VNIVERSITAT DE VALÈNCIA

DEPARTAMENTO DE INFORMÁTICA

MÁSTER EN COMPUTACIÓN AVANZADA Y SISTEMAS INTELIGENTES

INTENTIONAL LEARNING: PERFORMANCE STUDIES ON COMMITMENT

REPORTE FINAL DE MÁSTER

PRESENTADO POR:
D. CARLOS ALBERTO GONZÁLEZ ALARCÓN

DIRIGIDO POR
DR. D. FRANCISCO GRIMALDO MORENO

VALENCIA, JULIO DE 2012

Este trabajo es financiado por la beca 214787, otorgada por el Consejo Nacional de Ciencia y Tecnología para los estudios del Doctorado en Tecnologías de la Información, Comunicaciones y Matemática Computacional, en la Universidad de Valencia. Número de becario, 273098.

Resumen

JILDT *(Jason Induction of Logical Decison Trees)* es una librería que proporciona una clase de agente aprendiz intencional para Jason, la bien conocida implementación en Java de *AgentSpeak(L)*. Los agentes definidos como instancia de esta clase son capaces de aprender sobre las razones para adoptar intenciones de acuerdo a su propia experiencia. A partir de esta clase de agente es posible definir agentes aprendices con comportamientos particulares, como lo son los agentes SingleMinded, que implementan una estrategia de compromiso racional y son capaces de aprender sus propias políticas de abandono.

La inducción de árboles lógicos de decisión es un mecanismo idóneo para sustentar aprendizaje en el contexto de agentes Intencionales BDI, principalmente gracias a que las entradas requeridas para este método son fácilmente obtenidas del estado mental de los agentes; además, cada recorrido del nodo raíz a un nodo hoja corresponde a una conjunción de literales de primer orden, es decir, el tipo de representación necesaria para definir el contexto de un plan. Estos árboles pueden expresar hipótesis sobre las ejecuciones exitosas o fallidas de una intención.

Se implementó el algoritmo de aprendizaje en dos niveles de programación: un nivel basado en Java, pensado para mejorar el desempeño computacional de agentes que aprenden sin interacción social; y un nivel basado en *AgentSpeak(L)* que abre la puerta a algunas formas particulares de aprendizaje social. La representación en primer orden de las entradas del algoritmo de aprendizaje mostrada en este trabajo, mejora el desempeño computacional de los procesos de aprendizaje, en comparación con versiones preliminares de JILDT. Los resultados experimentales sobre racionalidad son muy prometedores. Se observa que agentes definidos como agentes aprendices y aprendices con estrategia de compromiso racional son adaptables: eran confiados con respecto a un plan, y luego adoptaron una estrategia cautelosa después de haber tenido problemas con la ejecución de éste. El uso de Aprendizaje Intencional proporciona la convergencia con el nivel adecuado de confianza-cautela basado en la experiencia de los agentes. Los agentes adoptan una actitud confiada hacia los planes de éxito, y una actitud cautelosa hacia los planes que fallan.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1
Introducción

El estudio de la Inteligencia Artificial tiene, entre sus múltiples propósitos, el estudio de la comprensión y construcción de entidades inteligentes, a diferencia de otras disciplinas como la filosofía o la psicología cuyo objeto de estudio se basa sólo en resaltar la comprensión de estas entidades. Es por ello que la construcción de agentes racionales constituye el curiosamente llamado nuevo enfoque de la Inteligencia Artificial, definido en el texto introductorio de Russell & Norvig (2003).

Hoy en día, el estudio de los sistemas multiagentes comprende muchas áreas de investigación importantes, como simulación social, auto organización, planeación y robótica colectiva, entre otras; sin embargo, el aprendizaje, en un contexto intencional de agencia racional ha recibido poca atención, aún cuando éste cuenta con presupuestos filosóficos sólidos (Dennett, 1987; Bratman, 1987; Searle, 1962).

Haciendo frente a esta problemática, se presenta a JILDT (Jason Induction of Logical Decison Trees), una librería que provee un mecanismo de aprendizaje basado en la inducción de árboles lógicos de decisión, para Jason, la bien conocida implementación en Java de *AgentSpeak(L)* (Bordini et al., 2007).

1.1. Marco Teórico

Este trabajo de investigación hace énfasis en el estudio del Aprendizaje Intencional en sistemas multiagente, y se desenvuelve dentro de tres áreas de investigación en el estudio de la Inteligencia Artificial: Agencia BDI, Aprendizaje Intencional y Aprendizaje Lógico Inductivo.

1.1.1. Agentes BDI: AgentSpeak(L) y Jason

El modelo de agencia racional BDI *(Belief-Desire-Intention)* ha resultado ampliamente relevante dentro del contexto de la Inteligencia Artificial, particularmente dentro del estudio de sistemas multiagente. Esto debido a que el modelo cuenta con sólidos presupuestos filosóficos, basados en la postura Intencional de Dennett (1987), la teoría de planes, intenciones y razonamiento práctico de Bratman (1987) y la comunicación con actos de habla de Searle (1962).

Estas tres nociones de Intencionalidad (Lyons, 1995) proveen las herramientas necesarias para describir los agentes a un nivel adecuado de abstracción, al adoptar la postura intencional y definir funcionalmente a estos agentes de manera compatible con tal postura, como sistemas de razonamiento práctico.

La primera idea que necesitamos para entender el modelo de agencia BDI, es la idea de que podemos hablar sobre programas computacionales como si éstos tuvieran un estado mental. Así, cuando hablamos de un sistema BDI, estamos hablando de programas con analogías computacionales de creencias, deseos e intenciones.

AgentSpeak(L) es un lenguaje de programación abstracto basado en una lógica restringida de primer orden con eventos y acciones, y representa un marco de trabajo elegante para programar agentes BDI. Por su parte, *Jason* es un intérprete programado en Java, el cual es fiel al lenguaje *AgentSpeak(L)*. En la sección 2.4.1 se presenta la sintáxis de *AgentSpeak(L)* y Jason, mientras que su semántica operacional se puede observar en el apéndice A.

1.1.2. Aprendizaje Intencional

Como se ha mencionado en el texto introductorio, a pesar de que el modelo racional de agencia BDI se puede explicar basándose en sus términos filosóficos de Intencionalidad, razonamiento práctico y actos de habla, poco se ha hecho para sustentar el aprendizaje Intencional en sistemas multiagente. El **aprendizaje** puede ser visto como una actualización en el comportamiento, habilidades o conocimiento en general con la finalidad de mejorar el desempeño.

El estudio de la **Intencionalidad** tiene su origen en las discusiones filosóficas medievales sobre la diferencia entre la existencia natural de las cosas y la existencia mental o intencional de las cosas, o *esse intentionale*, que deriva del latín *intentio* y significa dirigir la atención del pensamiento hacia algo, o simplemente apuntar hacia un objetivo, o *ser acerca de* (Lyons, 1995). Muchos de nuestros estados mentales están en cierto sentido dirigidos a objetos o asuntos del mundo. Si tengo una creencia, debe ser una creencia que tal y tal es el caso; si deseo algo

debe ser el deseo de hacer algo, o que algo ocurra; si tengo una intención, debe ser la intención de hacer algo; etc. Es esta característica de direccionalidad en nuestros estados mentales, lo que muchos filósofos han etiquetado como Intencionalidad (Searle, 1979).

Michael Bratman (1987) se encargó de construir los fundamentos filosóficos que tratan de modelar la racionalidad de aquellas acciones tomadas por los seres humanos en determinadas circunstancias. Esto tiene sus orígenes en una tradición filosófica que busca comprender lo que llamamos **razonamiento práctico**, esto es, el razonamiento dirigido hacia las acciones: hacia el proceso de decidir qué hacer; a diferencia del razonamiento teórico que está dirigido hacia las creencias.

En este trabajo, el término **Aprendizaje Intencional** está fuertemente ligado a la teoría de racionalidad práctica de Bratman (1987), donde los planes están predefinidos y los objetivos del proceso de aprendizaje son las razones para adoptar intenciones.

1.1.3. Aprendizaje Lógico Inductivo basado en interpretaciones

A diferencia de la mayoría de los trabajos de aprendizaje que utilizan un formalismo *atributo-valor*, la programación lógica inductiva está constituida de relaciones entre objetos, las cuales son un conjunto de tuplas de constantes. A cada una de estas relaciones, sumada a un conocimiento general *(background)*, se le conoce como una **interpretación**. Cada interpretación es independiente de las otras.

Gracias a la estructura en primer orden de los formalismos del modelo BDI de un agente, la Programación Lógica Inductiva resulta ser una buena opción para procesar los ejemplos necesarios que permitan sustentar el aprendizaje, mientras que un árbol lógico de decisión facilita obtener una hipótesis a partir de sus ramas (Guerra-Hernández et al., 2004b).

TILDE, o Top-Dow Induction of Logical Decision Trees, es un algoritmo desarrollado en la Universidad de Leuven, en Bélgica por Hendrik Blockeel et al. (1999). TILDE genera árboles lógicos de decisión, de los cuales sus nodos son conjunciones de primer orden, a diferencia de los árboles inducidos por ID3 o C4.5 (Quinlan, 1986), donde los nodos son atributos. La sección 3.3.2 introduce la morfología de los árboles lógicos de decisión, mientras que en la sección 4.1 se explica como se construyen.

1.1.4. Caso de estudio: Estrategias de compromiso

Las estrategias de compromiso en agentes racionales servirán como caso de estudio para comprobar la adaptabilidad de un agente a su entorno a través de la inducción de árboles lógicos de decisión como mecanismo de aprendizaje. Las estrategias de compromiso permiten conocer a un agente bajo qué circunstancias debe abandonar o no una intención que haya sido previamente adoptada. En este caso, el objetivo del aprendizaje no es únicamente conocer las razones para adoptar intenciones, sino también conocer las razones para abandonarlas. Tres estrategias de compromiso son bien conocidas en la literatura de los agentes racionales (Rao & Georgeff, 1991): los compromisos ciego, racional, y emocional, los cuales se describen en la sección 2.3.

1.2. Antecedentes y planteamiento del problema

JILDT es un proyecto que se inició en el Departamento de Inteligencia Artificial de la UniversidadVeracruzana, en México, por González-Alarcón (2010) y Guerra-Hernández et al. (2010a,b). En un comienzo, se implementó una clase de agente capaz de aprender las razones para adoptar un plan cada vez que haya fallado en la ejecución del mismo. En su primera aproximacións, se hacia uso del sistema ACE/TILDE (Blockeel & De Raedt, 1998) como un comando externo para construir los árboles lógicos de decisión necesarios para aprender el nuevo contexto. Ortiz-Hernández (2007) describe formalmente el mecanismo necesario para ejecutar el aprendizaje Intencional con base en las interpretaciones que el agente podía formar a partir de la percepción de su entorno. La figura 1.1 ilustra como era inducido el aprendizaje en la primera aproximación de JILDT.

