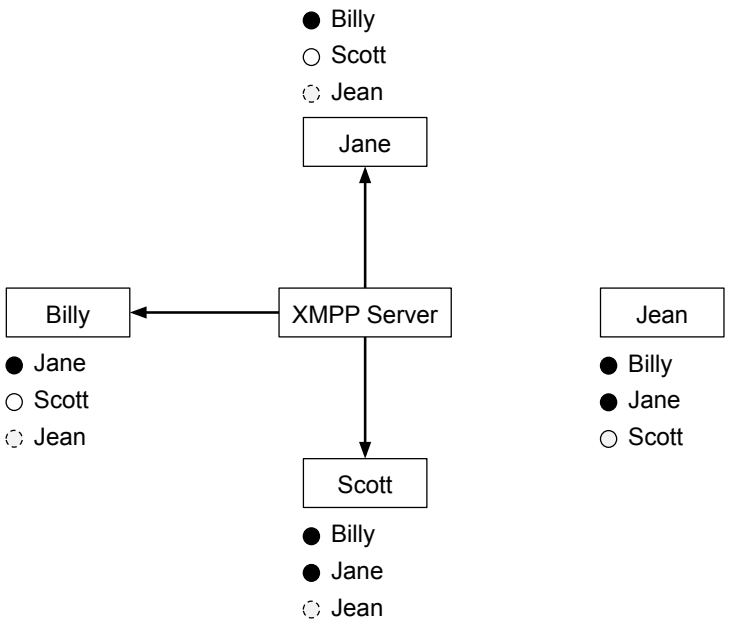


Figura 2.2: Notificación de presencia en SPADE.

Conjuntamente, cada agente SPADE proporciona una interfaz gráfica (que se puede activar o desactivar) a la que se puede acceder a través de la web. En esta se puede consultar su nombre, la lista de sus comportamientos y la de sus contactos, como se muestra en la Figura 2.3. Se pueden revisar sus mensajes entrantes y salientes y el menú del perfil del agente donde puede detenerse. A través de esta interfaz se tiene acceso a todos los comportamientos que se han agregado al agente, tanto los activos como los finalizados, pudiendo detener cualquier comportamiento que esté operando.

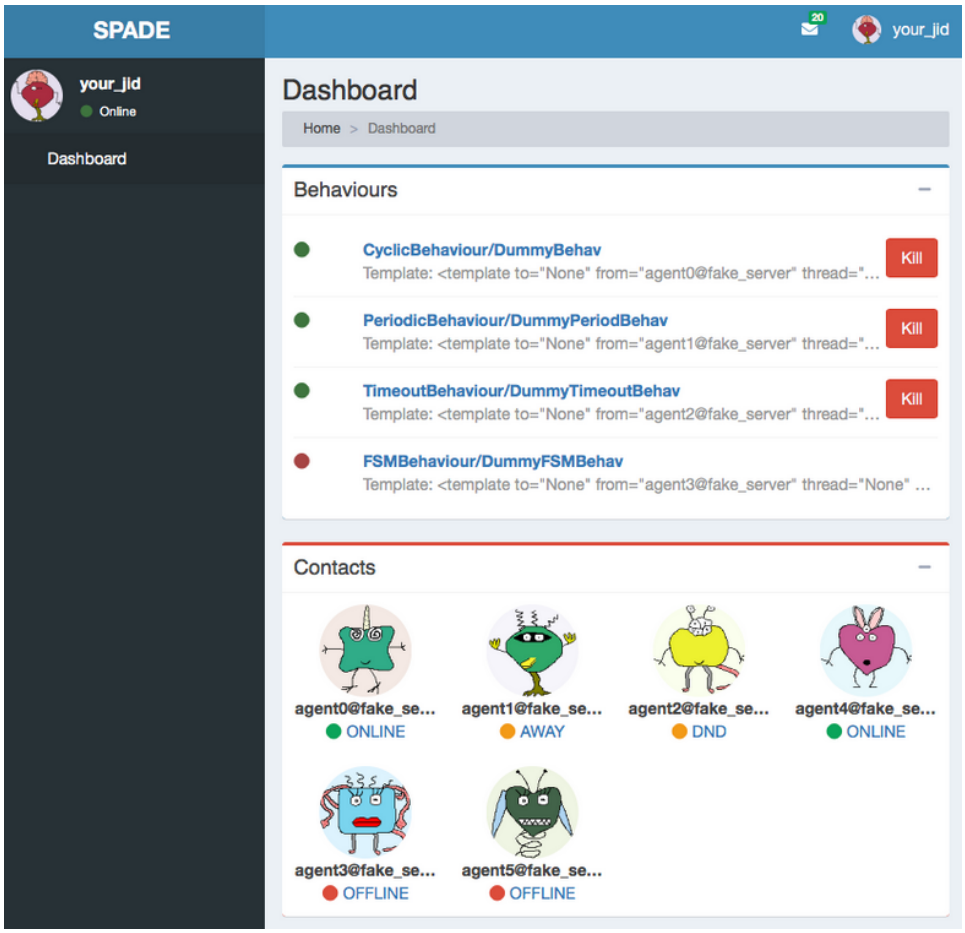


Figura 2.3: Interfaz gráfica de un agente SPADE.

Teniendo en cuenta lo antes expuesto , se presenta la arquitectura de un agente SPADE en la Figura 2.4.

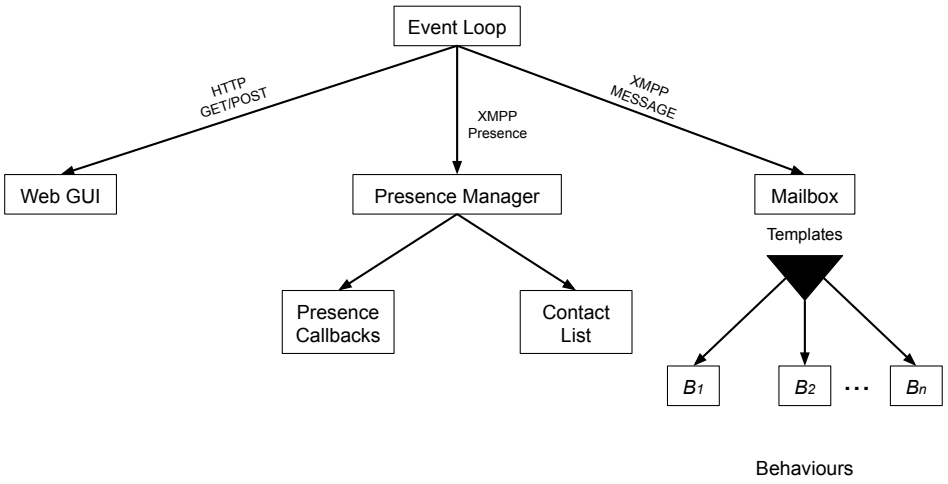


Figura 2.4: Arquitectura de un agente SPADE.

2.2. Arquitecturas deliberativas

Dentro de las arquitecturas deliberativas más destacadas están GRATE, BDI, IRMA y PRS , las cuales se comentan a continuación.

La arquitectura GRATE [13] está diseñada para construir aplicaciones en las que los agentes deben cooperar entre sí para conseguir una serie de metas, pudiendo tomar dos roles notables: como entidad individual o como miembro de una comunidad. Los agentes GRATE tienen dos componentes claramente identificables: una capa de cooperación y control, y un sistema de nivel de dominio. En el nivel del dominio se resuelven problemas del dominio (de control industrial, transporte, etc.) mientras que en el nivel de cooperación y control se tiene como objetivo asegurar que las actividades del agente están coordinadas con las de otros agentes en la comunidad.

El modelo BDI(Belief-Desire-Intention) [14], detallado con mayor profunidad en la Sección 2.2.1, es uno de los más conocidos de las arquitecturas deliberativas [15]. Los agentes BDI tienen un conjunto de creencias, deseos e intenciones. BDI también tiene la capacidad de razonar sobre su propio estado interno, es decir, reflexionar sobre su propias creencias, deseos e intenciones, modificándolas a su gusto.

IRMA es una arquitectura para agentes con recursos limitados que describe cómo un agente selecciona su curso de acción basándose en representaciones explícitas de su percepción, creencias, deseos e intenciones. IRMA

adopta una postura pragmática hacia la arquitectura BDI. En particular, no proporciona un modelo formal explícito de creencias, objetivos e intenciones [16].

PRS [17] es un sistema dotado de las actitudes de creencia, deseo e intención. En cualquier momento dado, las acciones consideradas por PRS dependen no solo de sus deseos u objetivos actuales, sino también de sus creencias e intenciones previamente formadas.

2.2.1. BDI

La arquitectura BDI dota al agente de un conjunto B de creencias, un conjunto D de deseos y un conjunto I de intenciones. La idea es que B contenga hechos del estado actual del entorno y D sea representativo del estado ideal que se desea alcanzar. De esta manera, BDI modela la forma en que un agente, que puede tomar decisiones, logra sus deseos, o cómo cambia el mundo para que sus deseos se conviertan en sus creencias. Para ilustrar mejor la idea se puede pensar en el caso donde un agente puede tener la creencia de estar en una habitación H_1, pero puede desear estar en la habitación H_2. El modelo BDI intenta representar cómo el agente selecciona algún plan de acciones para que logre estar en la habitación H_2, satisfaciendo así su deseo de estar en esa habitación y por ende tener la creencia de estar en esa habitación.

Con el objetivo de lograr este comportamiento, la arquitectura BDI añade las intenciones I antes mencionadas, que se definen como un subconjunto de deseos que el agente tiene que lograr. Este subconjunto está formado por los deseos que no son mutuamente inconsistentes, dígase aquellos que se pueden realizar al mismo tiempo. En el ejemplo anteriormente comentado, el deseo del agente de estar en H_2 y el deseo de estar en otra habitación H_3 al mismo tiempo serían mutuamente inconsistentes. El problema es que existe una restricción física en el logro de ambos deseos, ya que el agente no puede estar en dos lugares al mismo tiempo, por lo que solo puede satisfacer uno de sus deseos a la vez. Este problema aparece en muchos casos, ya que los agentes son entidades con recursos limitados y es posible que no puedan lograr todos sus deseos debido a la falta de recursos. Como resultado, deben seleccionar un subconjunto de aquellos deseos que creen que son capaces de realizar dadas sus limitaciones. Cuando se determinan las intenciones, el agente trata de convertirlas en creencias identificando y siguiendo un plan de acción apropiado, lo cual puede realizarse con la selección de un plan dentro de una biblioteca de planes definidas antes de la ejecución [14, 18]. En la Figura 2.5 se muestra el modelo BDI.

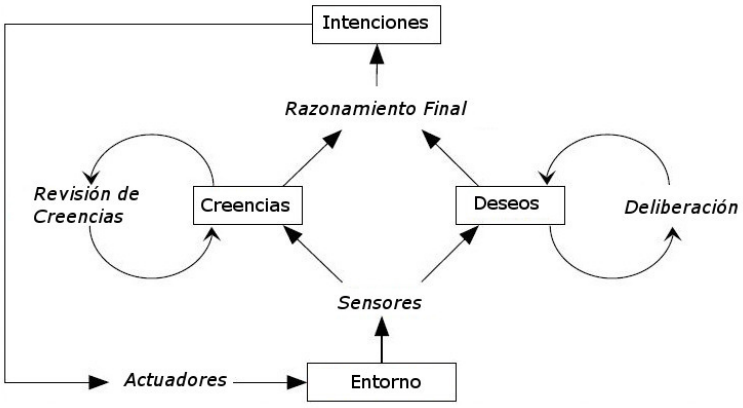


Figura 2.5: Modelo BDI.

La efectividad del modelo BDI en problemas concretos de gran complejidad quedó demostrada desde sus inicios. Entre las aplicaciones más conocidas, se pueden mencionar las siguientes:

- Sistema para el diagnóstico, control y monitoreo de una red de telecomunicaciones en tiempo real (IRTNMS[2])[19].
- Sistema administrador de procesos de negocios (SPOC[3])[20].
- Sistema de control de tráfico aéreo (OASIS[4])[21].
- Sistema para el manejo del mal funcionamiento de un transbordador espacial de la NASA (RCS[5])[22].
- Simulador para la fuerza aérea australiana (SWARMM [6])[23].