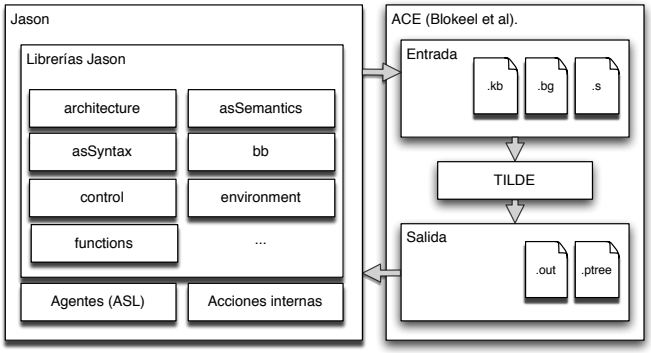


Figura 1.1 Modo operacional del Aprendizaje Intencional en JILDT, en un primera aproximación.

Primero se obtiene el estado BDI del agente y se generan los ejemplos a través de acciones internas de Jason. Cuando el agente falla en la ejecución de un plan, se lanza el proceso de aprendizaje, generando los archivos de configuración necesarios para ejecutar ACE/TILDE, el cual es ejecutado como un comando externo. Finalmente, se leen los archivos generados por el algoritmo de inducción, y se modifica el contexto del plan fallido.

Si bien es cierto que se podía sustentar el aprendizaje a través del sistema ACE/TILDE, ésta no era una solución multiplataforma viable, ya que el sistema ACE/TILDE se ejecuta únicamente en sistemas operativos UNIX. Es por ello que surge la necesidad de desarrollar el algoritmo de inducción como parte de las acciones internas que el intérprete Jason pueda utilizar (ver la figura 1.2). Al ser programado en Java y bajo una licencia GNU LGPL, este algoritmo de inducción podía ser manipulado de acuerdo al problema que se esté intentando resolver, cosa que no se puede hacer en su implementación en ACE/TILDE, ya que éste no se considera Software Libre. Considerando la problemática anterior, se implementó, además del algoritmo de inducción, una biblioteca de acciones internas necesarias para formar ejemplos de entrenamiento y generar el conjunto de archivos de entrada necesarios para la inducción, y de esta manera se pueda obtener el conocimiento adquirido para modificar los planes de agentes definidos como instancias de la clase de agente aprendiz.

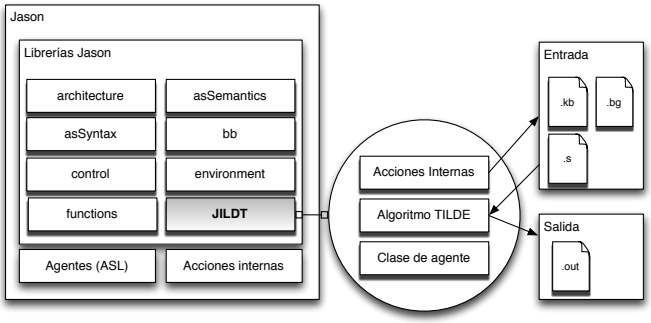


Figura 1.2 Propuesta de integración de JILDT a Jason *AgentSpeak(L)*.

A pesar de los buenos resultados obtenidos (González-Alarcón, 2010; Guerra-Hernández et al., 2010a,b), JILDT aún presenta algunas carencias. La principal de éstas, es la falta de un protocolo de aprendizaje social, el cual constituye el problema a resolver en la fase doctoral del proyecto. Por ello, en esta nueva versión se extiende la librería, dirigiéndola hacia una librería de aprendizaje social, por lo que primero se busca mejorar dos aspectos importantes: la representación de las entradas del algoritmo de aprendizaje, y la implementación del algoritmo de aprendizaje en un nivel *AgentSpeak(L)* (ver la figura 1.3).

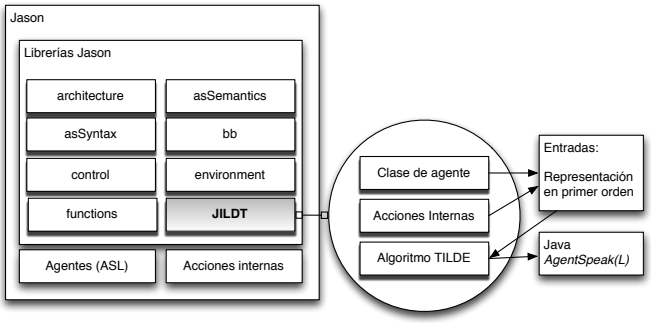


Figura 1.3 Propuesta de mejora de implementación de JILDT.

En la versión de JILDT al momento de iniciar este trabajo[1], las entradas para ejecutar el algoritmo de aprendizaje eran tres ficheros externos que almacenan los ejemplos (llamados modelos), las configuraciones (que incluyen el sesgo del lenguaje) y el conocimiento general del agente; el hecho de que sean ficheros externos que deban generarse cada vez que se ejecuta el proceso de aprendizaje, supone un mayor consumo de recursos, tanto en tiempo de ejecución, como en espacio en memoria RAM y Disco Duro; por otro lado, implementar el algoritmo de inducción de árboles lógicos de decisión a un nivel *AgentSpeak(L)*, presume mayor flexibilidad en el proceso de aprendizaje. En el capítulo 4 se presenta el análisis y diseño para enfrentar la problemática anterior, mientras que en el capítulo 5 se describe la implementación de la primera versión liberada de JILDT.

1.3. Trabajos relacionados

Existen algunos trabajos que se relacionan con este proyecto de investigación, pero que muestran variaciones significativas. Subagdja et al. (2009) propone una arquitectura para Aprendizaje Intencional, sin embargo, su uso del término Aprendizaje Intencional es ligeramente diferente. En su investigación, el uso del aprendizaje era el objetivo de los agentes BDI en lugar de un resultado imprevisto, esto quiere decir que sus agentes saben como aprender, pero no ejecutan este mecanismo para aprender sobre sus fallos. Nuestro uso del término **Aprendizaje Intencional** está estrictamente circunscrito a la teoría de la racionalidad práctica (Bratman, 1987) donde los planes están predefinidos y los objetivos de los procesos de aprendizaje son las razones para adoptar o abandonar las intenciones.

[1] En adelante, nos referimos a esta versión como la versión preliminar de JILDT, para distinguirla de la versión reportada en este proyecto.

Nowaczyk & Malec (2007) presentan un objetivo similar donde los agentes pueden ver el aprendizaje de la función de selección para los planes aplicables. La principal diferencia con nuestro trabajo es que ellos proponen una solución *ad hoc* para un determinado agente no BDI.

Singh et al. (2010, 2011) presentan un trabajo más parecido a esta investigación, el cual ofrece un *framework* de aprendizaje desarrollado en la plataforma JACK (Busetta et al., 1999). La tarea de aprendizaje consiste en la selección del plan que debe ejecutarse con el fin de mejorar un evento-meta dado, tomando en cuenta los datos de la ejecución previa y el estado actual del mundo. El mecanismo que utilizan es inducción de árboles de decisión, pero usando un enfoque proposicional, empleando el algoritmo J48 de Weka (Witten & Frank, 2000), mientras que en este trabajo se inducen árboles lógicos de decisión, lo que nos permite explotar la representación en primer orden del estado mental de los agentes BDI. Además, a diferencia de JILDT, el trabajo de Singh et al. (2010) no provee una semántica operacional para manejar fallas y recolectar ejemplos de entrenamiento.

1.4. Motivación

Implementar una librería que cuente con un mecanismo de aprendizaje intencional integrado al conjunto de acciones internas de Jason permitirá definir agentes adaptables a su entorno. Para ello, es necesaria una representación en primer orden de las entradas del algoritmo de aprendizaje que permita reducir el consumo de recursos, homogeneizando estas entradas con las creencias que forman parte del conocimiento del agente. Por otra parte, implementar un agente que cuente con una estrategia de compromiso racional (ver sección 2.3), le da un nuevo uso al contexto aprendido: aprender cuándo es racional abandonar una intención. Finalmente, implementar un mecanismo de aprendizaje en un nivel *AgentSpeak(L)* abre las puertas a una libreria de aprendizaje social.

1.5. Objetivos

El objetivo principal de este proyecto es extender la librería de aprendizaje JILDT, implementando el mecanismo de aprendizaje a un nivel *AgentSpeak(L)*. Para conseguir este objetivo se buscará conseguir los siguientes objetivos:

- Mejorar la representación de las entradas del algoritmo de aprendizaje definiéndolas como literales de primer orden.

- Diseñar una base de creencias para almacenar los ejemplos de entrenamiento, y las creencias relacionadas con el proceso de aprendizaje.
- Implementar las funciones necesarias para ejecutar la inducción de árboles lógicos de decisión a un nivel de programación *AgentSpeak(L)*.
- Implementar una clase de agente que ejecute intencionalmente un plan de aprendizaje cuando sea necesario y redefina sus planes originales *(Learner)*.
- Dado que usaremos la estrategia de compromiso *single-minded*, se implementará una clase de agente que defina esta estrategia de compromiso, el cual puede abandonar una acción, toda vez que crea que no será capaz de finalizarla *(SingleMindedLearner)*.
- Publicar en la Web una primera distribución liberada de la librería.

1.6. Exposición y método

El documento está dividido en tres partes. La primera parte (capítulos 2-3) presenta el **estado del arte** en el que se desenvuelve este trabajo de investigación. El capítulo 2 introduce los conceptos de agencia BDI, Intencionalidad, estrategias de compromiso, donde estas últimas son caso de estudio en esta investigación, como se mencionó anteriormente. Además se introduce el lenguaje de programación orientada a agentes *AgentSpeak(L)* y Jason, su intérprete basado en Java. El capítulo 3 describe el aprendizaje como un mecanismo de inducción lógica, en el cual se describen algoritmos de inducción de árboles de decisión en su enfoque proposicional (ID3, C4.5), árboles lógicos de decisión (*First Order Logical Decision Trees* (FOLDT)) y un enfoque de representación de información basado en interpretaciones.

En la segunda parte (capítulos 4-7) se presenta el desarrollo del trabajo. El capítulo 4 muestra un análisis de la problemática y como se ha diseñado la librerá para enfrentarla. En el capítulo 5, se explica a detalle el diseño y el *modus operandi* de JILDT, mientras que en el capítulo 6 se presentan algunos experimentos realizados para probar la librería. Una discusión sobre este trabajo es presentada en el capítulo 7, que incluye las conclusiones y una breve descripción de los trabajos futuros.