El modelo de agentes BDI ha sido usado en la actualidad también en variadas aplicaciones como en IoTS[7] [24], reconfiguración de sistemas de energía [25],ToM[8][26] y predicción de sequías en África [27], por mencionar algunas.

[2]Interactive Real-Time Telecommunications Network Management System
[3]Single Point of Contact
[4]Optimal Aircraft Sequencing using Intelligent Scheduling
[5]Reaction Control System
[6]Smart Whole AiR Mission Model
[7]Internet of Things and Services
[8]Theory of Mind

2.2.2. Implementaciones BDI

Existen varias implementaciones de agentes BDI [28, 29, 30], pero la complejidad del código y las simplificaciones asumidas dificultan su comprensión y sustento teórico. Otras más modernas se desarrollan sobre plataformas multiagentes ya existentes, como BDI4JADE[31] y Jadex [32] donde se implementan como una capa o un $add-on$ sobre JADE siguiendo la arquitectura BDI.

Una implementación clásica es ASL[9], un lenguaje de programación restringido de primer orden con eventos y acciones [33]. Es un lenguaje sencillo con flexibilidad para programar agentes BDI y uno de los más aceptados. Define un conjunto de construcciones de programación con una sintaxis específica para un intérprete que se basa en un modelo de toma de decisiones con esta arquitectura. Las construcciones principales son:

- Creencias: Predicados que representan hechos sobre el estado del entorno y de los agentes.
- Objetivos: Predicados que identifican lo que el agente quiere hacer. Las metas no se almacenan explícitamente en el estado del agente, sino que se declaran como requeridas y se asignan contextualmente a un comportamiento que realizará la meta.
- Eventos: Los eventos conducen el comportamiento de un agente. Internamente, el agente contiene una cola de eventos. En cada iteración del intérprete, se selecciona un evento de la cola de eventos y se procesa mediante la asignación contextual del evento a un controlador de eventos (regla del plan). ASL incluye eventos para la adopción de nuevas creencias, la retractación de las creencias existentes y la adopción de objetivos.
- Reglas del plan: Definen los comportamientos centrales del agente, mapeando contextualmente esos comportamientos a los eventos que los desencadenan. Los comportamientos se especifican como una secuencia de operadores del plan, que soportan la adopción o retracción de creencias, adopción de subobjetivos, consulta de creencias y acciones privadas.

Jason es un intérprete para una versión extendida de ASL desarrollado en JAVA [34]. Es la implementación actual de referencia para el desarrollo de agentes BDI. Entre sus ampliaciones, están:

[9] ASL: AgentSpeak Language

- Negación fuerte.
- Manejo de fallo de planes.
- Anotaciones en las etiquetas del plan.
- Posibilidad de ejecutar un sistema multiagente distribuido.
- Una librería de "acciones internas".
- Extensibilidad mediante acciones internas definidas por el usuario.

Además, en la comunicación de agentes, incluye nuevas performativas, que permiten el intercambio de planes. Las ilocuciones disponibles son brevemente descritas a continuación, donde s y r denotan a los agentes que envían y reciben los mensajes respectivamente.

tell: s pretende que r crea que el contenido del mensaje es cierto.

untell: s pretende que r no crea que el contenido del mensaje es cierto.

achieve: s solicita que r intente alcanzar un estado del entorno donde el contenido del mensaje es cierto.

unachieve: s solicita que r intente dejar la intención de alcanzar un estado del entorno donde el contenido del mensaje es cierto.

tellHow: s le informa a r un plan.

untellHow: s solicita que r descarte cierto plan, o sea, que lo borre de su biblioteca de planes.

askIf: s quiere saber si el contenido del mensaje es cierto para r.

askAll: s quiere todas las respuestas de r a una pregunta.

askHow: s quiere todos los planes de r para cuando se desencadene un evento.

Jason se puede ejecutar sobre algunas plataformas de sistemas multiagente como JADE, AgentScape, Agent Factory y Magentix2. Es la implementación de ASL usada en la actual versión de JGOMAS para describir los comportamientos deliberativos de los agentes. Entre sus limitaciones están:

- Escalabilidad.

- Por defecto no ofrece distribución en distintas máquinas.
- Dificultad para el acceso a datos físicos.

Python-agentspeak es un intérprete para ASL desarrollado en Python [35]. Es casi equivalente a Jason, aunque no están implementadas todas las funcionalidades de este aún. Algunas de sus limitaciones son:

- La carencia de anotación de planes.
- Disponibilidad solo de las ilocuciones de *tell,untell* y *achieve*.
- Ausencia de acciones para manipular planes en la librería estándar.
- Los agentes se crean en una abstracción de entorno "virtual" que no puede ser accedido desde otros procesos.
- Imposibilidad de ejecutar un sistema multiagente distribuido debido al entorno "virtual" en que son creados los agentes. No existe un mecanismo de envío y recepción de mensajes.

No obstante, al estar escrito en Python, es una oferta interesante para tomar como punto de partida e intentar integrar en SPADE. Además, los autores demuestran que tiene una mayor escalabilidad que Jason [36].

2.3. Conclusiones

Como conclusión del estudio realizado, no cabe duda en cuanto a usar como plataforma de agente a SPADE. Además de las ventajas que presenta ante las otras analizadas, está desarrollada en Python y por el grupo de investigación donde se está realizando este trabajo.

Como se mencionó, el modelo BDI es el más utilizado y conocido de las arquitecturas deliberativas. Dentro de las variadas implementaciones de BDI que existen, Jason es la más usada, pero no se podría integrar en SPADE. Se propone utilizar Python-agentspeak, que se supone ser casi equivalente a Jason, y al estar desarrollado en Python sí se podría ejecutar sobre SPADE.

Capítulo 3

Agente Híbrido SPADE

Como ya se comentó, un agente híbrido, tiene tanto comportamiento reactivo como deliberativo, con el fin de tomar las ventajas de ambos. Esta aproximación se lleva a cabo integrando en la misma arquitectura comportamientos que controlen las funciones de bajo nivel y que estén a cargo de la reactividad del sistema, y un módulo de decisión que sea responsable de la toma de decisiones a alto nivel [37]. En el contexto de trabajos colaborativos en sistemas multiagente, se debe mantener un balance adecuado entre los dos extremos, pura reactividad y profunda deliberación social [38]. En este capítulo se plantea el modelo híbrido propuesto y se detalla su implementación. Finalemente, en la Sección 3.3 se muestra cómo instalar el módulo propuesto.

3.1. Agente híbrido

Siguiendo la arquitectura típica de un agente SPADE, mostrada en la Figura 2.4, se propone incluir una capa deliberativa que interactúe con la reactiva. Para crear un modelo híbrido en SPADE, la forma más adecuada de crear las capas de respuesta del agente sería creando un comportamiento cíclico que se encargue de la deliberación y razonamiento a alto nivel. El modelo híbrido propuesto tendrá ahora un entorno deliberativo, como se muestra en la Figura 3.1.

3.1.1. Entorno deliberativo

Para la incorporación de un modelo BDI a SPADE se ha usado el intérprete Python-agentspeak descrito en la Sección 2.2.2. Si bien esta es una

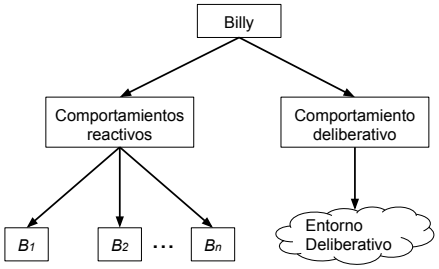


Figura 3.1: Agente híbrido.

alternativa que permite el tratamiento de ficheros ASL y la creación de entornos deliberativos desde SPADE, se introduce el problema de que Python-agentspeak crea un entorno abstracto "virtual" donde se simula una comunicación entre agentes. El entorno deliberativo se crea a partir del fichero ASL que se le asocie al agente.

Al igual que Jason, Python-agentspeak permite crear acciones que pueden ser ejecutadas por los agentes. Estas acciones se definen dentro de la plataforma SPADE y se pueden llamar desde un código escrito en ASL. Aprovechando esto y el envío y recepción de mensajes que brinda SPADE, se puede definir e implementar la comunicación entre agentes con comportamiento deliberativo. De esta manera, los entornos "virtuales" de Python-agentspeak serían más reales ya que podrá intercambiarse información de uno a otro, como se muestra en la Figura 3.2, ya sea con una ilocución de *tell,untell*, etc. En este ejemplo "Billy" y "Jean" tienen su entorno deliberativo brindado por Python-agentspeak, donde están las creencias, planes, etc. La comunicación entre los dos entornos deliberativos, mostrada con líneas discontinuas, no está soportada por Python-agentspeak. Para lograrlo, se hará a través de SPADE, como muestran las líneas continuas. En la siguiente sección, se describe cómo se implementó esto.

3.1.2. BDIBehaviour

Como se explicó en la Sección 2.1.1, SPADE permite crear comportamientos que hereden de los básicos y asignárselos al agente. Siguiendo esta filosofía, para crear agentes deliberativos e híbridos, se desarrolló un nuevo comportamiento, *BDIBehaviour*. A este comportamiento se le especifica una plantilla con performativa "BDI" y se ejecuta de manera cíclica. Servirá de intermediario entre el mundo deliberativo brindado por Python-agentspeak

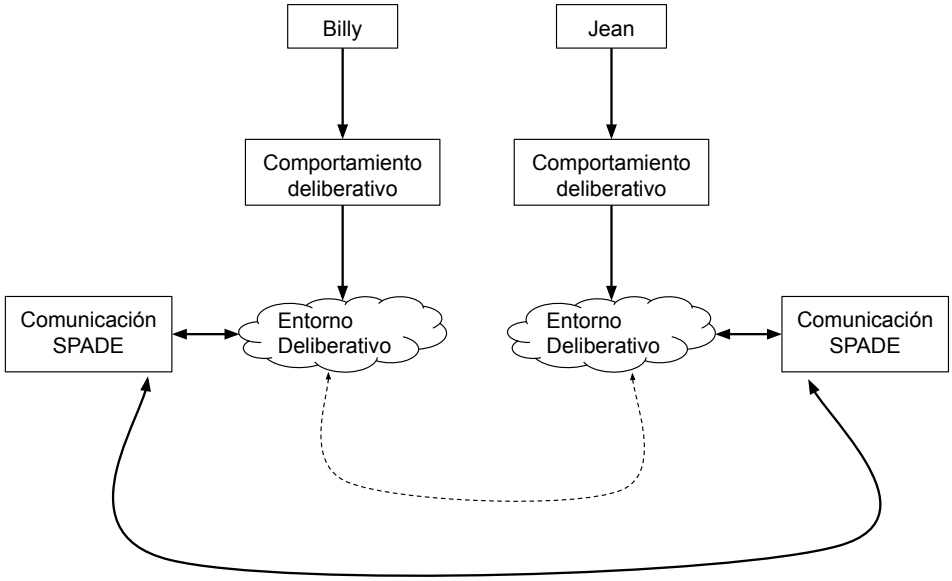


Figura 3.2: Intercambio de información.

que se desea agregar y los comportamientos reactivos de SPADE ya existente, como se muestra en la Figura 3.3.

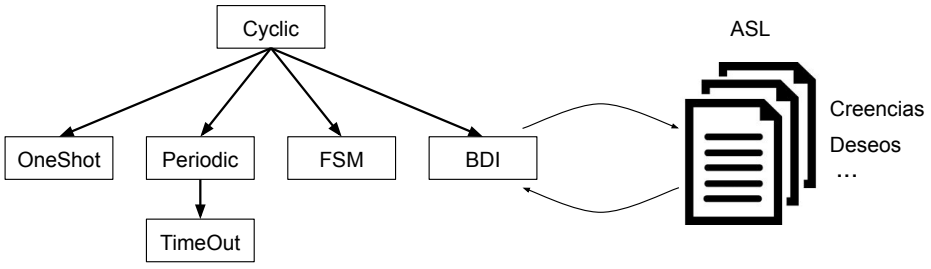


Figura 3.3: Comportamiento BDI. Interacción con el entorno deliberativo.