Finalmente, en la tercera parte se añade a manera de **apéndice**, la semántica operacional de *AgentSpeak(L)* y Jason; además del código de los agentes definidos para realizar los experimentos de la sección 6.1.

Parte I

Estado del arte

Capítulo 2
Agencia

Antes de presentar los detalles de implementación y los resultados experimentales obtenidos en esta investigación, es necesario introducir algunos conceptos básicos que permitan ofrecer un panorama general del estado de arte en el que ésta se encuentra. Se empieza introduciendo el término **agente**, que como algunos otros términos relacionados con la **Inteligencia Artificial (IA)**, no cuenta con una definición precisa sobre sí mismo. Sin embargo, existe una gran cantidad de conceptualizaciones, que si bien resultan muy generales en algunos casos, son aceptadas por los investigadores en el área. La primera sección de este capítulo conceptualiza el término agente dentro de lo que Wooldridge & Jennings (1995) definen como una noción débil de agencia, en torno a la autonomía, la iniciativa y la sociabilidad. Esta primera aproximación es suficiente para caracterizar los atributos ineludibles en el comportamiento de un agente, mismos que nos permiten diferenciar lo que es un agente, de lo que no lo es.

Una noción más fuerte de agencia es presentada en la segunda sección, en la que se introduce el modelo de agencia BDI *(Beliefs-Desires-Intentions)* de agencia, el cual define agentes con **estados mentales**, que cuentan con presupuestos filosóficos sólidos, basados en la postura intencional de Dennett (1987) , la teoría de planes, intenciones y razonamiento práctico de Bratman (1987) y los actos de habla de Searle (1962). La tercera sección describe los tipos de compromiso que un agente BDI puede adoptar, introducidos por Rao & Georgeff (1991), los cuales son caso de estudio de esta investigación.

Por último, en la cuarta sección se presenta el paradigma de programación empleado en esta investigación, la **Programación Orientada a Agentes (POA)**, introduciendo el lenguaje *AgentSpeak(L)*, un lenguaje abstracto BDI ampliamente utilizado y Jason su intérprete basado en Java.

2.1. Agentes racionales

Históricamente, el término **agente**, ha sido empleado bajo dos acepciones: Primero, desde la época de los griegos clásicos y hasta nuestros días, los filósofos usan el término agente para referirse a una entidad que actúa con un propósito dentro de un contexto social. Segundo, la noción legal de agente, como la persona que actúa en beneficio de otra con un propósito específico, bajo la delegación limitada de autoridad y responsabilidad, estaba ya presente en el derecho Romano y ha sido ampliamente utilizada en economía (Muller-Freienfels, 1999). Franklin & Graesser (1997) argumentan que todas las definiciones del término agente en el contexto de la IA, se basan en alguna de estas dos acepciones históricas.

Dentro del contexto computacional (Wooldridge, 2002), el concepto de agente responde a las necesidades de cinco características actuales en los ambientes computacionales: ubicuidad, interconexión, inteligencia, delegación y orientación humana[1]. Esto es, en entornos donde existe una diversidad de dispositivos de cómputo distribuidos e interconectados en nuestro entorno, los agentes inteligentes emergen como la herramienta para delegar tareas adecuadamente y abordar esta problemática desde una perspectiva más familiar para usuarios, programadores y diseñadores. Una definición consensual de agente (Wooldridge & Jennings, 1995; Russell & Norvig, 2003) es la siguiente:

Definición 1 (Agente) *Un agente es un sistema computacional capaz de actuar de manera autónoma para satisfacer sus objetivos y metas, mientras se encuentra situado persistentemente en su medio ambiente.*

Aunque puede parecer demasiado general, esta definición provee una abstracción del concepto de agente basada en su presencia e interacción con el medio ambiente (ver la figura 2.1). Russell & Subramanian (1995) encuentran que esta abstracción presenta al menos tres ventajas:

1. Permite observar las facultades cognitivas de los agentes al servicio de encontrar cómo hacer lo correcto.
2. Permite considerar diferentes tipos de agente, incluyendo aquellos que no se supone tengan tales facultades cognitivas.
3. Permite considerar diferentes especificaciones sobre los subsistemas que componen los agentes.

[1] Por orientación humana, queremos referirnos a la tendencia de dirigir la programación desde un punto de vista orientado a máquinas hacia conceptos y metáforas que reflejen la forma en que el ser humano entiende el mundo.

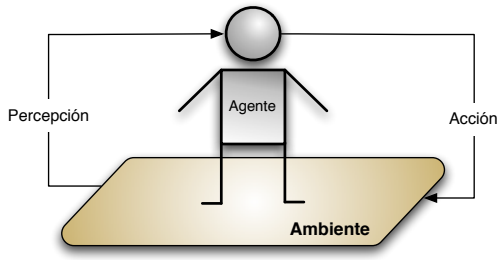


Figura 2.1 Abstracción de un agente a partir de su interacción con el medio ambiente.

En términos generales, un **agente racional** es aquel que hace lo correcto. Esto es, para cada secuencia de percepciones, un agente es capaz de realizar una acción que le permita llevar a cabo una tarea, obteniendo el mejor resultado, maximizando una medida de desempeño previamente definida. Russell & Norvig (2003) menciona que la racionalidad de un agente en un tiempo dado depende de cuatro factores.

1. La **medida de rendimiento** que define el criterio de éxito, y permite cuantificar el nivel de confiabilidad.
2. La **secuencia de percepciones** del agente, esto es, todo lo que el agente ha percibido en un tiempo dado.
3. El **conocimiento** del agente sobre el medio ambiente en el que está situado.
4. La **habilidad** del agente, es decir, las acciones que el agente puede llevar a cabo con cierta destreza.

Es necesario precisar que dado que la racionalidad de un agente se define en relación con el éxito esperado, con base en lo que el agente ha percibido, no podemos exigir a un agente que tome en cuenta lo que no puede percibir, o haga lo que sus actuadores no pueden hacer. Considerando los puntos anteriores, Russell & Norvig (2003) definen a un agente racional de la siguiente manera:

Definición 2 (Agente racional) *En cada posible secuencia de percepciones, un agente racional es aquel capaz de comprender aquella acción que supuestamente maximice su medida de rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones y en el conocimiento del medio que el agente mantiene almacenado .*

2.1.1. Comportamiento flexible y autónomo

Foner (1993) argumenta que para ser percibido como inteligente, un agente debe exhibir cierto comportamiento, caracterizado más tarde por Wooldridge & Jennings (1995), como **comportamiento flexible y autónomo**. Este tipo de comportamiento se caracteriza por su:

- **Reactividad.** Los agentes inteligentes deben ser capaces de percibir su medio ambiente y responder a tiempo a los cambios en él, a través de sus acciones.
- **Iniciativa.** Los agentes inteligentes deben exhibir un comportamiento orientado por sus metas, tomando la iniciativa para satisfacer sus objetivos de diseño (*pro-activeness*).
- **Sociabilidad.** Los agentes inteligentes deben ser capaces de interaccionar con otros agentes, posiblemente tan complejos como los seres humanos, con miras a la satisfacción de sus objetivos.

Una caracterización más detallada de autonomía es presentada por Covrigaru & Lindsay (1991). Su desiderata, que incluye algunos de los aspectos ya mencionados, expresa que un agente se percibe como autónomo en la medida en que:

1. Su comportamiento esté orientado por sus metas y sea capaz de seleccionar que meta va a procesar a cada instante.
2. Su existencia se dé en un periodo relativamente mayor al necesario para satisfacer sus metas.
3. Sea lo suficientemente robusto como para seguir siendo viable a pesar de los cambios en el ambiente.
4. Pueda interactuar con su ambiente en la modalidad de procesamiento de información.
5. Sea capaz de exhibir una variedad de respuestas, incluyendo movimientos de adaptación fluidos; y su atención a los estímulos sea selectiva.
6. Ninguna de sus funciones, acciones o decisiones, esté totalmente gobernada por un agente externo.
7. Una vez en operación, el agente no necesite ser programado nuevamente por un agente externo.

El argumento central aquí es que ser autónomo, no sólo depende de la habilidad para seleccionar metas u objetivos de entre un conjunto de ellos, ni de la habilidad de formularse nuevas metas, sino de tener el tipo adecuado de metas. Los agentes artificiales son usualmente diseñados para llevar a cabo tareas por nosotros, de forma que debemos comunicarles que es lo que esperamos que hagan. En un sistema computacional tradicional ésto se reduce a escribir el programa adecuado y ejecutarlo. Un agente puede ser instruido sobre qué hacer usando un programa, con la ventaja colateral de que su comportamiento estará libre de incertidumbre; pero atentando

contra su autonomía, teniendo como efecto colateral la incapacidad del agente para enfrentar situaciones imprevistas mientras ejecuta su programa. Las metas, las funciones de utilidad, y los mecanismos de aprendizaje son maneras de indicarle a un agente qué hacer, sin decirle cómo hacerlo.

2.2. Agencia BDI

El modelo de **agencia intencional BDI** ***(Belief-Desire-Intention)*** ha resultado de gran relevancia dentro del estudio de **Sistemas Multiagente (SMA)**. Esto obedece a que el modelo cuenta con sólidos presupuestos filosóficos, basados en la postura intencional de Dennett (1987), la teoría de planes, intenciones y razonamiento práctico de Bratman (1987) y la comunicación con actos de habla de Searle (1962). Estas tres nociones de **Intencionalidad** (Lyons, 1995) proveen las herramientas para describir a los agentes en un nivel adecuado de abstracción, al adoptar la postura intencional y definir funcionalmente a estos agentes de manera compatible con tal postura, como sistemas de razonamiento práctico. Estas formas de Intencionalidad han sido formalmente expresadas y estudiadas bajo diferentes lógicas multimodales de elegante semántica (Rao, 1996; Singh, 1995; Wooldridge, 2000).

La primera idea que necesitamos para entender el modelo BDI, es la idea de que podemos hablar sobre programas computacionales como si éstos tuvieran un **estado mental**. Así, cuando hablamos de un sistema BDI, estamos hablando de programas con analogías computacionales de creencias, deseos e intenciones. En términos generales, el modelo BDI consiste en un conjunto de creencias, deseos e intenciones, y sus propiedades y relaciones, las cuales se definen informalmente del siguiente modo:

- **Creencias**. Representan información sobre el medio ambiente del agente y constituyen la parte *informativa* del agente. Cada creencia se representa como una literal de base (sin variables) de la lógica de primer orden. Las literales no instanciadas se conocen como fórmulas de creencia y son usadas en la definición de planes, aunque no son consideradas creencias del agente. Las creencias se actualizan por la percepción del agente y la ejecución de sus intenciones, que producen la acción del mismo.

- **Deseos**. Representan las metas o tareas asignadas al agente y constituyen la parte *motivacional* del agente. Normalmente se consideran como lógicamente consistentes entre sí. Los deseos incluyen lograr *(achieve)* que una creencia se vuelva verdadera y verificar *(test)* si una situación, representada como una conjunción o disyunción de fórmulas de creencia, es verdadera o falsa.

- **Intenciones**. Representan los cursos de acción que el agente se ha comprometido a cumplir y constituyen la parte *deliberativa* del agente.

- **Eventos**. Las percepciones del agente se mapean a eventos discretos almacenados temporalmente en una cola de eventos. Los eventos incluyen la adquisición o eliminación de una creencia, la recepción de mensajes en el caso multiagente y la adquisición o eliminación de una nueva meta (deseo).

- **Planes**. Todo agente BDI cuenta con una librería de planes. Un plan está constituido por un evento disparador *(trigger event)* que especifica cuando un plan debe ejecutarse, un contexto que determina si el plan puede ejecutarse y un cuerpo que determina los posibles cursos de acción a ejecutar. El cuerpo toma la forma de un árbol donde los nodos se consideran estados del ambiente y los arcos son acciones o metas del agente.

Un agente BDI ejecuta las operaciones descritas en el orden adecuado, sobre las estructuras de datos de la arquitectura. Existen diferentes algoritmos BDI, uno de ellos es el que se muestra en el algoritmo 1. En un ciclo infinito, los eventos se actualizarán con base en las percepciones del agente. Mientras haya un evento, se buscará un plan aplicable y relevante, el cual se convertirá en un deseo. Teniendo este deseo, se forma una nueva intención, la cual es ejecutada.

Algoritmo 1 Agente BDI

```
procedure AGENTE-BDI(planes, creencias, deseos)
    while true do
        eventos ← percepción()
        while eventos ≠ ∅ do
            ev ← pop(eventos)
            deseos ← relevantes(aplicables(planes, creencias, ev)))
            int ← σ_des(deseos)
            intenciones ← pushInt(int, intenciones)
        end while
        ejecuta(top(σ_int(intenciones)))
    end while
end procedure
```

2.2.1. Sistemas Intencionales

La noción de **agencia débil** (Wooldridge & Jennings, 1995) define un aparato conceptual en torno a la autonomía, la iniciativa y la sociabilidad, suficiente para caracterizar los atributos ineludibles en el comportamiento de un agente, mismos que permiten diferenciar lo que es un agente, de lo que no lo es. Sin embargo, en el contexto de la IA, solemos estar interesados en descripciones de los agentes más cercanas a las que usamos al hablar del comportamiento de los seres humanos como agentes racionales con creencias, deseos, intenciones y otros atributos más allá de la agencia débil. Siendo ese nuestro interés general, es necesario abordar una **noción fuerte de agencia** que provea los argumentos a favor de tal postura. La piedra angular en esta noción fuerte de agencia es la **Intencionalidad**[2], entendida como la propiedad de los estados mentales de ser acerca de algo (Lyons, 1995).

El estudio de la Intencionalidad tiene su origen en las discusiones filosóficas medievales sobre la diferencia entre la existencia natural de las cosas, o *esse naturae*, y la existencia mental o intencional de las cosas, o *esse intentionale*, que deriva del latín *intentio* y significa dirigir la atención del pensamiento hacia algo, o simplemente apuntar hacia un objetivo, o *ser acerca de* (Lyons, 1995). La doctrina escolástica afirma que todos los hechos de conciencia poseen y manifiestan una dirección u orientación hacia un objeto. Esta orientación, que se afirma de todo pensamiento, volición, deseo o representación, en general consiste en la presencia o existencia mental del objeto que se conoce, quiere o desea; y en la referencia de este hecho a un objeto real (Ferrater Mora, 2001). Pero fue Brentano (1973) quien desarrolló la idea de que la Intencionalidad es la característica propia de todos los fenómenos mentales.

Muchos de nuestros estados mentales están en cierto sentido dirigidos a objetos o asuntos del mundo. Si tengo una creencia, debe ser una creencia que tal y tal es el caso; si deseo algo debe ser el deseo de hacer algo, o que algo ocurra; si tengo una intención, debe ser la intención de hacer algo; etc. Es esta característica de direccionalidad en nuestros estados mentales, lo que muchos filósofos han etiquetado como Intencionalidad (Searle, 1979).

Lo que es relevante en la definición anterior es que los estados mentales Intencionales parecen tener una estructura o prototipo que consiste en una actitud, como creer, desear, intentar, etc., que opera sobre el contenido del estado, que a su vez está relacionado con algo más allá de sí mismo, el objeto hacia el cual apunta. En este sentido, los estados Intencionales son representaciones de segundo orden, es decir, representaciones de representaciones. Además, si el contenido de un estado Intencional se puede expresar en forma proposicional, hablamos de una **actitud proposicional**.

[2] Para distinguir este uso técnico del término Intencionalidad, se le denotará con una mayúscula inicial, mientras que intención, con minúscula inicial, denotará el sentido común del término, como en “tiene la intención de graduarse de la universidad”.

Ejemplo 1 *La Figura 2.2 ilustra las definiciones mencionadas arriba. La proposición ϕ puede ser "la estrella es blanca" y es acerca de la estrella blanca en el medio ambiente. El agente puede tener como actitud creer (BEL), desear (DES) o intentar (INT) esa proposición. Las actitudes son acerca de la proposición, no acerca de la estrella en el medio ambiente.*

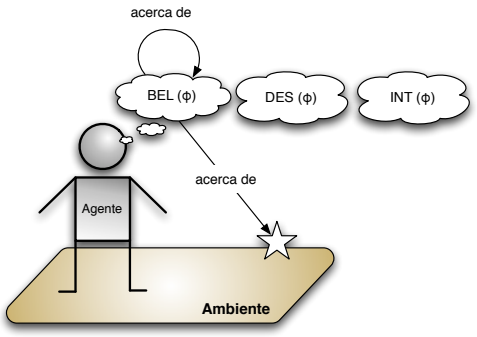


Figura 2.2 Las actitudes proposicionales son representaciones de segundo orden.

Dennett (1971, 1987), define los sistemas Intencionales como todas y sólo aquellas entidades cuyo comportamiento puede ser explicado o predicho, al menos algunas veces, asumiendo una *postura Intencional*. Tal postura consiste en la interpretación del comportamiento de la entidad en cuestión (persona, animal o artefacto) asumiendo que se trata de un agente racional que gobierna su selección de acción considerando sus actitudes proposicionales (creencias, deseos, intenciones, etc). Una escala de Intencionalidad, cuyo orden provee una escala de inteligencia, puede verse como:

- Sistemas Intencionales de primer orden. Sistemas Intencionales con creencias, deseos y otras actitudes proposicionales pero sin creencias ni deseos acerca de sus propias creencias y deseos (sin actitudes proposicionales anidadas). Ejemplo: *José intenta obtener buenas calificaciones.*

- Sistemas Intencionales de segundo orden. Sistemas Intencionales con creencias, deseos y otras actitudes proposicionales, mas creencias y deseos acerca de sus propias creencias y deseos (con actitudes proposicionales anidadas). Ejemplo: *La mamá de José desea que José intente obtener buenas calificaciones.*

- Sistemas Intencionales de orden $n > 2$. La jerarquía de Intencionalidad puede extenderse tanto como sea necesario. Ejemplo: *Todos creemos, que la mamá de José desea que José intente obtener buenas calificaciones.*

McCarthy (1979) fue uno de los primeros en argumentar a favor de la adscripción de estados mentales a máquinas, distinguiendo entre la legitimidad y el pragmatismo de tal práctica. En su opinión, adscribir creencias, deseos, intenciones, conciencia, o compromisos a una máquina o a un programa de cómputo es legítimo cuando tal adscripción expresa la misma información sobre la máquina, que expresara sobre una persona. Es útil cuando la adscripción ayuda a entender la estructura de la máquina, su comportamiento pasado y futuro, o como repararla o mejorarla. Quizá nunca sea un requisito lógico, aún en el caso de los humanos, pero si queremos expresar brevemente lo que sabemos del estado de la máquina, es posible que necesitemos de cualidades mentales como las mencionadas, o isomorfas a ellas. Es posible construir teorías de las creencias, el conocimiento y los deseos de estas máquinas en una configuración más simple que la usada con los humanos, para aplicarlas posteriormente a los éstos. Esta adscripción de cualidades mentales es más directa para máquinas cuya estructura es conocida, pero es más útil cuando se aplica a entidades cuya estructura se conoce muy parcialmente.

Tradicionalmente, tres actitudes proposicionales son consideradas para modelar agentes racionales BDI: Creencias, deseos e intenciones. El cuadro 2.1 presenta una categorización más extensa de actitudes proposicionales y su uso en el modelado de los agentes racionales (Ferber, 1995).

Uso	**Actitudes proposicionales**
Interactivas	Percepciones, informaciones, comandos, peticiones, normas.
Representacionales	Creencias, hipótesis.
Conativas	Deseos, metas, impulsos, demandas, intenciones, compromisos.
Organizacionales	Métodos, tareas.

Cuadro 2.1 Clasificación de las actitudes proposicionales de acuerdo a su utilidad en el diseño de un agente. (Ferber, 1995).

Otros argumentos computacionales para el uso de la postura Intencional han sido formulados por Singh (1995):

- Las actitudes proposicionales nos son familiares a todos, diseñadores, analistas de sistemas, programadores y usuarios;
- La postura provee descripciones sucintas del comportamiento de los sistemas complejos, por lo que ayudan a entenderlos y explicarlos;
- Provee de ciertas regularidades y patrones de acción que son independientes de la implementación física de los agentes;
- Un agente puede razonar sobre sí mismo y sobre otros agentes adoptando la postura Intencional.

2.2.2. Razonamiento Práctico

Si bien es cierto que Rao & Georgeff (1991) fueron quienes se encargaron de desarrollar la teoría computacional BDI, fue Michael Bratman (1987) quien se encargó de construir los fundamentos filosóficos que tratan de modelar la racionalidad de aquellas acciones tomadas por los seres humanos en determinadas circunstancias. Esto tiene sus orígenes en una tradición filosófica que busca comprender lo que llamamos razonamiento práctico, es decir, el razonamiento dirigido hacia las acciones: hacia el proceso de decidir qué hacer; a diferencia del razonamiento teórico que está dirigido hacia las creencias.