Dentro de este comportamiento se deben garantizar dos aspectos que serán requeridos por cualquier aplicación que haga uso del mismo:

1. El agente debe ser capaz de consultar las creencias y además modificarlas (ya sea cambiando sus valores o eliminándolas). Cada inserción

o borrado de creencias desde el comportamiento deliberativo debe generar los eventos correspondientes de ASL.

2. Implementar un mecanismo de comunicación de forma que distintos agentes que tengan comportamiento BDI sean capaces de intercambiarse mensajes con información de su entorno deliberativo.

Inserción, eliminación y modificación de creencias

Para garantizar el primer aspecto, se han implementado las siguientes funciones dentro del comportamiento deliberativo. Estas funciones pueden ser invocadas desde otros comportamientos para poder inyectar y recuperar conocimiento.

- `get_belief`
  Para consultar una creencia que tenga el agente del entorno.
- `get_beliefs`
  Para consultar todas las creencias que tenga el agente del entorno.
- `get_belief_value`
  Para consultar el valor que toma el predicado de una creencia que tenga el agente del entorno.
- `set_belief`
  Para establecer o modificar una creencia que tenga el agente del entorno.
- `remove_belief`
  Para eliminar una creencia que tenga el agente del entorno.

La función `set_belief` recibe la creencia que se desea establecer, y el o los valores que tomará el predicado de la misma. Buscará si el agente tiene alguna creencia que iguale el predicado especificado. En caso negativo, la añadirá y generará un evento de adición de ASL. De lo contrario, si existe, pero no coinciden el o los valores establecidos, la eliminará y generará un evento de borrado, para posteriormente añadirla con el o los nuevos valores establecidos y generar un evento de adición. Si coincidieran el o los valores determinados y el predicado, no hará nada, pues es una creencia que ya existe.

La función `remove_belief` recibe la creencia que se desea eliminar. Buscará si existe esta creencia y en caso de ser así, la eliminará, generando un evento de borrado. Si dicha creencia no existe, no hará nada.

Comunicación

Para resolver el tema de la comunicación, se ha implementado una acción de envío. Esta acción, `.send`, se llama desde un fichero ASL, especificando el agente receptor, la ilocución y el contenido del mensaje. Dentro del comportamiento BDI, se construye un mensaje de SPADE, con el contenido a enviar, destinatario y receptor correspondientes, y en los metadatos se especifica la ilocución y la performativa ("BDI") con que se enviará el mensaje. Una vez creado, el mensaje es puesto en el flujo de comunicación de SPADE por el *dispatcher* de mensajes.

La rutina principal de este comportamiento, que se ejecuta de manera cíclica, se encarga de recibir mensajes. Solo recibirá mensajes cuyos atributos igualen a aquellos definidos en la plantilla del comportamiento, dígase mensajes que tengan una performativa "BDI". Lo primero que se debe hacer es extraer de los metadatos la ilocución del mensaje. Se debe garantizar que la ilocución del mensaje modifique la base de creencias o biblioteca de planes según corresponda y genere los eventos adecuados. Por ejemplo, "tell" debe añadir la creencia especificada en el cuerpo del mensaje y generar un evento de inserción mientras que "untell" debería eliminarla y generar un evento de borrado. En cambio, "achieve" debe añadir la meta que se precisa en el cuerpo del mensaje y generar un evento de inserción. Teniendo esto hecho, se puede delegar en Python-agentspeak la actualización del estado del entorno deliberativo.

3.2. Ejemplos del modelo propuesto

Para ilustrar el uso del modelo de agente híbrido que se propone, se realizaron una serie de experimentos y pruebas. En esta sección se ejemplifican dos de ellos, que se encuentran disponibles en el github del proyecto.

El objetivo del primer experimento es comprobar que el comportamiento *BDIBehaviour* implementado funcione correctamente como un intermediario entre los comportamientos reactivos de SPADE y el entorno deliberativo del agente híbrido. En él se crea un agente que se encargará de contar. El modo de conteo puede ser ascendente o descendente, y se realizará desde código ASL. El comportamiento reactivo del agente solo se encargará de consultar y añadir creencias. Para esto tiene 3 comportamientos *PeriodicBehaviour*, con distinta frecuencia cada uno, y 1 comportamiento *TimeoutBehaviour*, como se muestra a continuación:

```
from spade_bdi.bdi import BDIAgent
from spade.template import Template
from spade.behaviour import PeriodicBehaviour
from spade.behaviour import TimeoutBehaviour
from datetime import datetime
from datetime import timedelta


class CounterAgent(BDIAgent):
    async def setup(self):
        template = Template(metadata={"performative": "B1"})
        self.add_behaviour(self.Behav1(period=1,
                           start_at=datetime.now()), template)

        template = Template(metadata={"performative": "B2"})
        self.add_behaviour(self.Behav2(period=5,
                           start_at=datetime.now()), template)

        template = Template(metadata={"performative": "B3"})
        self.add_behaviour(self.Behav3(period=10,
                           start_at=datetime.now()), template)

        start_in = datetime.now() + timedelta(seconds=60)
        template = Template(metadata={"performative": "B4"})
        self.add_behaviour(self.Behav4(start_at=start_in), template)

    class Behav1(PeriodicBehaviour):
        async def on_start(self):
            self.counter = self.agent.bdi.get_belief_value("counter")[0]

        async def run(self):
            if self.counter != self.agent.bdi.get_belief_value("counter")[0]:
                self.counter = self.agent.bdi.get_belief_value("counter")[0]
                print(self.agent.bdi.get_belief("counter"))

    class Behav2(PeriodicBehaviour):
        async def run(self):
            self.agent.bdi.set_belief('counter', 0)

    class Behav3(PeriodicBehaviour):
        async def run(self):
            try:
                type = self.agent.bdi.get_belief_value("type")[0]
                if type == 'inc':
                    self.agent.bdi.set_belief('type', 'dec')
                else:
                    self.agent.bdi.set_belief('type', 'inc')
            except Exception as e:
                print("No␣belief␣'type'.")
```

```
class Behav4(TimeoutBehaviour):
    async def run(self):
        self.agent.bdi.remove_belief('type', 'inc')
        self.agent.bdi.remove_belief('type', 'dec')
```

De los comportamientos periódicos, uno de ellos se encargará de leer el valor de la creencia "counter" del agente en su entorno deliberativo, mediante la función `get_belief_value`, y actualizar su valor cada 1 segundo. Otro establecerá el valor de esta creencia a 0, cada 5 segundos, haciendo uso de `set_belief`. Finalmente, el tercer comportamiento periódico se encargará de modificar la creencia "type" cada 10 segundos, alternando su valor entre "inc" y "dec". Para esto se consulta la creencia con `get_belief_value` y en función del valor que tenga se establece una creencia con `set_belief`. El *TimeoutBehaviour* se disparará transcurrido 1 minuto después de la ejecución, y hará uso de `remove_belief` para eliminar las creencias "type(inc)" y "type(dec)".

A continuación se muestra una parte del código en ASL. Los planes mostrados se lanzan según el contexto en el que estemos. Cuando el agente tenga la creencia "type(inc)", leerá el valor de la creencia "counter", y añadirá una nueva creencia con el valor que tenía incrementado en 1. En cambio si se cree que "type(dec)" es cierto, se hará lo contrario. En caso que no exista una creencia que contenga el predicado "type" sencillamente esperará.

```
+!obj2: type(inc)
 <-
  .print("Increasing");
  ?counter(X);
  -+counter(X+1);
  .wait(1000);
  !obj2.

+!obj2: type(dec)
 <-
  .print("Decreasing");
  ?counter(X);
  -+counter(X-1);
  .wait(1000);
  !obj2.
```

```
+!obj2: not type(_)
 <-
  .print("Waiting");
  .wait(1000);
  !obj2.
```

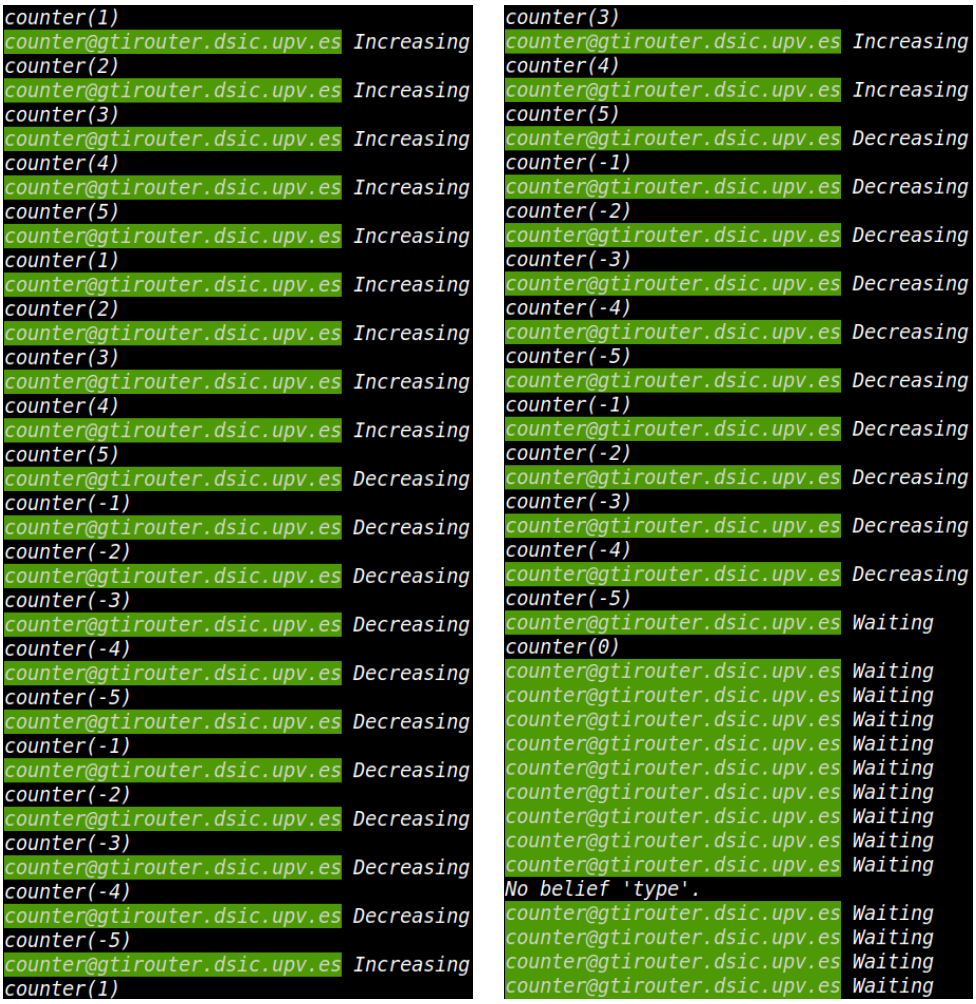
Se pudo verificar que las creencias del agente se podían consultar y eliminar, así como añadir. Cabe destacar que pese a que el ejemplo es bien sencillo, demuestra la potencialidad del agente propuesto. Por un lado, se tiene una parte reactiva que se encarga de interactuar con el entorno, y por otro, una parte con un nivel de abstracción más elevado donde se pueden tomar decisiones con un razonamiento más profundo. En la Figura 3.4 se muestra la salida por pantalla de una ejecución de este primer experimento.

El segundo ejemplo que se presenta tiene como objetivo confirmar el intercambio de información entre agentes que sigan el modelo propuesto. Para ello se han creado 3 agentes. Uno de ellos tendrá el rol de jefe y los otros dos el de subordinado. Al igual que el ejemplo anterior, se realizará un conteo alternado, pero esta vez el jefe le dirá a los subordinados si el conteo es ascendente o descendente y en cuánto se incrementará o decrementará el mismo.