Ejemplo 2 *Si se cree que todos los hombres son mortales, y que Sócrates es hombre, normalmente se llegará a la conclusión de que Sócrates es mortal. Al proceso de concluir que Sócrates es mortal se le llama razonamiento teórico, debido a que éste afecta solamente mis creencias acerca del mundo. Por otro lado, al proceso de decidir tomar un tren en lugar de un autobús, se le conoce como razonamiento práctico, dado que se encuentra dirigido hacia las acciones.*

El razonamiento práctico es un asunto de sopesar consideraciones conflictivas y en contra de las opciones de la competencia, donde las consideraciones relevantes son proporcionadas por los deseos / valores / atenciones del agente sobre lo que el agente cree. El razonamiento práctico humano comprende al menos dos actividades:

1. Deliberación, es decir, decidir cuáles son las metas a satisfacer;
2. Razonamiento medios-fines, es decir, decidir cómo es que el agente va a lograr satisfacer esas metas

Descrito así, el razonamiento práctico parece ser un proceso sencillo, y podría serlo en un mundo ideal, pero en el mundo real existen varias complicaciones, la principal es que tanto la deliberación como el razonamiento medios-fines son procesos computacionales, por tanto, tales procesos computacionales se pueden ejecutar bajo ciertas limitaciones computacionales. Wooldridge (2000), argumenta que esta discusión conlleva al menos a dos implicaciones importantes:

1. Puesto que la computación es un recurso valioso para los agentes situados en ambientes de tiempo real, un agente debe controlar su razonamiento eficientemente para tener un buen desempeño.
2. Los agentes no pueden deliberar indefinidamente, deben detener su deliberación en algún momento, elegir los asuntos a atender y comprometerse a satisfacerlos.

2.2.2.1. Deliberación

El proceso de **deliberación** nos da como resultado que el agente adopte **intenciones**, las cuales son asuntos que el agente ha elegido y se ha **comprometido** a satisfacer. Las intenciones pueden adaptarse a las siguientes características:

- **Pro-actividad**. Las intenciones pueden motivar el cumplimiento de metas, son controladoras de la conducta.
- **Persistencia**. Las intenciones persisten, es decir, una vez adoptadas se resisten a ser revocadas. Sin embargo, no son irrevocables. Si la razón por la cual se creo la intención desaparece, entonces es racional abandonar la intención.
- **Intenciones futuras**. Una vez adoptada una intención, ésta restringirá los futuros razonamientos prácticos. En particular el agente no considerará adoptar intenciones incompatibles con la intención previamente adoptada, es por ello que las intenciones proveen un filtro de admisibilidad para las posibles intenciones que un agente puede considerar.

Las creencias y las intenciones mantienen ciertas relaciones, las cuales son consideradas por Bratman como principios de racionalidad. Quizá la más conocida de estas relaciones es la expresada en la **tesis de asimetría** (Bratman, 1987).

- Tener la intención de lograr ϕ, mientras se cree que no se hará ϕ, se conoce como **inconsistencia intención-creencia**, y representa un comportamiento irracional.
- Tener la intención de lograr ϕ, sin creer que se logrará ϕ, se conoce como **incompletitud intención-creencia**, y representa una propiedad aceptablemente racional .

Ejemplo 3 *Es un comportamiento irracional, que José tome la intención de ser maestro si creé que no lo podrá hacer (inconsistencia intención-creencia), al menos debe creer que podrá ser un académico, aunque no tenga la certeza de que lo será (incompletitud intención-creencia).*

Se puede hacer una distinción de dos tipos de intenciones con respecto a su proyección: i) la intención presente nos sugiere qué hacer en el ahora, y que la acción llevada a cabo es intencional; ii) la intención futura puede ser vista más bien como un compromiso, el cual dirige el curso de acción, y está estrechamente relacionada con lo que llamamos planes. Las intenciones dirigidas a futuro tienden a encausar los cursos de acción con mayor fuerza que los deseos. Las intenciones deben tener un grado de persistencia y consistencia, no tiene sentido adoptar una intención determinada para abandonarla posteriormente; las intenciones no se manejan así en la vida real.

Ejemplo 4 *Si José tiene la intención de viajar a Madrid, no se rendirá al primer intento (en caso de que José sea un agente consistente). Tampoco suena conveniente adoptar una segunda intención, que resulte inconsistente con la primera, por ejemplo planear un viaje a Barcelona en la misma fecha; la consistencia aquí funciona como un filtro de admisión, que restringe las posibles condiciones que un agente debe considerar. Esto último acota el razonamiento y resulta útil en la práctica donde los sistemas tienen recursos limitados y se manejan en tiempo real.*

2.2.2.2. Razonamiento medio-fines

El **razonamiento medios-fines** es el proceso de decidir como satisfacer una meta utilizando los medios disponibles. Por ejemplo las acciones que se pueden realizar en el entorno. El razonamiento medios-fines (ver la figura 2.3) regresa los **deseos** del agente y toma como entrada representaciones de:

1. Una meta o **intención**, que es algo que el agente quiere lograr.
2. Las **creencias** actuales del agente sobre el estado de su entorno.

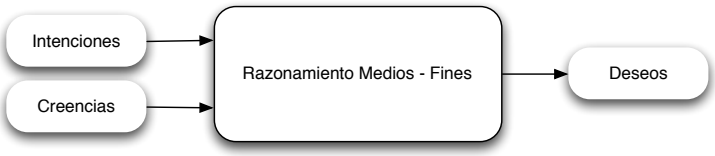


Figura 2.3 Razonamiento medios-fines.

Para ejemplificar todos los conceptos sobre intencionalidad y razonamiento práctico introducidos en esta sección, y la forma en que son usados por un agente, se introduce una arquitectura intencional abstracta, inspirada en IRMA (Bratman et al., 1988)). La notación de Wooldridge (2000) es adoptada para presentar esta arquitectura (ver la figura 2.4).

La percepción es normalmente empaquetada en paquetes discretos llamados *perceptos*. El conjunto de perceptos es denotado por *Percepciones*, mientras que los perceptos individuales se denotan por ρ. La función *percepSig* regresa la siguiente percepción disponible para el agente. Por simplicidad, sólo se muestra en la figura 2.4 como una entrada al agente ligada directamente a las creencias. La signatura de la función es la siguiente:

$$percepSig : \wp(Percepciones) \rightarrow Percepciones$$

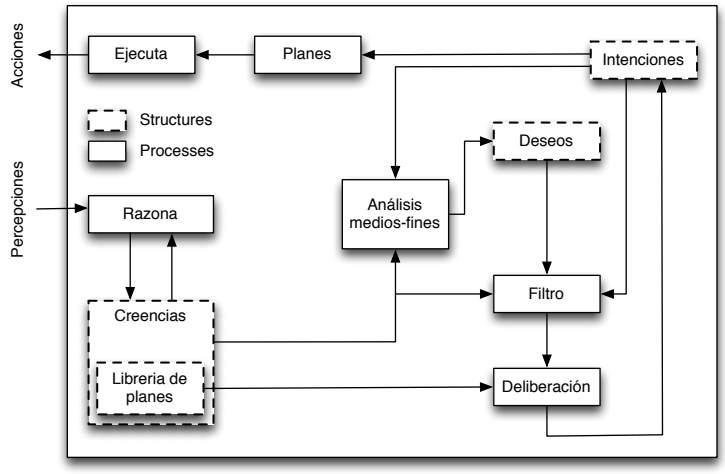


Figura 2.4 Arquitectura para agentes racionales basada en IRMA (Bratman, 1987).

Un agente actualiza sus creencias con una función de revisión de creencias basada en su percepción y sus creencias actuales. Algunas veces esta función se conoce como razonador y tiene la siguiente signatura:

$$razona : \wp(Creencias) \times Percepciones \rightarrow \wp(Creencias)$$

Un agente selecciona los planes relevantes usando un analizador medios-fines (*means-end reasoner*) basado en las creencias actuales del agente y sus intenciones previas. Los planes seleccionados por esta función se consideran como los deseos del agente. El analizador medios-fines completa los planes parciales y mantiene la coherencia medios-fines. Como hemos visto, esto promueve la formación de nuevas intenciones. La signatura de esta función es:

$$analisisMediosFines : \wp(Creencias) \times \wp(Intenciones) \rightarrow \wp(Deseos)$$

Los planes generados por el analizador medios-fines son filtrados por el agente, basado en sus creencias actuales, sus deseos y sus intenciones previas. La función filtro de admisión mantiene la consistencia y restringe las opciones que serán consideradas en la deliberación. El filtrado debe ser computacionalmente acotado, de forma que un proceso para detener el filtrado es necesario para equilibrar inercia y reconsideración. La signatura de la función filtro es:

$$filtroDeAdmision : \wp(Creencias) \times \wp(Deseos) \times \wp(Intenciones) \rightarrow \wp(Deseos)$$

Finalmente una función de deliberación selecciona las opciones que serán incorporadas como intenciones, basada en razones creencia-deseo y la biblioteca de planes. La signatura de esta función es:

$$deliberacion : \wp(Creencias) \times \wp(Deseos) \times Planes \rightarrow \wp(Intenciones)$$

Debido a que las intenciones están estructuradas como planes, una función *plan* es utilizada para seleccionar la intención que será ejecutada, la signatura de la función *plan* es:

$$plan : \wp(Intenciones) \rightarrow Planes$$

El procedimiento para operar esta arquitectura intencional abstracta se muestra en el algoritmo 2. La arquitectura IRMA (Bratman et al., 1988) fue la primera implementación de un sistema computacional basado en razonamiento práctico. De cualquier forma, la implementación mejor conocida de un sistema Intencional es PRS, desarrollado por Georgeff & Ingrad (1989) y sus diferentes re-implementaciones como UM-PRS (Lee et al., 1994), dMARS (d'Inverno et al., 1998), o Jam! (Huber, 1999).

Algoritmo 2 Agente Intencional

```
procedure AGENTE-INTENCIONAL(creencias, intenciones, planes)
    while true do
        ρ ← percepSig()
        creencias ← razona(creencias, ρ)
        deseos ← analisisMediosFines(creencias, intenciones)
        deseos ← filtroDeAdmision(creencias, deseos, intenciones)
        intenciones ← deliberacion(creencias, deseos, planes)
        π ← plan(intenciones)
        ejecuta(π)
    end while
end procedure
```

2.2.3. Actos de Habla

El uso de Intencionalidad desde una perspectiva de agentes Intencionales comunicativos, tiene sus raíces en la filosofía analítica iniciada por Frege y Russell, quienes sacan la Intencionalidad de la conciencia y la citan en el significado de las palabras, en las actitudes proposicionales (Moro Simpson, 1964). A esta postura se añaden los supuestos generales del conductismo clásico, resultando así un concepto de Intencionalidad que se define como un comportamiento lingüistico.