El jefe incorpora además del comportamiento BDI desarrollado, un *PeriodicBehaviour* y un *TimeoutBehaviour*. El comportamiento periódico se encargará de modificar la creencia "type" al igual que en el ejemplo anterior, y el *TimeoutBehaviour* será el mismo. Para la comunicación con los subordinados, se ha usado la acción implementada `.send`. Con esta se le enviará a cada uno de los subordinados un mensaje con ilocución *tell* y una creencia, que puede ser "increase(X)" o "decrease(X)", donde X es la cantidad a decrementar o incrementar. Al dispararse el *TimeoutBehaviour* y eliminarse las creencias "type", el jefe enviará un mensaje con ilocución "untell", para que los subordinados no crean que "increase(X)" o "decrease(X)" son ciertas.

A continuación se presenta parte del código ASL del jefe. Las creencias "slave1" y "slave2" se le establecen al jefe desde un comportamiento reactivo SPADE haciendo uso de `set_belief`.

```
+!obj2: type(inc)
 <-
  ?slave1(X);
  ?slave2(Y);
  .send(X, tell, increase(2));
```



Primeros 21 segundos. Pasado los 60 segundos.

Figura 3.4: Salida por pantalla de una ejecución del primer ejemplo. Las líneas que empiezan con el nombre del agente, en verde, son impresas desde el fichero ASL. Las otras desde el código en Python. Se ha capturado y mostrado el estado de la terminal solo al inicio y al final de la ejecución.

```
  .send(Y, tell, increase(5));
  .wait(2000);
  !obj2.

+!obj2: type(dec)
 <-
  ?slave1(X);
  ?slave2(Y);
  .send(X, tell, decrease(2));
  .send(Y, tell, decrease(5));
  .wait(2000);
  !obj2.

+!obj2: not type(_)
 <-
  ?slave1(X);
  ?slave2(Y);
  .print("Finishing");
  .send(X, untell, increase(2));
  .send(Y, untell, increase(5));
  .send(X, untell, decrease(2));
  .send(Y, untell, decrease(5)).
```

A los subordinados se les establece una creencia "master" para que sepan quién es el jefe al que deben escuchar e incrementar o decrementar su contador. El código ASL de los subordinados se muestra a continuación.

```
+increase(Inc)[source(S)]: master(M) & .substring(M,S,R)
 <-
  .print("increasing");
  ?counter(X);
  .print(X);
  -+counter(X+Inc).

+decrease(Dec)[source(S)]: master(M) & .substring(M,S,R)
 <-
  .print("decreasing");
  ?counter(X);
  .print(X);
  -+counter(X-Dec).
```

```
-increase(Inc)[source(S)]: master(M) & .substring(M,S,R)
 <-
  .print("DELETING increase BELIEF from an untell message").

-decrease(Dec)[source(S)]: master(M) & .substring(M,S,R)
 <-
  .print("DELETING decrease BELIEF from an untell message").
```

Con este ejemplo se pudo comprobar que el mecanismo de intercambio de información propuesto funciona de manera correcta. Aunque en este ejemplo solo se hace uso de ilocuciones "tell" y "untell", también se probó con "achieve", obteniendo los resultados esperados. También, se habilitó y deshabilitó el comportamiento deliberativo del agente subordinado 2 como se puede ver en la Figura 3.5, donde se muestra una captura de la salida por pantalla de una ejecución de este ejemplo. Como se puede ver, el subordinado 2 no recibe los mensajes "untell" cuando el jefe los envía, puesto que para ese entonces ya se ha detenido su comportamiento deliberativo.

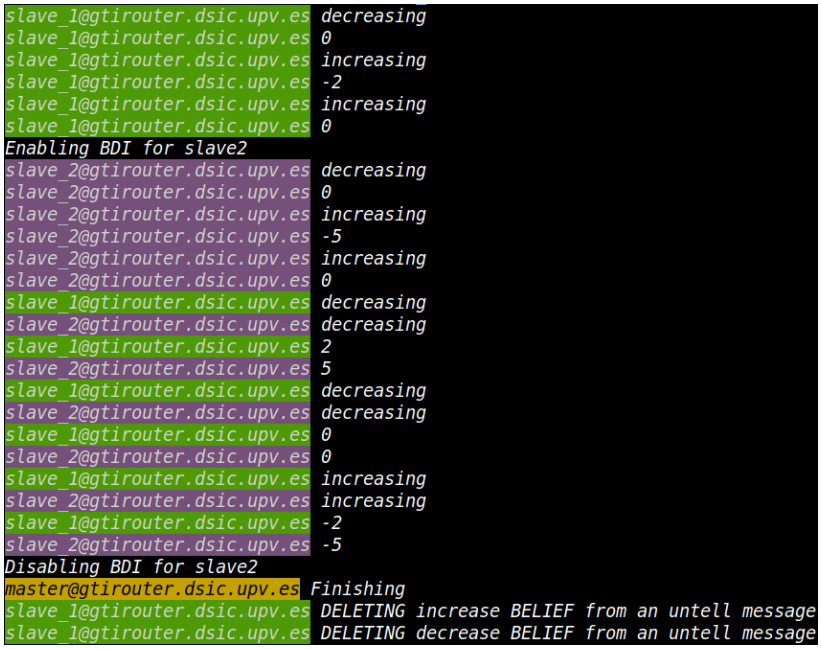


Figura 3.5: Salida por pantalla de una ejecución del segundo ejemplo.

3.3. Implantación

En **https://github.com/sfp932705/spade_bdi** se encuentra disponible el código fuente del comportamiento BDI que permite crear agentes híbridos. Para instalarlo, se puede descargar el código, y ejecutar en el directorio raíz `python setup.py install`.

Alternativamente, para facilitar el proceso de instalación de toda la plataforma de una manera sencilla se ha creado un instalador usando *PyPi*. Si se desea, se puede instalar con *pip3*, ejecutando `pip3 install spade_bdi`.

El proyecto contiene una carpeta con ejemplos (incluyendo los dos comentados en la Sección 3.2), que sirven de base para futuros usuarios que deseen implementar este tipo de agente. La ejecución es bien sencilla. Así, para lanzar el segundo ejemplo descrito en la Sección 3.2, basta con escribir en una terminal desde ese directorio la siguiente línea:

```
python control.py --server gtirouter.dsic.upv.es
```

Capítulo 4

pGomas

A lo largo de este capítulo se describe la versión del juego "Captura La Bandera" desarrollada sobre SPADE usando agentes híbridos. Finalmente, en la Sección 4.7 se muestra cómo instalar el juego. La partida se puede observar a través de visores gráficos que se pueden conectar a un servidor HTTP que se lanza en la aplicación.

4.1. Descripción del juego

En pGomas existen dos equipos, Aliados y Eje, que se encaran en un terreno limitado durante un tiempo limitado. Cada equipo tiene su propia base, donde se sitúan inicialmente cada uno de sus integrantes. Como el nombre sugiere, el objetivo del juego es capturar la bandera.

El equipo de los Aliados debe intentar capturar la bandera y llevarla a su base. Por el contrario, el Eje debe impedir esto. El equipo de los Aliados ganará la partida si consigue su objetivo, mientras que el del Eje lo hará si elimina a todos los miembros de los Aliados o si se agota el tiempo.

Existen distintos roles que pueden asumir los integrantes de un equipo. Independientemente de la función que realice cada uno, siempre estará equipado con armas de fuego y empezará la partida con el máximo de municiones que puede cargar. Los disparos afectan tanto a los contrarios como a los asociados al mismo equipo. Hasta el momento solo hay tres tipos de combatientes: soldados, médicos y operadores de campo; pero se podría extender. Las principales diferencias entre ellos son las siguientes:

- Soldados:
  Los soldados son el eslabón del equipo más apto para combatir. Tienen

armas que infligen más daño que las del resto. Pueden recibir llamadas de refuerzo por parte de sus compañeros.

- Médicos:
  Como su nombre indica, los médicos son los encargados de curar a los miembros de su equipo. Para ello, llevan paquetes de medicina que le proporcionan una determinada cantidad de salud a aquel integrante que lo reciba. Pueden recibir llamadas de asistencia médica por parte de sus compañeros y acudir a ellos.

- Operadores de campo:
  La función de estos es proporcionar paquetes de municiones a quien lo solicite mediante llamadas de recargo de municiones. Llevan paquetes que le aumentará el número de municiones a quien lo reciba en una cantidad determinada.

Los paquetes creados por los médicos y los operadores de campo se mantienen en la posición en que han sido soltados hasta que un compañero lo tome (pasándole por encima) o vencido un tiempo establecido. Un paquete de salud o de munición puede ser tomado por un enemigo también.

4.2. Agentes de pGomas

En pGomas existen varios agentes que interactúan entre sí durante la partida, algunos son puramente reactivos y otros incorporan razonamiento deliberativo siguiendo la arquitectura propuesta en el Capítulo 3. En las siguientes secciones se describen las funcionalidades de cada uno de ellos. De ahora en adelante se referirá a un combatiente como agente tropa.

4.2.1. Agente Manager

El Manager es una especie de Gran Hermano que monitorea la partida desde arriba y ofrece el servicio de gestor. Sus funciones principales son la coordinación y sincronización de los otros agentes y la aportación de información de lo que está en el campo de visión de estos. Entre sus tareas están:

- Iniciar la partida. Le manda un mensaje de inicio a los agentes de los equipos Aliados y Eje con el mapa del terreno donde se desenvolverá la batalla para que se preparen para combatir. Además les informa sobre la ubicación inicial de la bandera y crea el agente de servicios.

- Recibir información de cada combatiente de su posición, velocidad, orientación, salud y número de municiones. Con estos datos se encargará de conducir la partida.
- Informar a los agentes tropa qué hay en su campo de visión: paquetes de medicina, soldados, etc.
- Calcular las intersecciones de los disparos con las víctimas, e informarles a estas en caso de ser heridas.
- Informar a los agentes tropa si han recibido algún paquete y qué tipo de paquete es.
- Levantar el servidor HTTP para aceptar conexiones de los clientes de visualización. Enviarle mensajes a estos con información de la bandera, cada uno de los agentes tropa y paquetes para que se pueda visualizar la partida.
- Acabar la partida e informar el equipo ganador. Puede ser debido a que la salud de todos los combatientes del equipo Aliados sea nula, a que se venció el tiempo de la partida o a que la bandera se llevó a la base del equipo Aliados.

4.2.2. Agente Service

Este agente tiene la función de conocer e informar los servicios que brindan los combatientes. Cada vez que se crea un agente tropa, registra el servicio que ofrece informándoselo a este agente. Al morir un combatiente en la partida, debe informárselo a este agente para que sepa que ya no puede ofrecer el servicio que antes brindaba. Sabiendo quiénes son los agentes tropa activos, puede responder a peticiones de solicitud de servicios. De esta manera, un soldado puede averiguar quiénes son los operadores de campo, médicos y otros soldados de su equipo para demandar su ayuda.

4.2.3. Agente Pack

Los agentes Pack son los paquetes que se crean durante la partida, y pueden ser de tres tipos:

- ObjectivePack:
  Este agente se crea al inicio del juego por el agente Manager y representa la bandera.

- MedicPack:
  Son creados por los médicos. Su función es incrementar la salud de quien lo reciba. Se auto destruyen pasado un tiempo si no son recogidos por nadie.

- AmmoPack:
  Son creados por los operadores de campo. Su función es incrementar el número de municiones de quien lo reciba. Se auto destruyen pasado un tiempo si no son recogidos por nadie.

4.2.4. Agente BDITroop

Estos son agentes híbridos que incorporan la arquitectura deliberativa desarrollada. Su comportamiento reactivo incluye a bajo nivel acciones de traslación, generar puntos de control y disparar entre otras. Además, cada agente tropa puede solicitarle información al agente Service para conocer los médicos, operadores de campo y soldados de su equipo disponibles. La parte deliberativa de los agentes puede ser programada desde un nivel más alto en ASL, y hará uso de información generada en las capas reactivas. Existen hasta el momento tres tipos de agentes tropa, cada uno con acciones específicas del rol que desempeñan en la batalla:

- BDIMedic:
  Son los médicos del equipo. Dan servicio médico, que consiste en crear paquetes de medicina.

- BDIFieldOp:
  Son los operadores de campo del equipo. Brindan el servicio de recargar municiones, creando paquetes de municiones para ello.

- BDISoldier:
  Son los soldados del equipo. Ofrecen servicio de refuerzo. Este servicio es sencillamente asistir a un compañero del equipo, dirigiéndose a su posición para reforzar el ataque.

Teniendo en cuenta las funciones de cada uno de los agentes, se presenta en la Figura 4.1 la comunicación entre ellos en pGomas. Todos los agentes son creados en SPADE, haciendo uso de la mensajería instantánea y demás ventajas de estas.

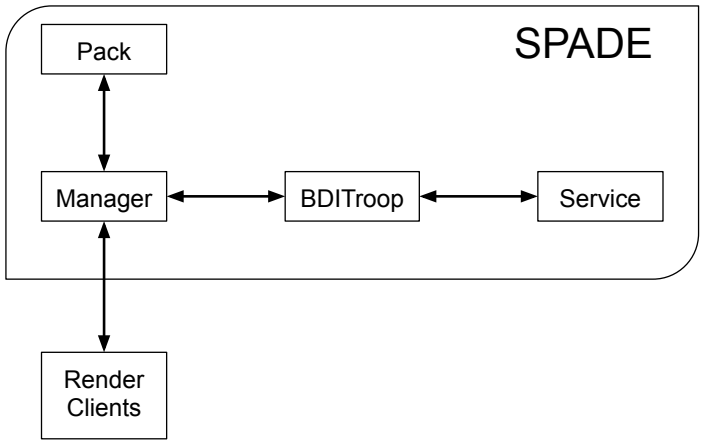


Figura 4.1: Comunicación entre agentes en pGomas.

4.3. Clases en pGomas

Para desarrollar el juego, se han implementado las clases mostradas en la Figura 4.2 (salvo Agent y BDIAgent). A continuación se pormenorizan las funcionalidades de ellas, salvo las que se encargan de crear los agentes del juego, que ya se describieron en la Sección 4.2.

- Agent:
  Es la clase base de agente proporcionada por SPADE.

- BDIAgent:
  Es la clase base de agente SPADE híbrido con comportamiento deliberativo como se describe en el Capítulo 3.

- AbstractPlayer:
  Clase dependiente del juego desarrollado, que añade atributos como equipo del agente, nombre (JID) y nombre del agente de servicios. Incorpora comportamientos de registrar y desregistrar servicios.

- MicroPlayer:
  Clase usada por el agente Manager para poder coordinar la partida. Contiene atributos de cada agente tropa de la batalla, específicamente el nombre, equipo, tipo de agente, posición, velocidad, orientación, salud, cantidad de municiones y si lleva o no la bandera.

- DinObject:
  Clase usada por el agente Manager para representar los paquetes. Contiene atributos de los paquetes como la posición, el nombre, equipo, tipo de paquete, si ha sido tomado o no y quién lo ha tomado en caso afirmativo.

- Sight:
  Usada por el Manager para saber qué paquete (dígase DinObject) o tropa (dígase MicroPlayer) se encuentra en el campo de visión de qué agente tropa. A los agentes tropa les proveerá información como qué tipo de agente es, de qué equipo es, su salud, su posición y la distancia a la que se encuentra de él.

- Server:
  Pone en funcionamiento el servidor HTTP que es lanzado por el Manager para que se conecten los visores gráficos.

- Threshold:
  Contiene atributos umbrales de los agentes tropa, como máximo número de disparos por ráfaga, de municiones y de salud.

- TerrainMap:
  Se encarga de cargar el mapa del campo de batalla. Realiza el *parsing* de los ficheros de texto que describen al terreno del campo.

- Vector3D:
  Clase que da soporte vectorial a pGomas. Contiene métodos de creación de vectores así como suma, resta, normalización, módulo y productos escalares y vectoriales de estos.

- Mobile:
  Hace uso de los vectores Vector3D para crear algunos de los atributos de movimiento de los agentes tropa: posición, velocidad, orientación y destino. Brinda métodos útiles como cálculo de nuevas posiciones, velocidades y orientaciones.

- AAlgorithm:
  Implementa un algoritmo A usado por los agentes tropa para desplazarse por el campo de batalla. La heurística se puede cambiar por el usuario, pero se recomienda usar la distancia Euclidiana dado que se permiten desplazamientos diagonales.

- Config:
  Contiene información de configuración del juego. Entre ellas la ruta de los mapas por defecto, ontologías, performativas y valores de precisión.

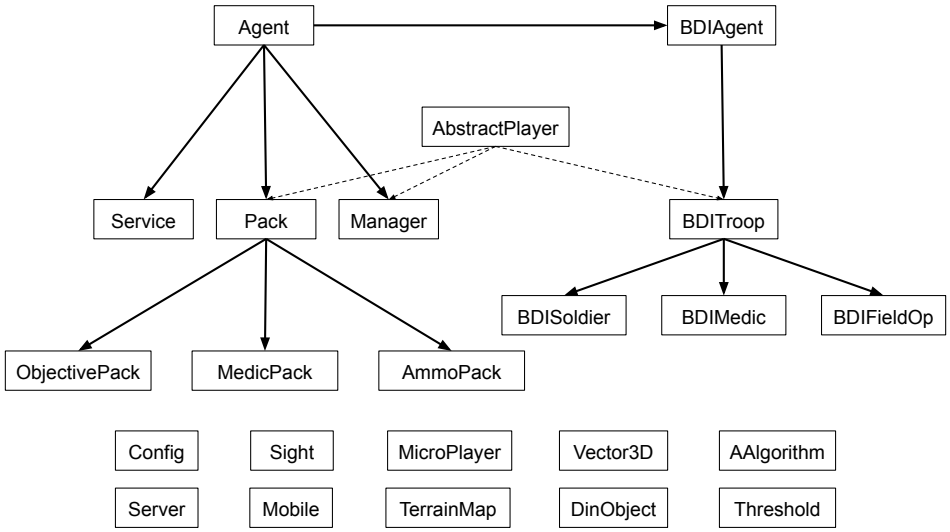


Figura 4.2: Diagrama de clases en pGomas.

4.4. Visualización de la partida

Como se mencionó al inicio del capítulo, la partida se puede visualizar desde clientes de visores gráficos que se conectan a la aplicación. Para esto, el agente Manager crea un servidor HTTP que aceptará conexiones de cada uno de los clientes a través de las cuales se enviará información acerca del estado actual de la partida. Esta información consiste en indicarle el número de agentes que hay en el campo de batalla, dígase tropas o paquetes. Para cada agente tropa, se enviará la siguiente información:

- Nombre del agente tropa (JID del agente).
- Tipo de agente tropa (soldado, médico u operador de campo).
- Equipo al que pertenece.
- Salud.

- Número de municiones que lleva.
- Si lleva la bandera o no.
- Vectores de posición, velocidad y orientación.

Para cada paquete, salvo la bandera en caso de que esté tomada, se le comunicará a cada visor:

- Nombre del paquete (JID del agente).
- Tipo de paquete (bandera o paquete de medicina o munición).
- Vector de posición.

Hasta el momento se han desarrollado 3 visores gráficos:

1. Un visor en consola, como se muestra en la Figura 4.3, usando símbolos para diferenciar el tipo de tropa o paquete, como "+" para los médicos y "A" para los paquetes de municiones. Para el campo de batalla, los espacios en blanco son posiciones donde no hay obstáculos y "*" donde sí. También se pinta en color el fondo de los píxeles que ocupa el símbolo así como la base de cada equipo, en azul para el equipo Eje y en rojo para el otro. La salud y número de municiones de cada agente tropa así como su nombre son escritos en la misma consola fuera del mapa.
2. Un visor en consola mejorado usando Pygame, mostrado en la Figura 4.4. En este se puede ver además el campo de visión de cada agente. El contenido gráfico, aunque sigue siendo en 2D, tiene mayor calidad.
3. Un visor gráfico desarrollado con el motor gráfico Unity, con gráficos en 3D mejorados y más realista como se ve en la Figura 4.5. Permite ver la partida desde distintos puntos de vista, ya que la cámara se puede mover.

Alternativamente, se puede hacer una visualización *offline* de la partida. Para esto se ha desarrollado un cliente que recibe la información de la partida enviada por el agente Manager y los vuelca en un fichero. Posteriormente este fichero se puede cargar y visualizar la batalla con el segundo visor gráfico comentado en esta sección.

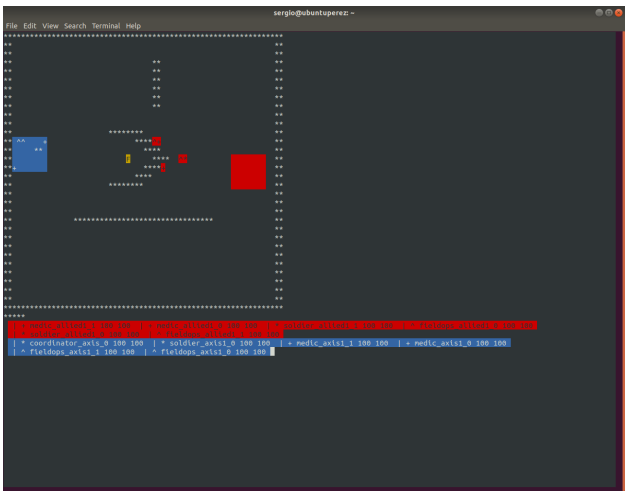


Figura 4.3: Visor de texto en consola.

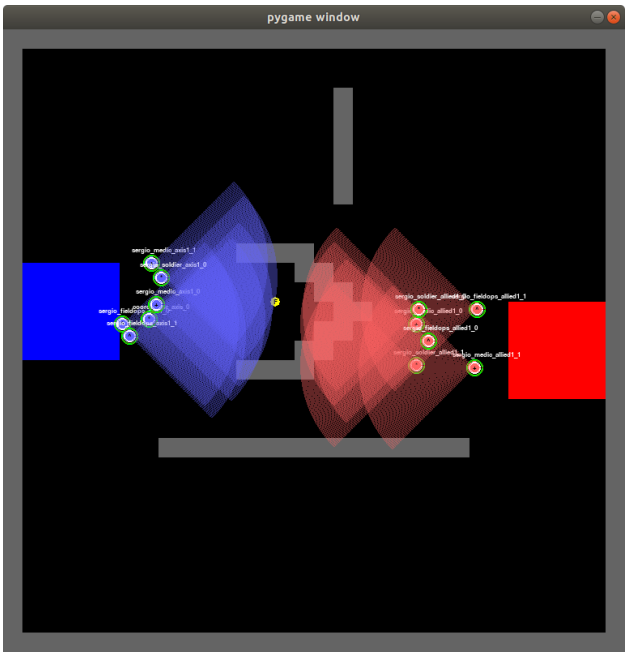


Figura 4.4: Visor en consola usando Pygame.

Figura 4.5: Visor gráfico desarrollado en Unity.

4.5. Campo de batalla

La creación de mapas de combate es muy sencilla. El campo de batalla se puede especificar mediante dos ficheros ".txt" que contengan información del mismo. Uno de ellos debe precisar las dimensiones del terreno y la ubicación de la bandera y las bases de ambos equipos. El otro es el mapa de obstáculos, que se representan con asteriscos, como se muestra en la Figura 4.6.

Para desplazarse por el campo de batalla, los agentes tropa tienen un comportamiento reactivo periódico asociado. Cada agente tropa tiene una cola de destinaciones, que contiene todas las casillas del mapa a las que debe visitar para llegar a un destino. Como los campos de batalla tienen obstáculos, se ha decidido usar un algoritmo A para generar las casillas intermedias por las que deben transitar.

Cada casilla del mapa se visitará teniendo en cuenta la velocidad y orientación del agente. Esto es sumamente importante para los visores gráficos 3D ya que no es deseable que caminen en una dirección orientándose hacia otra. Además, para propósitos del juego también, pues al cambiar la orientación del agente cambia su campo de visión.