Antes de Austin (1975), se consideraba que el significado de un enunciado era determinado exclusivamente por su valor de verdad lógico. Sin embargo, Austin observó que algunos enunciados no pueden clasificarse como verdaderos o falsos, ya que su enunciación constituye la ejecución de una acción y por lo tanto los llamó **enunciados performativos** *(performatives sentences)*. Por ejemplo (adaptado de Perrault & Allen (1980)), dados los siguientes enunciados:

1. Pásame la sal.
2. ¿Tienes la sal?
3. ¿Te queda cerca la sal?
4. Quiero sal.
5. ¿Me puedes pasar la sal?
6. Juan me pidió que te pidiera la sal.

El significado de estos enunciados no tiene nada que ver con su valor de verdad, y está en relación con el efecto que consiguen en el medio ambiente y en los otros agentes, esto es, si conseguí la sal o no. Estos enunciados tienen una Intencionalidad asociada: todas pueden interpretarse como peticiones de la sal. Toda enunciación puede ser descrita como una acción o **acto de habla**. Austin (1975) clasificó los actos de habla en tres clases:

- **Locuciones.** Es el acto de decir algo, por ejemplo, al pronunciar una secuencia de palabras de un vocabulario en un lenguaje dado, conforme a su gramática.
- **ilocuciones.** Es el acto que se lleva a cabo al decir algo: promesas, advertencias, informes, solicitudes. Una enunciación tiene una fuerza ilocutoria F si el agente que la enuncia intenta llevar a cabo la ilocución F con ese acto. Los verbos que nombran a las ilocuciones son llamados **verbos performativos**.
- **Perlocuciones.** Es el acto que se lleva a cabo por decir algo. En (6) Juan puede, a través de su petición, convencer a los otros agentes de que le pasen la sal, y hacerse finalmente con ella.

Las ilocuciones puede definirse mediante las condiciones de necesidad y suficiencia para la ejecución exitosa del acto de habla (Searle, 1983). En particular, señala que el agente emisor ejecuta la ilocución si y sólo si intenta que el receptor reconozca su Intencionalidad en ese acto, al reconocer la fuerza ilocutoria del mismo.

Searle (1979) replantea la cuestión medieval sobre la Intencionalidad en los siguientes términos. ¿Cuál es la relación exacta entre los estados Intencionales y los objetos o asuntos a los que apuntan o son acerca de? Esta relación no es tan simple.

Ejemplo 5 *Podemos creer que el rey de Francia es de baja estatura, aún cuando no hay monarquía en ese país, sin que lo sepamos. Es decir, podemos tener un estado Intencional sobre un contenido para el cual no hay referente, e incluso es inexistente.*

La respuesta de Searle es que los estados Intencionales representan objetos y asuntos del mundo, en el mismo sentido que los actos de habla los representan. Los puntos de similitud entre estos conceptos pueden resumirse como sigue (Searle, 1979):

- Su **estructura**, resaltando por una parte la distinción entre la **fuerza ilocutoria** de los actos de habla y la **actitud** de los estados Intencionales; y por otra, la distinción entre el **contenido proposicional** de ambos.
- Las distinciones en la **dirección de ajuste** entre palabra y realidad (ver la figura 2.5) familiares en los actos de habla, también aplican en el caso de los estados Intencionales. Se espera que los **actos asertivos** (afirmaciones, descripciones, aserciones) de alguna forma tengan **correspondencia con el mundo** cuya existencia es independiente. Pero los **actos directivos** (ordenes, comandos, peticiones) y los **comisorios** (promesas, ruegos, juramentos) no se supone que tengan correspondencia con la realidad independientemente existente, sino por el contrario causan que el mundo corresponda con lo expresado. En ese sentido, no son verdaderos, ni falsos, sino que son exitosos o fallidos, se mantienen o se rompen, etc. Y aún más, puede haber casos de no direccionalidad, por ejemplo cuando felicito a alguien.

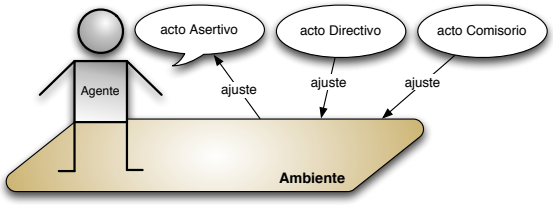


Figura 2.5 Direccionalidad en el ajuste entre los actos de habla y el medio ambiente del agente.

Lo mismo sucede en los estados Intencionales (ver la figura 2.6). Si mis creencias fallan, es culpa de mis creencias, no del mundo y de hecho puedo corregir la situación cambiando mis creencias (**mantenimiento de creencias**). Pero si mis intenciones o mis deseos fallan, no puedo corregirlos en ese sentido. Las creencias, como las aserciones, son falsas o verdaderas; las intenciones como los actos directivos y los comisorios, fallan o tienen éxito.

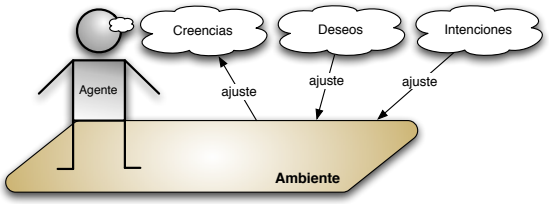


Figura 2.6 Direccionalidad en el ajuste entre los estados Intencionales BDI y el medio ambiente del agente.

- En general, al ejecutar un acto de habla ilocutorio con contenido proposicional, expresamos cierto estado Intencional con ese contenido proposicional, y tal estado Intencional es la **condición de sinceridad** (ver la figura 2.7) de ese tipo de acto de habla. Por ejemplo, si afirmo ϕ, estoy expresando que creo ϕ; si prometo ϕ estoy expresando que intento ϕ; si ordeno ϕ estoy expresando mi deseo por ϕ. Esta relación es interna en el sentido de que el estado Intencional no es un acompañante de la ejecución del acto de habla, tal ejecución es necesariamente la expresión del estado Intencional. Las declaraciones constituyen una excepción a todo esto, pues no tienen condiciones de sinceridad ni estado Intencional. Y por supuesto, siempre es posible mentir, pero un acto de habla insincero, consiste en ejecutar el acto para expresar un estado Intencional que no se tiene; por lo cual lo dicho aquí se mantiene.

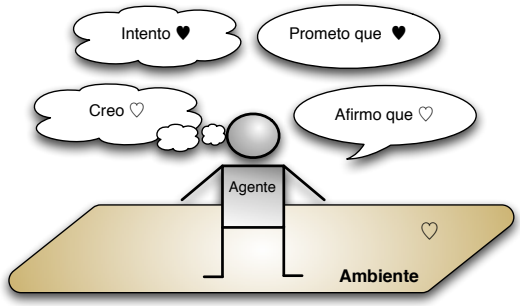


Figura 2.7 Los actos de habla ilocutorios expresan estados Intencionales que son a su vez la condición de sinceridad de lo expresado.

- El concepto de **condiciones de satisfacción** aplica en ambos casos. Por ejemplo, una afirmación se satisface si y sólo si es verdadera; una promesa se satisface si y sólo si se cumple; una orden se satisface si y sólo si se obedece; etc. Lo mismo para los estados Intencionales: mi creencia se satisface si las cosas son como creo; mis intenciones se satisfacen si son logradas; mis deseos se satisfacen si son llevados a cabo. La noción de satisfacción parece natural y aplica siempre que haya una dirección de ajuste presente. Lo relevante aquí es

que el acto de habla se satisface si y sólo si el estado Intencional expresado se satisface. La excepción aquí se da cuando el estado Intencional se satisface por causas ajenas a la ejecución del acto de habla, por ejemplo: prometí hacerme de un libro de sistemas multiagente y me lo regalaron. El estado Intencional (hacerme del libro) se satisface, pero no así mi promesa.

Una taxonomía de los actos de habla es una taxonomía de los estados Intencionales. Aquellos estados Intencionales cuyo contenido son proposiciones completas, las llamadas **actitudes proposicionales**, se dividen convenientemente en aquellas que ajustan el estado mental al mundo, el mundo al estado mental y las que no tienen dirección de ajuste. No todas las intenciones tienen contenido proposicional completo, algunos estados son sólo acerca de objetos.

2.3. Estrategias de compromiso

Como se mencionó en la página 21, cuando se eligen los planes para formar intenciones a través de la función de filtro, se dice que el agente se ha **comprometido** a ejecutar esa intención. El compromiso implica persistencia temporal, esto es, una vez adoptada una intención, no debería evaporarse inmediatamente. Una cuestión fundamental es el grado de compromiso que un agente debe tener sobre sus intenciones. Es decir, ¿Cuánto tiempo debe persistir una intención? ¿Bajo qué circunstancias se debe eliminar una intención?

El mecanismo que un agente utiliza para determinar cuándo y cómo eliminar las intenciones es conocido como **estrategia de compromiso**. Tres estrategias de compromiso son bien conocidas en la literatura de los agentes racionales (Rao & Georgeff, 1991): los compromisos ciego, racional, y emocional[3].

2.3.1. Compromiso ciego (blind)

Un agente se compromete ciegamente si mantiene sus intenciones hasta que cree que ha logrado satisfacerlas. Formalmente, el teorema 1 especifica que, si un agente intenta que inevitablemente ϕ sea eventualmente *verdadero*, entonces el agente inevitablemente mantendrá sus intenciones hasta creer ϕ.

[3] Originalmente estos compromisos se llaman *blind*, *single-minded* y *open-minded* respectivamente. Guerra-Hernández (2010) propone cambios a su denotación en español basándose en que la segunda es una estrategia *creencia-intención* y la tercera, una estrategia *deseo-intención*. Los términos *racional* y *emocional* enfatizan mejor el fundamento de la reconsideración, que los términos *inflexible* y *flexible*.

Teorema 1 $\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \rightarrow \mathsf{A}(\mathsf{INTEND}(\mathsf{A}\Diamond\phi)\ \mathsf{U}\ \mathsf{BEL}(\phi))$

Observe que este axioma se define para i-fórmulas[4]. Nada se dice sobre la intención de un agente por lograr opcionalmente algún medio o fin particular. Resulta evidente que esta estrategia es demasiado fuerte, pues para un agente comprometido ciegamente resulta inevitable eventualmente creer que ha logrado sus medios (o fines). Esto se debe a que este tipo de compromiso sólo permite caminos futuros en los cuales, o bien el objeto de la intención es creído, o bien la intención se mantiene para siempre. Sin embargo, debido a la propiedad de no-deliberación infinita *(no-infinite deferral)*, tenemos que $\mathsf{INTEND}(\phi) \rightarrow \mathsf{A}\Diamond(\neg\mathsf{INTEND}(\phi))$, por lo que tal clase de caminos no está permitida, y en consecuencia obtenemos agentes que creen que eventualmente han logrado sus intenciones (Teorema 2).