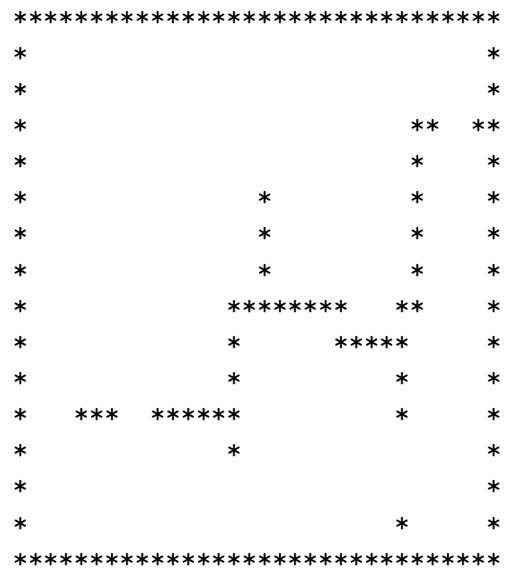


Figura 4.6: Mapa de obstáculos de un terreno de 16×32.

4.6. Acciones en pGomas

Como se comentó en la Sección 3.1.1, Python-agentspeak permite crear acciones que pueden ser llamadas desde el entorno deliberativo. Se han implementado varias acciones para los agentes tropa que serán necesarias para el funcionamiento del juego. Entre ellas:

- `.goto([X,Y,Z])`:
  Establecer las coordenadas [X,Y,Z] destino del agente. Pone al agente tropa en marcha hacia dicho lugar, usando un algoritmo A para desplazarse por el terreno.

- `.stop`:
  Detener el movimiento del agente tropa en el punto en que se encuentra.

- `.turn(R)`:
  Modificar la orientación del agente tropa una cantidad (positiva o negativa) R de radianes. Útil para alterar el campo de visión.

- `.look_at([X,Y,Z])`:
  Orientar el agente tropa hacia una posición [X,Y,Z] del campo de batalla.

- `.create_control_points([X,Y,Z],D,N,C)`:
  Crear un grupo de N puntos aleatorios de control a una distancia D dada de una ubicación [X,Y,Z] en el mapa. La lista de puntos se almacena en la última variable especificada. Estos puntos sirven, por ejemplo, para patrullar alrededor de la bandera.

- `.shoot(N,[X,Y,Z])`:
  Disparar un número de N disparos a una ubicación [X,Y,Z] dada.

- `.register_service("servicio_a")`:
  Enviar un mensaje al agente de servicios para registrar un servicio especificado.

- `.get_medics`:
  Enviar un mensaje al agente de servicios solicitando los médicos de su equipo.

- `.get_fieldops`:
  Enviar un mensaje al agente de servicios solicitando los operadores de campo de su equipo.

- `.get_backups`:
  Enviar un mensaje al agente de servicios solicitando los soldados de su equipo.

- `.get_service("servicio_a")`:
  Enviar un mensaje al agente de servicios solicitando otro servicio (distinto de los tres anteriores) a los agentes tropa de su equipo que lo ofrezcan.

Además de estas acciones generales para los agentes tropa, se han desarrollado las siguientes:

- `.cure`:
  Crear paquetes de medicina. Solo los médicos pueden realizar esta acción.

- `.reload`:
  Crear paquetes de munición. Solo los operadores de campo pueden realizar esta acción.

- `.reinforce([X,Y,Z])`:
  Acudir a una posición [X,Y,Z] para brindar refuerzo. Solo los soldados pueden ejecutar esta acción.

Cualquier nuevo rango de agente tropa que se desee crear, puede heredar estas acciones, e implementar nuevas. Así por ejemplo, se pueden implementar acciones que calculen distancias entre dos coordenadas del mapa, o que ordenen listas de puntos a patrullar, entre otras.

4.7. Implantación

En **https://github.com/sfp932705/pygomas** se encuentra disponible el código fuente de la nueva versión del juego "Captura La Bandera" que se ha desarrollado. Para instalarlo, se puede descargar el código, y ejecutar `python setup.py install` en el directorio raíz.

Alternativamente, para facilitar el proceso de instalación del juego de una manera sencilla se ha creado un instalador usando *PyPi*. Si se desea, se puede instalar con *pip3*, ejecutando `pip3 install pygomas`.

Para lanzar una partida se debe lanzar primero el Manager, especificándole los parámetros obligatorios como JID, JID del agente de servicios, mapa del campo de batalla y número de agentes tropa en la partida. Un ejemplo sería el siguiente:

```
pygomas manager -j cmanager@gtirouter.dsic.upv.es -m map_01
                -sj cservice@gtirouter.dsic.upv.es -np 6
```

Una vez echo lo anterior, se puede determinar un fichero JSON que contenga los agentes tropa que lucharán. En este fichero se debe especificar quiénes son los agentes Manager y Service para que los combatientes puedan comunicarse con ellos. Además se deben establecer los agentes tropa que se crearán. Para cada agente tropa será necesario definir el equipo, el nombre, la contraseña y el rango (soldado, médico, operador de campo u otro rango que el usuario implemente). También se puede especificar la cantidad de agentes de ese tipo que se desean crear así como el fichero ASL donde se encuentran los planes, deseos e intenciones. La ayuda del juego brinda un fichero JSON para que sirva de ejemplo. Una posible ejecución sería:

```
pygomas run -g troops.json
```

Si se desea visualizar la partida, luego de crear el Manager se pueden lanzar los clientes visores. En el proyecto de pGomas están disponibles los dos primeros descritos en la Sección 4.4. Se pueden ejecutar, respectivamente, de la siguiente manera:

```
pygomas render --text
```

```
pygomas render
```

El tercero, se puede descargar desde la página de JGOMAS, disponible en **http://www.gti-ia.upv.es/sma/tools/jgomas/downloads.php**.

Para la visualización *offline* que se comentó, se debe primero volcar en un fichero la información de la partida que brinda el Manager. Esto se haría así:

```
pygomas dump --log partida.log
```

Para visualizar este fichero, se puede ejecutar la siguiente línea:

```
pygomas replay --game partida.log
```

Capítulo 5

Validación

En este capítulo se describen una serie de experimentos que se realizaron para validar la versión desarrollada del juego "Captura La Bandera". Se presentan imágenes y enlaces a vídeos del visor gráfico desarrollado en Pygame de algunos experimentos.

5.1. Desplazamiento por el campo de batalla

Lo primero que se validó fue el desplazamiento de los agentes tropa. Con esto se pudo comprobar algunas de las acciones descritas en la Sección 4.6, como `.goto`,`.stop` y `.turn`. En la Figura 5.1 se muestra un campo de batalla tipo laberinto y en **https://youtu.be/xOIoXBgZXHg** hay un vídeo disponible donde se puede ver el desplazamiento por el campo de batalla de un agente tropa.

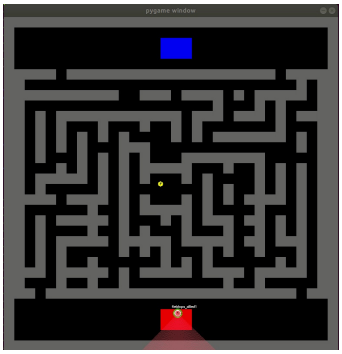


Figura 5.1: Desplazamiento usando el algoritmo A implementado.

5.2. Campo de visión

Se realizaron pruebas para validar que los agentes tropa detectaban los paquetes y otros agentes tropa (amigos o enemigos) que estuviesen en su campo de visión. En **https://youtu.be/eq3XAu5IO1Y** se puede ver uno de los casos donde se evaluó una estrategia de seguir al líder. En este escenario hay tres agentes tropa: un soldado, un médico y un operador de campo. Al iniciar la partida, el médico y el operador de campo son enviados a dos puntos distintos del terreno, y al llegar deben mantenerse explorando, girando su orientación mediante la acción `.turn` implementada. Harán esto hasta que encuentren un soldado de su equipo, momento a partir del cual lo seguirán. La simplicidad del código ASL del médico viene dada por la semántica de alto nivel que se puede emplear. Se presenta a continuación:

```
+flag(Position)
  <-
  .goto(100,0,130).

+heading(H): exploring
  <-
  .wait(2000);
  .turn(0.785).

+target_reached(Position)
  <-
  +following;
  +exploring;
  .turn(0.785).

+friends_in_fov(ID,Type,Angle,Distance,Health,Position): Type == 1 & following
  <-
  -exploring;
  .goto(Position).
```

La creencia `flag(Position)` tiene los valores de las coordenadas de la bandera, y se añade al inicio de la partida. `heading(H)` guarda las componentes del vector de orientación del agente, y cada vez que cambia su orientación se actualiza. La creencia `target_reached(Position)` se añade al llegar a un destino. `friends_in_fov(ID,Type,Angle,Distance,Health,Position)` establece por cada agente tropa del mismo equipo que esté en su campo de

visión el identificador (un número), el tipo de tropa, el ángulo y la distancia a la que se encuentra así como su salud y coordenadas en el mapa.

El código del operador de campo solo difiere en la posición inicial a la que es enviado. El soldado primero va a un punto intermedio entre los otros dos combatientes, y una vez llegado se dirigirá hacia la bandera. El código se muestra a continuación:

```
+flag(Position)
  <-
  .wait(30000);
  +gather;
  .goto(90,0,120).

+target_reached(Position): gather
  <-
  -gather;
  ?flag(F);
  .goto(F).
```

5.3. Solicitar servicios

La siguiente prueba es una estrategia de patrulla alrededor de un punto del terreno. Está disponible en **https://youtu.be/Q0xbD7U5DK8** y corrobora el correcto funcionamiento de las acciones `.create_control_points`, `.get_medics` y `.get_fieldops`. En este ejemplo, el soldado solicitará al agente de servicios los agentes de su equipo que brindan servicio médico y servicio de recarga de municiones. Esto se realiza con las acciones `.get_medics` y `.get_fieldops`, que añaden respectivamente las creencias `myMedics` y `myFieldops`. Teniendo el nombre de estos, el soldado le puede mandar mensajes solicitando su ayuda. Posteriormente crea 5 puntos de control a un radio de 25 unidades de la bandera. Las coordenadas de los puntos estratégicos creados con `.create_control_points` se añaden como una creencia (`control_points`). El agente irá de uno en uno, y cuando haya recorrido dos veces la lista, les enviará un mensaje a sus compañeros.

El código ASL del soldado usado fue el siguiente:

```
+flag (Position)
  <-
  .get_medics;
  .get_fieldops;
```

```
  .create_control_points([X,Y,Z],25,C);
  +control_points(C).

+control_points(C)
  <-
  .length(C,L);
  +total_control_points(L);
  +patrolling;
  +loop(0);
  +patroll_point(0).

+target_reached(Position): patrolling
  <-
  ?patroll_point(P);
  -+patroll_point(P+1);
  -target_reached(Position).

+patroll_point(P): total_control_points(T) & P<T
  <-
  ?control_points(C);
  .nth(P,C,A);
  .goto(A).

+patroll_point(P): total_control_points(T) & P==T
  <-
  ?loop(L);
  -+loop(L+1);
  -patroll_point(P);
  +patroll_point(0).

+loop(L): L == 2
  <-
  ?myMedics(All_medics);
  .nth(0,All_medics,M);
  ?position(Position);
  .send(M, tell,cure(Position));
  ?myFieldops(All_fieldops);
  .nth(0,All_fieldops,F);
  .send(F, tell,need_ammo(Position)).
```

El médico está en espera de que le pidan ayuda. Irá hacia la ubicación donde se necesita su servicio y creará paquetes de medicina usando la acción `.cure` implementada. Una vez hecho esto regresará a la base. El código en ASL se muestra a continuación:

```
+cure(Position)
  <-
  .print("Going to give medic packs to friend at: ",Position);
  +curing;
  .goto(Position).

+target_reached(Position):curing
  <-
  .print("In ASL, Medic cured at :",Position);
  .cure;
  -curing;
  ?base(B);
  .goto(B).
```

Para el operador de campo se hace lo mismo, solo cambiando el predicado de algunas creencias y la acción de crear paquetes de medicina por la de crear paquetes de municiones. El soldado no toma los paquetes de medicina ni los de municiones puesto que tiene la salud y el número de municiones al máximo posible.

5.4. Disparos

Para comprobar las acciones de disparo se desarrolló una partida donde hay un agente tropa del equipo Aliado y tres agentes tropa del Eje. Como se puede ver en **https://youtu.be/qhjypwy8eJE**, los disparos van reduciendo la salud de la víctima y disminuyendo el número de municiones de los que disparan (son los anillos en colores que se pintan alrededor de los agentes.) Para el equipo Eje, a los agentes tropa se les ordena ir hacia la bandera una vez que conozcan la ubicación de esta. Si encuentran algún enemigo en el camino, lo cual sucede al añadirse la creencia `enemies_in_fov(ID,Type,Angle,Distance,Health,Position)`, le dispararán. El código en ASL es el mostrado a continuación:

```
+flag (Position)
  <-
  .goto(Position).
```

```
+enemies_in_fov(ID,Type,Angle,Distance,Health,Position)
  <-
  .shoot(3,Position).
```

Para el soldado Aliado, como se muestra a continuación, se le ordena ir a por la bandera y regresar a la base cuando la capture. De manera análoga, debe disparar a enemigos que se encuentre por el camino. Como está más cerca de la bandera, mediante la acción `.wait`, se le da una espera al inicio, para forzar que se encuentre con los agentes tropa enemigos en el camino. Además se puede ver cómo toma la bandera y luego la pierde.

```
+flag (Position)
  <-
  .wait(4500);
  .goto(Position).

+flag_taken
  <-
  .print("TEAM_ALLIED flag_taken");
  ?base(Position);
  .goto(Position).

+enemies_in_fov(ID,Type,Angle,Distance,Health,Position)
  <-
  .shoot(3,Position).
```

Con estos experimentos se pudo comprobar el funcionamiento del sistema desarrollado. Las acciones implementadas se ejecutan satisfactoriamente, así como la inyección y consulta de conocimiento que se refleja en la adición de creencias y planes.

5.5. Pruebas de escalabilidad

En esta sección se comentan las pruebas de escalabilidad que se realizaron para medir el rendimiento del sistema.

5.5.1. Pruebas centralizadas

En la versión actual de JGOMAS, se pueden ejecutar partidas de 16 agentes tropa en total como máximo. Esto limita a los estudiantes a poder

desarrollar estrategias militares en grupos numerosos, ya que solo disponen de 8 agentes tropa por equipo. Sin embargo, con la versión desarrollada, pGomas, se pueden lanzar partidas con un mayor número de combatientes. Hay un vídeo disponible en **https://youtu.be/Tart7sje_jk** donde se muestra una partida con 24 agentes tropa (12 por equipo.). Se han programado los agentes tropa del equipo Eje para que creen puntos de control alrededor de la bandera y la protejan. A los Aliados se les ordena ir a por la bandera y regresar una vez que la tengan. Las estrategias usadas por parte de ambos equipos son sencillas, y los agentes no coordinan entre sí refuerzos de combatientes ni recargas de municiones o paquetes de medicina. Cuando se queden sin municiones, no podrán disparar más. Como se puede apreciar, la partida se desarrolla tal y como se esperaba.

Mediante los clientes visores se puede comprobar que la aplicación funciona correctamente con este número de agentes. Además, en los terminales donde se lanzan el agente Manager y los agentes tropa se puede verificar en el log su correcto funcionamiento. El intercambio de mensajes entre el Manager y los agentes tropa no pierde la sincronía. El Manager es capaz de recibir y procesar los mensajes de todos los agentes tropa, así cómo enviarle respuestas a cada uno y cumplir el resto de sus funciones.

Se siguió aumentando el número de combatientes en la partida. Se muestra una ejemplo en **https://youtu.be/oEKPYJL29hM** con 60 agentes tropa. La única diferencia en la estrategia usada esta vez es que el agente tropa que capture la bandera se retirará a la base haciendo giros, para poder variar el campo de visión y disparar a cualquier enemigo que se encuentre por el medio. La visualización de la partida muestra que el sistema propuesto puede soportar este número de agentes.

Posteriormente se aumentó el número de agentes, hasta partidas de 84, 126 y 144 agentes. Estas partidas se pueden ver en los siguientes enlaces:

https://youtu.be/8utdAWD-1Is
https://youtu.be/PcCnT2x0EB8
https://youtu.be/Yt3jooQQwWg

Esto supone un incremento de un orden de magnitud en cuanto al número de agentes por partida con respecto a la versión actual. Todas estas pruebas fueron realizadas en un mismo ordenador con las siguientes características:

```
Procesador: Intel Core i7-7700 CPU @ 3.60GHz x 8
Memoria: 16 GB
Disco: SSD 240 GB
Sistema Operativo: Ubuntu 18.04 (64-bit)
```

Como se puede apreciar en los vídeos, el visor gráfico no es capaz de

renderizar a 30 cuadros por segundo la partida. Esto se debe al exceso de información que debe procesar. No obstante, para concluir que la partida se está ejecutando correctamente, no es necesario visualizarla, pues se pueden ver los logs en las consolas. El agente Manager es el cuello de botella de la partida, ya que debe recibir mensajes de cada uno de los combatientes. Si se acumulan mensajes en su buzón y no es capaz de procesarlos y responderlos en tiempo real se muestra un error por pantalla para detener la partida. Teniendo en cuenta esto, se desarrollaron batallas con 210, 294, 378 y 420 agentes sin visualización gráfica. No hubo error en ningún caso, pero las partidas sí tardaron mucho más que antes, como se muestra en la Figura 5.2, donde se han lanzado partidas en el mismo campo de batalla y con la misma estrategia . La mayor demora está al principio, pues al ir desde la base a la bandera (se encuentran un poco distantes), cada agente debe calcular un algoritmo A. Al estar lanzados tantos agentes en un mismo proceso, se ralentiza la búsqueda de puntos intermedios para llegar al destino. También existe una demora a la hora de procesar los mensajes que llegan del Manager periódicamente indicándoles sus coordenadas y qué está en su campo de visión.

# de agentes total	# de agentes por proceso	Duración (s)
420	60	612
378	54	487
294	42	415
210	30	285
126	18	183
84	12	139

Figura 5.2: Comparativa del tiempo de duración de partidas variando el número de agentes.

5.5.2. Pruebas distribuidas

Para evitar esta demora en las partidas, se hicieron pruebas distribuidas, usando máquinas virtuales. Se crearon 39 máquinas virtuales, cada una con las siguientes características:

```
Procesador: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
Memoria: 1.8 GB
Disco: 48 GB
Sistema Operativo: CentOS Linux 7 (Core) (64-bit)
```

En una máquina virtual se lanzó el Manager, y en otras se fueron creando agentes tropa, con un máximo de 12 por cada una. Como muestran los resultados que se presentan en la Figura 5.3, el sistema distribuido con máquinas virtuales presenta tiempos de duración menor para casos donde se lancen pocos agentes por máquina.

# de agentes total	# de agentes por máquina	Duración (s)
456	12	947
380	10	687
304	8	325
228	6	269
152	4	176
128	4	146
76	2	118

Figura 5.3: Comparativa del tiempo de duración de partidas de manera distribuida con máquinas virtuales.

En uno de los experimentos se usaron 32 máquinas virtuales, y se crearon 4 agentes tropa en cada una. Esta partida tuvo 128 agentes y una duración de 146 segundos. En cambio, en los experimentos de la Sección 5.5.1 una partida de 126 agentes tardó 183 segundos. Al incrementar el número de agentes tropa por máquina virtual, el sistema distribuido tardó más.

Es importante destacar que estas máquinas virtuales son mucho menos potentes que la usada en los experimentos de la Sección 5.5.1. Si se deseara aumentar el número de agentes y disminuir el tiempo de las partidas, mejorando el rendimiento del sistema, se pudiera usar una aproximación distribuida pero con máquinas físicas.

5.6. Estrategias más avanzadas

En esta sección se detallan dos ejemplos de mayor complejidad que hacen uso de la potencialidad que brinda la arquitectura híbrida desarrollada en SPADE e integrada en la versión de pGomas.

En el primero el equipo de los Aliados tiene la misma estrategia que hemos visto: ir a por la bandera y una vez tomada, regresar a la base. El equipo del Eje, se alinea junto a la bandera, y se orienta a esta. En ASL, se lograría de la siguiente forma:

```
+flag(F)
  <-
  .get_service("axis");
  +never_shot;
  .randint(-20, 20, R);
  .nth(0,F,X);
  .nth(1,F,Y);
  .nth(2,F,Z);
  .goto([X-R,Y,Z+R]).
```

La acción `.randint(a,b,R)` se implementó para devolver un número entero aleatorio R tal que a $\leq$ R $\leq$ b. Se consultan todos los agentes tropa de su equipo mediante la acción `.get_service("axis")` para poder enviarles mensajes. Una vez que llegan a la bandera, se orientan a esta.

```
+target_reached(T): never_shot
  <-
  ?flag(F);
  .look_at(F).
```

Al recibir un disparo, explorarán su alrededor para detectar quién ha sido y dónde está. Cuando encuentren al enemigo, su posición será transmitida al resto del equipo.

```
+health(H): H < 100 & never_shot
 <-
 .stop;
 +exploring;
 .turn(1.57).

+heading(H): exploring
  <-
  .turn(1.57);
  .wait(100).

+enemies_in_fov(ID,Type,Angle,Distance,Health,Position): never_shot
  <-
  -exploring;
  -never_shot;
  ?axis(A);
  .send(A,tell,enemy_at(Position));
```

```
  .look_at(Position);
  .shoot(1,Position).

+enemy_at(P)
<-
 .look_at(P).
```

No obstante, al recibir el mensaje con la posición del enemigo, los receptores no eliminan su creencia "exploring", pues podrían venir más de otros lugares y necesitan estar al tanto. Cuando un agente tropa necesite paquetes de municiones o de medicina, mandará un mensaje pidiendo ayuda. Los médicos y los operadores de campo crearan los paquetes de medicina y recarga de municiones respectivamente. El rol de cada agente tropa se puede diferenciar por la creencia "class(C)", siendo 2 para los médicos y 4 para los operadores de campo. Una vez mandado el mensaje, comenzará a explorar para ver si encuentra el paquete que necesita. Los paquetes tienen un atributo tipo que lo identifican, siendo 1001 para los de medicina y 1002 para los de recarga de municiones. Cuando tenga alguno en su campo de visión, irá a por él.

```
+health(H): H <= 80
 <-
  +need_cure;
  !help_plan.

+ammo(A): A <= 10
 <-
  +need_ammo;
  !help_plan.

+!help_plan
 <-
  ?axis(Axis);
  .send(Axis,tell,need_help);
  +need_help;
  +exploring;
  .turn(1.57).

+need_help
 <-
```

```
 .wait(1000);
 !action.

+!action: class(C) & C == 4
 <-
  .reload.

+!action: class(C) & C == 2
 <-
  .cure.

+packs_in_fov(ID,Type,Angle,Distance,Health,Position): Type == 1002 & need_ammo
  <-
  -need_ammo;
  -exploring;
  .goto(Position).

+packs_in_fov(ID,Type,Angle,Distance,Health,Position): Type == 1001 & need_cure
  <-
  -need_cure;
  -exploring;
  .goto(Position).
```

Finalmente, para seguir disparando a un enemigo y dejar de explorar, debe tener cierta cantidad de municiones.

```
+enemies_in_fov(ID,Type,Angle,Distance,Health,Position): ammo(A) & A > 10
  <-
  -exploring;
  .look_at(Position);
  .shoot(1,Position).

+enemies_in_fov(ID,Type,Angle,Distance,Health,Position): ammo(A) & A <= 10
  <-
  +exploring;
  .shoot(1,Position).
```

En **https://youtu.be/W6OEEWqU4gw** hay un vídeo con una partida donde cada equipo tiene un soldado, un médico y un operador de campo. En este caso, la victoria es perfecta, el equipo Eje no sufre ni

una pérdida y la bandera nunca es tomada. Se han usado solo tres tropas por equipo para que se pueda apreciar cómo crean los paquetes y van por ellos, ya que con más combatientes no se vería bien debido a los conos de campos de visión que se pintan. No obstante, la estrategia es válida para un número mayor de combatientes por equipo, como se muestra en **https://youtu.be/MYXZhMUYP0k**. En este caso, hay 6 agentes tropa por equipo. Aunque la bandera es tomada una vez, el equipo Eje logra eliminar al enemigo sin perder ningún combatiente.