Teorema 2 $\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \rightarrow \mathsf{A}\Diamond(\mathsf{BEL}(\phi))$

2.3.2. Compromiso racional (single-minded)

Se puede definir una estrategia de compromiso racional relajando la estrategia de compromiso ciega, de manera que el agente mantenga sus intenciones en tanto considere que siguen siendo una opción viable. Formalmente:

Teorema 3 $\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \rightarrow \mathsf{A}(\mathsf{INTEND}(\mathsf{A}\Diamond\phi)\ \mathsf{U}\ (\mathsf{BEL}(\phi) \vee \neg\mathsf{BEL}(\mathsf{E}\Diamond\phi)))$

En tanto el agente crea que sus intenciones se pueden lograr, un agente con compromiso racional no abandonará sus intenciones y seguirá comprometido hacia sus deseos. Así, un agente básico con compromiso racional, de manera inevitable, eventualmente creerá que ha logrado satisfacer sus medios (o fines) sólo si continúa creyendo que, hasta creer que sus medios (o fines) se han satisfecho, el objeto de sus intenciones sigue siendo una opción. Formalmente:

Teorema 4 $\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \wedge \mathsf{A}(\mathsf{BEL}(\mathsf{E}\Diamond\phi)\ \mathsf{U}\ \mathsf{BEL}(\phi)) \rightarrow \mathsf{A}\Diamond(\mathsf{BEL}(\phi))$.

2.3.3. Compromiso emocional (open-minded)

Un agente bajo el compromiso emocional mantiene sus intenciones mientras éstas sigan siendo deseadas. Formalmente:

Teorema 5 $\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \rightarrow \mathsf{A}(\mathsf{INTEND}(\mathsf{A}\Diamond\phi)\ \mathsf{U}\ \ (\mathsf{BEL}(\phi) \vee \neg\mathsf{DES}(\mathsf{E}\Diamond\phi)))$

[4] Se hace uso de operadores BDI (BEL,DES,INT), y operadores temporales (Inevitablemente(A), Eventualmente($\Diamond$) y Hasta(U)). Los operadores BDI y CTL están bajo el alcance del operador *inevitable* (A).

Un agente básico con compromiso emocional, de manera inevitable, eventualmente creerá que ha logrado satisfacer sus medios (o fines) si mantiene sus intenciones como deseos, hasta que cree que ha logrado satisfacerlos. Formalmente:

Teorema 6 $\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \wedge \mathsf{A}(\mathsf{DES}(\mathsf{E}\Diamond\phi))\ \mathsf{U}\ \mathsf{BEL}(\phi) \rightarrow\ \mathsf{A}\Diamond(\mathsf{BEL}(\phi))$.

2.4. Programación Orientada a Agentes

Shoham (1990) propuso un nuevo paradigma de programación, el cual es denotado como **Programación Orientada a Agentes (POA)**. La idea principal que denuncia la POA, es la programación de agentes directamente en términos de nociones mentalistas (como las creencias, los deseos y las intenciones) que los teóricos de agentes han desarrollado para representar las propiedades de los agentes. La motivación detrás de la propuesta es que los seres humanos utilizan conceptos como un mecanismo de abstracción para representar las propiedades de sistemas complejos, de la misma manera que utilizamos estos conceptos mentalistas para describir y explicar el comportamiento de los seres humanos, por lo que podría ser útil para el uso de programas computacionales.

En la POA, los agentes pueden abstraerse como objetos en la Programación Orientada a Objetos (POO), cuyos estados son Intencionales. Una computación en este paradigma está relacionada con los agentes informando, requiriendo, ofertando, aceptando, rechazando, compitiendo, y ayudándose. El cuadro 2.2 resume esta comparación.

Concepto	POO	POA
Unidad básica	Objeto	Agente
Estado	No restringido	Creencias, Deseos, Intenciones, etc.
Cómputo	Paso de mensajes y métodos	Paso de mensajes y métodos
Tipos de Mensajes	No restringido	Informes, Solicitudes, Promesas, etc.
Restricciones en métodos	Ninguno	Honestidad, Coherencia, etc.

Cuadro 2.2 Comparación entre la Programación Orientada a Agentes (POA) y la Orientada a Objetos (POO)(Shoham, 1990).

Un sistema completo de POA deberá contar con:

- Un *lenguaje formal* restringido, de sintaxis y semántica claras, para describir los estados Intencionales;
- Un lenguaje de programación *interpretado* para definir los programas de agentes; el cual debe ser fiel a la semántica de los estados Intencionales; y
- Un "agentificador" que convierta entidades neutras en agentes programables.

2.4.1. AgentSpeak(L)

AgentSpeak(L) es un lenguaje abstracto de programación orientada a agentes definido por Rao (1996) en respuesta al distanciamiento que se había presentado entre la teoría y la práctica en el desarrollo de agentes Intencionales BDI. Si bien es cierto que las lógicas BDI son lo suficientemente expresivas como para especificar, definir y verificar este tipo de agentes, su alta complejidad hacía que los desarrolladores prefirieran usar estructuras de datos para representar a los operadores Intencionales, y se distanciasen de su definición como operadores modales.

La propuesta de Rao fue proveer una formalización alternativa a la modal de los agentes BDI con una semántica operacional y teoría de prueba bien definidas, dando lugar a programas de agente similares a los usados en programación lógica con cláusulas de Horn. *AgentSpeak(L)* es un lenguaje de programación basado en una lógica restringida de primer orden con eventos y acciones. Las actitudes proposicionales no están representadas directamente como expresiones modales[5]. El estado actual de un agente, que es modelo de él mismo, su ambiente y otros agentes, puede considerarse como las *creencias* presentes del agente. Los estados que el agente quiere lograr con base en sus estímulos internos y externos, constituyen sus *deseos*, y la adopción de programas para satisfacer estos deseos, constituyen las *intenciones* del agente. Esta apuesta por adscribir intencionalidad a un modelo ejecutable del agente, constituyó un cambio de paradigma que acercaba la teoría a la praxis de la agencia racional. $AgentSpeak(L)$ se asemeja más a Agent-0 (Shoham, 1990) que a su referente dMARS (Rao, 1996) y al igual que éste último, ha resultado fundamental para el estudio de los lenguajes de programación orientados a agentes.

El alfabeto de este lenguaje formal consiste en variables, constantes, símbolos funcionales, símbolos de predicado, símbolos de acciones, conectivas, cuantificadores y signos de puntuación. Además de las conectivas de primer orden, se usan los caracteres "!" y "?" para metas; ";" para secuencias y $\leftarrow$ para implicación. Las definiciones estándar para término, fórmula bien formada (fbf), fbf cerrada, y ocurrencia libre y acotada de variables son adoptadas.

Definición 3 (Creencias) *Sea b un símbolo de predicado y $t_1,\ldots,t_n$ una secuencia de términos, entonces $b(t1,\ldots,t_n)$ o $b(\mathbf{t})$ es una creencia atómica. Si $b(t)$ y $c(s)$ son creencias atómicas, entonces $b(t)\wedge c(s)$ y $\neg b(t)$ son creencias. Una creencia atómica o su negación suelen ser identificadas como literal de creencia. Una creencia atómica sin variables libres suele ser llamada creencia de base.*

[5] Aunque se ha definido CTL AgentSpeak(L) para razonar acerca de estos programas (Guerra-Hernández et al., 2009).

El cuadro 2.3 presenta un agente *AgentSpeak(L)* para el mundo de los bloques, el cual cuenta con las creencias de que el bloque *b* esta sobre el bloque *a*, los bloques *a* y *c* están sobre la mesa y la mesa está libre (líneas 1-4). Además, deduce que el bloque *c* esta libre (línea 5).

```
1  on(b,a).
2  on(a,table).
3  on(c,table).
4  clear(table).
5  clear(X) :- not(on(_,X)).
6
7  @put_1
8  +!put(X,Y) : clear(X) <- move(X,Y).
9  @put_2
10 +!put(X,Y) : not (clear(X)) <- .print("Imposible mover el bloque ",X," al bloque ",Y).
```

Cuadro 2.3 Implementacion de un agente $AgentSpeak(L)$ en el mundo de los bloques, adaptado de (Bordini et al., 2007)

Definición 4 (Metas) *Si g es un símbolo de predicado y* $t_1,\ldots,t_n$ *son términos, entonces* $!g(t_1,\ldots,t_n)$ *o* $!g(\mathbf{t})$ *y* $?g(t_1,\ldots,t_n)$ *o* $?g(\mathbf{t})$ *son metas.*

Una meta es un estado del sistema que el agente desearía ver logrado. Los agentes en $AgentSpeak(L)$ consideran dos tipos de meta, como el resto de los agentes BDI: las metas que el agente quiere propiamente lograr (*achieve goals*) $!g(t)$; y las metas que el agente quiere verificar lógicamente (*test goals*) $?g(t)$. En el ejemplo del mundo de los bloques, poner el bloque *b* sobre el bloque *c* constituye una meta a lograr $!put(b,c)$ y preguntarse si el bloque *c* está libre, constituye una meta a verificar $?clear(c)$. Obsérvese que las metas logrables involucran razonamiento práctico, mientras que las verificables involucran razonamiento epistémico. Las metas equivalen a los **deseos** en el modelo BDI.

Definición 5 (Eventos disparadores) *Si* $b(t)$ *es una creencia atómica, y* $!g(t)$ *y* $?g(t)$ *son metas, entones* $+b(t)$, $-b(t)$, $+!g(t)$, $+?g(t)$, $-!g(t)$ *y* $-?g(t)$ *son eventos disparadores.*

Los cambios en el entorno del agente y en su estado interno generan eventos disparadores (*trigger events*). Estos eventos incluyen agregar $(+)$ y borrar $(-)$ metas o creencias al estado del agente. Por ejemplo, detectar que el bloque *b* se ha colocado sobre el bloque *c* toma la forma del evento disparador $+on(b,c)$ y adquirir la meta de colocar el bloque *b* sobre el bloque *c* toma la forma del evento disparador $+!put(b,c)$.

Definición 6 (Acciones) *Si a es un símbolo de acción y* $t_1,\ldots t_n$ *son términos de primer orden, entonces* $a(t_1,\ldots,t_n)$ *o* $a(\mathbf{t})$ *es una acción.*

El agente debe ejecutar acciones para lograr el cumplimiento de sus metas. Las acciones pueden verse como procedimientos a ejecutar. Una acción como $put(X,Y)$ deberá tener como resultado que un bloque X quede ubicado sobre un bloque Y. Normalmente, estas acciones son implementadas en el mismo lenguaje de programación en el que $AgentSpeak(L)$ ha sido implementado.