En el segundo ejemplo, se usó una estrategia más elaborada para el equipo de los Aliados. Cuando el Manager les notifique la bandera, irán a puntos aleatorios a una distancia de esta. Pero un de ellos, un médico, se quedará esperando 15 segundos, para llegar más tarde y sorprender. Al igual que en el ejemplo anterior, los operadores de campo crearán paquetes de recarga de municiones cuando ellos o sus compañeros lo necesiten. Esta vez, solo los médicos podrán tomar la bandera y huir con esta. El objetivo de los operadores de campo y los soldados será mantener abierto el fuego a los enemigos, para que algún médico tome la bandera y escape con ella. Cuando un médico tenga la bandera, mantendrá actualizada la posición de la misma para avisarle a sus compañeros en caso de perderla. La estrategia usada para el equipo Eje es la misma que en el ejemplo anterior.

En los siguientes enlaces hay dos vídeos donde gana el equipo Aliado usando esta estrategia.

https://youtu.be/ndbHSmnVg9Y
https://youtu.be/yJBlZ4wnSjc

Con estos dos ejemplos se puede ver que desarrollar nuevas formas de ataque y defensa no supone un reto complicado.

Capítulo 6

Conclusiones y trabajos futuros

6.1. Conclusiones

En este proyecto se ha propuesto, implementado e integrado en SPADE una nueva arquitectura de agente híbrido con comportamiento deliberativo basado en BDI y comportamientos reactivos. Esta arquitectura ha beneficiado a la plataforma enormemente, brindándole la posibilidad de ser utilizada en proyectos donde antes no era posible. Además, la programación de los agentes es ahora posible mediante el empleo de abstracciones de más alto nivel como planes, objetivos y creencias. Esto posibilita la creación de agentes inteligentes sin tener que preocuparse por funciones de bajo nivel que se pueden delegar en la capa reactiva de los mismos.

Se han mostrado ejemplos para corroborar su funcionamiento y se ha simplificado el proceso de instalación a una línea de comando para viabilizar su uso a futuros usuarios.

Adicionalmente, se ha usado esta arquitectura implementada en un proyecto complejo que comprueba su eficacia: pGomas, una nueva versión del juego "Captura La Bandera". El sistema desarrollado permite lanzar partidas con un número mucho mayor de agentes de lo que se podía hasta el momento en JGOMAS, lo cual se ha corroborado mediante pruebas de escalabilidad realizadas. Además, se han presentado ejemplos de planes en ASL con estrategias sencillas y otras más complejas así como enlaces a vídeos para que se puedan ver algunas partidas. Gracias a SPADE y el nuevo modelo de agente híbrido se pueden desarrollar estrategias de coordinación, control y ataque con mayor facilidad de lo que era antes.

6.2. Recomendaciones y trabajos futuros

En aras de profundizar en esta línea de investigación, se propone dotar al nuevo modelo de agente híbrido SPADE de una serie de mejoras. Por ejemplo, otras ilocuciones que sí están disponibles en Jason, entre ellas *tellHow*, para permitir el intercambio de planes entre agentes. Además, sería interesante implementar más acciones internas que puedan brindarle una mayor funcionalidad a los agentes. Para manejar listas u otras estructuras de datos sería llamativo poder contar con acciones del estilo `.delete`, como las de Jason, que permitan borrar elementos de una lista. El objetivo sería tener una gama de acciones que permitan a estos agentes poder ser usados en una amplia variedad de aplicaciones sin que los usuarios tengan la necesidad de implementarlas ellos mismos, lo que haría que el proyecto resultara más atractivo para ellos.

Con este trabajo, se dispone de un sistema multiagente en el cual se pueden desarrollar partidas. Hasta el momento, el comportamiento de los agentes ha sido, aunque desde un nivel de abstracción elevado, programado por un usuario. Se propone usar este sistema para desarrollar, aplicar y dar soporte a técnicas de aprendizaje por refuerzo y aprendizaje multiagente para que los agentes tropa aprendan a coordinarse y cooperar entre sí. Con esto se podrían desarrollar estrategias de combate complejas que garanticen la victoria en la mayoría de los casos.

Bibliografía

[1] Wooldridge, Michael y Nicholas R Jennings: *Agent theories, architectures, and languages: a survey.* En *International Workshop on Agent Theories, Architectures, and Languages*, páginas 1–39. Springer, 1994. (Citado en la página 1).

[2] Brooks, Rodney A: *Intelligence without representation.* Artificial intelligence, 47(1-3):139–159, 1991. (Citado en la página 1).

[3] Gregori, Miguel Escrivá, Javier Palanca Cámara y Gustavo Aranda Bada: *A jabber-based multi-agent system platform.* En *Proceedings of the fifth international joint conference on Autonomous agents and multi-agent systems*, páginas 1282–1284. ACM, 2006. (Citado en la página 2).

[4] Barella, Antonio, Soledad Valero y Carlos Carrascosa: *JGOMAS: New approach to AI teaching.* IEEE Transactions on education, 52(2):228–235, 2008. (Citado en la página 2).

[5] Bellifemine, Fabio, Agostino Poggi y Giovanni Rimassa: *JADE–A FIPA-compliant agent framework.* En *Proceedings of PAAM*, volumen 99, página 33. London, 1999. (Citado en las páginas 2 y 5).

[6] ALEMANY IBOR, SERGIO: *Diseño e implementación de un simulador basado en agentes estilo JGOMAS en Python.*, 2018. (Citado en la página 2).

[7] Kravari, Kalliopi y Nick Bassiliades: *A survey of agent platforms.* Journal of Artificial Societies and Social Simulation, 18(1):11, 2015. (Citado en la página 5).

[8] Howden, Nick, Ralph Rönnquist, Andrew Hodgson y Andrew Lucas: *JACK intelligent agents-summary of an agent infrastructure.* En *5th International conference on autonomous agents*, 2001. (Citado en la página 5).

[9] Oey, Michel, Sander van Splunter, Elth Ogston, Martijn Warnier y Frances MT Brazier: *A framework for developing agent-based distributed applications.* En *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volumen 2, páginas 470–474. IEEE, 2010. (Citado en la página 5).

[10] Luke, Sean, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan y Gabriel Balan: *Mason: A multiagent simulation environment.* Simulation, 81(7):517–527, 2005. (Citado en la página 6).

[11] Palanca, Javier, Andrés Terrasa, Carlos Carrascosa y Vicente Julián: *Improving the programming skills of students in multiagent systems master courses.* Computer Applications in Engineering Education. (Citado en la página 6).

[12] *SPADE's documentation.* `https://spade-mas.readthedocs.io/en/latest/`. Accessed: 2019-5-29. (Citado en la página 6).

[13] Jennings, NR, EH Mamdani, I Laresgoiti, J Perez y J Corera: *GRATE: a general framework for co-operative problem solving.* Intelligent Systems Engineering, 1(2):102–114, 1992. (Citado en la página 10).

[14] Rao, Anand S, Michael P Georgeff *y cols.*: *BDI agents: from theory to practice.* En *ICMAS*, volumen 95, páginas 312–319, 1995. (Citado en las páginas 10 y 11).

[15] Bratman, Michael E, David J Israel y Martha E Pollack: *Plans and resource-bounded practical reasoning.* Computational intelligence, 4(3):349–355, 1988. (Citado en la página 10).

[16] MÜLLER, JÖRG P.: *Architectures and applications of intelligent agents: A survey.* The Knowledge Engineering Review, 13(4):353–380, 1999. (Citado en la página 11).

[17] Georgeff, Michael P y Amy L Lansky: *Reactive reasoning and planning.* En *AAAI*, volumen 87, páginas 677–682, 1987. (Citado en la página 11).

[18] Georgeff, Michael y Felix Ingrand: *Decision-making in an embedded reasoning system.* En *International Joint Conference on Artificial Intelligence*, 1989. (Citado en la página 11).

[19] Georgeff, M y A Rao: *Intelligent real-time network management.* En *Proc. 10th Int. Workshop on Expert Systems and their Applications*, páginas 87–101, 1990. (Citado en la página 12).

[20] Georgeff, Michael Peter y Anand S Rao: *A profile of the australian artificial intelligence institute.* IEEE Intelligent Systems, (6):89–92, 1996. (Citado en la página 12).

[21] Ljungberg, Magnus y Andrew Lucas: *The OASIS air traffic management system.* 1992. (Citado en la página 12).

[22] Ingrand, François Felix, Michael P Georgeff y Anand S Rao: *An architecture for real-time reasoning and system control.* IEEE expert, 7(6):34–44, 1992. (Citado en la página 12).

[23] Murray, Graeme, Serena Steuart, Dino Appla, David McIlroy, Clinton Heinze, Martin Cross, Arvind Chandran, Richard Raszka, Gil Tidhar, Anand Rao *y cols.*: *The challenge of whole air mission modelling.* Dsfö-<S6-ooT7, página 143, 1995. (Citado en la página 12).

[24] Moin, Amir H: *Sense-deliberate-act cognitive agents for sense-compute-control applications in the internet of things and services.* En *International Internet of Things Summit*, páginas 23–28. Springer, 2014. (Citado en la página 12).

[25] Zhang, Wei, Weifeng Shi y Jinbao Zhuo: *BDI-agent-based quantum-behaved PSO for shipboard power system reconfiguration.* International Journal of Computer Applications in Technology, 55(1):4–11, 2017. (Citado en la página 12).

[26] Bosse, Tibor, Zulfiqar A Memon y Jan Treur: *A recursive BDI agent model for theory of mind and its applications.* Applied Artificial Intelligence, 25(1):1–44, 2011. (Citado en la página 12).

[27] Masinde, Muthoni y Antoine Bagula: *ITIKI: bridge between African indigenous knowledge and modern science of drought prediction.* Knowledge Management for Development Journal, 7(3):274–290, 2011. (Citado en la página 12).

[28] Shoham, Yoav: *Agent-oriented programming.* Artificial intelligence, 60(1):51–92, 1993. (Citado en la página 13).

[29] Georgeff, Michael P y Amy L Lansky: *Procedural knowledge.* Proceedings of the IEEE, 74(10):1383–1398, 1986. (Citado en la página 13).

[30] Burmeister, Birgit y Kurt Sundermeyer: *Cooperative problem-solving guided by intentions and perception.* ACM SIGOIS Bulletin, 13(3):10, 1992. (Citado en la página 13).

[31] Nunes, Ingrid, CJPD Lucena y Michael Luck: *BDI4JADE: a BDI layer on top of JADE*. En *Proc. of the Workshop on Programming Multiagent Systems*, páginas 88–103, 2011. (Citado en la página 13).

[32] Braubach, Lars, Winfried Lamersdorf y Alexander Pokahr: *Jadex: Implementing a BDI-Infrastructure for JADE Agents*, 2003. (Citado en la página 13).

[33] Rao, Anand S: *AgentSpeak (L): BDI agents speak out in a logical computable language*. En *European workshop on modelling autonomous agents in a multi-agent world*, páginas 42–55. Springer, 1996. (Citado en la página 13).

[34] Bordini, Rafael H y Jomi F Hübner: *BDI agent programming in AgentSpeak using Jason*. En *International Workshop on Computational Logic in Multi-Agent Systems*, páginas 143–164. Springer, 2005. (Citado en la página 13).

[35] *A Python-based interpreter for the agent-oriented programming language JASON*. `https://github.com/niklasf/python-agentspeak`. Accessed: 2019-6-14. (Citado en la página 15).

[36] Ahlbrecht, Tobias, Jürgen Dix y Niklas Fiekas: *Scalable multi-agent simulation based on MapReduce*. En *Multi-Agent Systems and Agreement Technologies*, páginas 364–371. Springer, 2016. (Citado en la página 15).

[37] Iocchi, Luca, Daniele Nardi y Massimiliano Salerno: *Reactivity and deliberation: a survey on multi-robot systems*. En *Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, páginas 9–32. Springer, 2000. (Citado en la página 16).

[38] Hannebauer, Markus, Jan Wendler y Enrico Pagello: *Balancing Reactivity and Social Deliberation in Multi-Agent Systems—A Short Guide to the Contributions*. En *Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, páginas 3–8. Springer, 2000. (Citado en la página 16).