Definición 7 (Planes) *Si e es un evento disparador, $b_1,\ldots,b_n$ son literales de creencia, y $g_1,\ldots g_m$ son metas o acciones, entonces $e : b_1 \wedge \cdots \wedge b_n \leftarrow g_1;\ldots;g_m$ es un plan. La expresión a la izquierda de la flecha se conoce como la cabeza del plan. La expresión a la derecha de la flecha se conoce como cuerpo del plan. La expresión a la derecha de los dos puntos en la cabeza del plan se conoce como contexto. Por convención un cuerpo vacío se escribe $true$.*

Como el resto de los agentes BDI, los agentes de $AgentSpeak(L)$ poseen una biblioteca de planes. El cuadro 2.3 (líneas 7-10) describe los planes a ejecutarse para poner un bloque X sobre un bloque Y cuando, el bloque X esté libre (línea 8) o cuando el bloque X no esté libre (línea 10).

La semántica operacional de *AgentSpeak(L)* se describe en del apéndice A.1.

2.4.2. Jason

Jason[6] (Bordini et al., 2005; Bordini & Hübner, 2006; Bordini et al., 2007) es un intérprete en Java que implementa una semántica operacional extendida de $AgentSpeak(L)$, la cual se describe en el apéndice A.2. Sus características principales incluyen:

- Comunicación entre agentes basada en **actos de habla**. En particular, se implementa el protocolo conocido como *Knowledge Query and Manipulation Language* (KQML) (Finin et al., 1992) y se consideran anotaciones en las creencias con información sobre las fuentes de los mensajes (Vieira et al., 2007).
- Anotaciones sobre los planes, las cuales pueden ser empleadas para diseñar funciones de selección basadas en **teoría de decisión** (Bordini et al., 2002). Las funciones de selección son totalmente configurables desde Java.
- La posibilidad de ejecutar sistemas multiagente **distribuidos** sobre una red, utilizando SACI, Jade (Bellifemine et al., 2007) o algún middleware (Bordini et al., 2004).
- **Acciones internas programables** en Java.
- Una clara noción de **medio ambiente**, que permite simular la situacionalidad de los agentes en cualquier ambiente implementado en Java.

[6] Disponible en `http://jason.sourceforge.net/Jason/Jason.html`

- Incorpora un ambiente de desarrollo basado en jEdit y un plugin para Elipse, que facilitan la implementación de sistemas multiagente en este lenguaje.

Hay dos aspectos más de *Jason* a resaltar:

- Sus **fundamentos teóricos** basados en $AgentSpeak(L)$, incluyendo la verificación formal de las propiedades de los agentes (Bordini et al., 2003b,a; Bordini & Moreira, 2004; Guerra-Hernández et al., 2008a, 2009).
- Su **modelo de desarrollo abierto**, basado en una licencia libre GNU LGPL y Java.

$$
\begin{array}{lcll}
ag & ::= & bs \quad ps & \\
bs & ::= & b_1 \ldots b_n & (n \geq 0) \\
ps & ::= & p_1 \ldots p_n & (n \geq 1) \\
p & ::= & te : ct \leftarrow h & \\
te & ::= & +at \mid -at \mid +g \mid -g & \\
ct & ::= & ct_1 \mid \top & \\
ct_1 & ::= & at \mid \neg at \mid ct_1 \wedge ct_1 & \\
h & ::= & h_1 ; \top \mid \top & \\
h_1 & ::= & a \mid g \mid u \mid h_1 ; h_1 & \\
at & ::= & P(t_1, \ldots, t_n) & (n \geq 0) \\
 & \mid & P(t_1, \ldots, t_n)[s_1, \ldots, s_m] & (n \geq 0, m > 0) \\
s & ::= & \texttt{percept} \mid \texttt{self} \mid id & \\
a & ::= & A(t_1, \ldots, t_n) & (n \geq 0) \\
g & ::= & !at \mid ?at & \\
u & ::= & +b \mid -b &
\end{array}
$$

Cuadro 2.4 Sintaxis de *Jason*. Adaptada de Bordini et al. (2007).

El cuadro 2.4 define una gramática simplificada del lenguaje de *Jason*. El símbolo $\top$ denota elementos vacíos en el lenguaje. Como es usual, un agente *ag* está definido por un conjunto de **creencias** *bs* y **planes** *ps*. Cada **creencia** $b \in bs$ toma la forma de un átomo de base (sin variables libres) de primer orden. Cada **plan** $p \in ps$ tiene la forma $te : ct \leftarrow h$, donde:

- *te* se conoce como el **evento disparador** del plan y define el suceso que provoca que éste sea considerado como relevante. Los sucesos a considerar son básicamente actualizaciones en las creencias y los deseos (metas, en este contexto) del agente. Las fbf para expresar estas actualizaciones incluyen agregar ($+$) o eliminar ($-$) un átomo de primer orden; o una meta (g). Las **metas** son de dos tipos: conseguir (!) que un átomo sea verdadero; y verificar (?) si ya lo es. El evento disparador de un plan no puede ser vacío. A diferencia de las creencias, el evento disparador puede incluir variables libres.

- *ct* se conoce como el **contexto** del plan y define las condiciones que hacen que éste sea efectivamente ejecutable. Tales condiciones se expresan como literales (átomos o su negación) de primer orden o una conjunción de ellas. El contexto vacío está asociado a planes que son ejecutables en cualquier circunstancia del agente.
- *h* se conoce como el **cuerpo** del plan. Un cuerpo no vacío es una secuencia finita de **acciones** (a), metas (g) y actualizaciones de creencias (u), las actualizaciones de creencias solo aceptan átomos de base como argumento.

La principal diferencia entre la sintaxis de *Jason* y la propuesta original de $AgentSpeak(L)$ son las **anotaciones** en los átomos del lenguaje con etiquetas (s) que indican el origen de la fórmula en cuestión. El origen puede ser la percepción (`percept`) del agente, su razonamiento (`self`) o un mensaje proveniente de otro agente identificado como `id`. En la práctica, existen otras extensiones, por ejemplo, el operador $-+$, que agrega una creencia inmediatamente después de remover la primera ocurrencia existente de esa creencia en la base de creencias del agente; además de que los planes también pueden etiquetarse.

2.5. Resumen

Este capítulo introduce el término agente bajo dos nociones que (Wooldridge & Jennings, 1995) define como **noción débil** y **noción fuerte de agencia**. La primera permite diferenciar lo que es una agente de lo que no lo es, mediante una conceptualización de agencia en torno a la autonomía, la iniciativa y la sociabilidad de un agente. Por otro lado, una noción fuerte de agencia, que es la que nos atañe en este proyecto de investigación, nos proporciona los argumentos necesarios para describir agentes dirigidos a la representación del comportamiento del ser humano como agente racional.

El comportamiento racional humano, puede representarse como un modelo BDI basado en sus estados mentales *(Deseos, Creencias e Intenciones)*, el cual cuenta con presupuestos filosóficos sólidos (Bratman, 1987; Dennett, 1987; Searle, 1962). Bajo este contexto, la *Intencionalidad* puede ser entendida como la propiedad de los estados mentales de ser acerca de algo. La intención de lograr una meta debería persistir con el tiempo, sin embargo, se ha demostrado que abandonar una intención cuando se cree que no se podrá ejecutar con éxito exhibe un comportamiento racional. Las estrategias de compromiso nos permiten determinar cuándo y cómo se deben eliminar intenciones.

Dadas estas definiciones, un nuevo paradigma de programación ha emergido, la *Programación Orientada a Agentes*, introducida por Shoham (1990). Un lenguaje de de Programación Orientado a Agentes ha sido presentado en este capítulo: *AgentSpeak(L)*, que en su concepción original surgió como un lenguaje destinado a

facilitar la comprensión de la relación entre la aplicación práctica de la arquitectura BDI y la formalización de las ideas detrás de la arquitectura BDI con lógicas modales. Si bien es cierto que las lógicas BDI eran lo suficientemente expresivas como para especificar, definir y verificar este tipo de agentes, su alta complejidad hacía que los desarrolladores prefirieran usar estructuras de datos para representar a los operadores Intencionales, y se distanciasen de su definición como operadores modales.

Tanto el lenguaje $AgentSpeak(L)$, como su intérprete en Java Jason presentan un marco de trabajo elegante para programar agentes BDI, basado en lógica de primer orden. Ambos han sido ampliamente aceptados por la comunidad de investigación en SMA.

Capítulo 3
Aprendizaje Lógico Inductivo

El **aprendizaje** es un fenómeno multifacético que incluye la adquisición de conocimiento nuevo, el desarrollo de habilidades motoras y cognitivas a través de la instrucción o práctica, la organización de conocimiento nuevo en representaciones efectivas y el descubrimiento de nuevos hechos y teorías a través de la observación y la experimentación. En otras palabras, el aprendizaje puede ser visto como una **actualización** en el comportamiento, habilidades o conocimiento en general con la finalidad de mejorar el desempeño (Mitchell, 1997).

En términos computacionales, desde el inicio de la era computacional los investigadores han tratado de implementar estas capacidades en las computadoras. La solución de este problema ha sido, y sigue siendo uno de los más desafiantes objetivos de la IA.

El aprendizaje es tema central en este trabajo de investigación, por ello este capítulo presenta el aprendizaje lógico inductivo como el mecanismo idóneo para proporcionar un comportamiento adaptativo en agentes *Agent Speak(L)*. La primera sección presenta una arquitectura abstracta de agentes que son capaces de aprender, y semántica introducida por Ortiz-Hernández (2007) para dar sustento al aprendizaje Intencional. En la segunda sección se introduce un mecanismo ya conocido dentro de la tarea de clasificación: los árboles de decisión, de los cuales se presentan sus algoritmos más utilizados en el área del aprendizaje automático: ID3 (Quinlan, 1986) y C4.5 (Quinlan, 1993). Por último, en la tercera sección se presenta el paradigma de Programación Lógica Inductiva, que se define como la intersección entre el aprendizaje autmático y la programación lógica. En esta última sección se introduce la morfología de la versión en primer orden de los árboles de decisión: los **árboles lógicos de decisión** los cuales son explicados a detalle en el capítulo 4.

3.1. Agentes que aprenden

Al dotar a un agente con un mecanismo de aprendizaje, éste adquiere la ventaja de operar en entornos total o parcialmente desconocidos, y volverse competente conforme pasa el tiempo a partir de su conocimiento inicial. Un agente con capacidad de aprendizaje se puede dividir en cuatro componentes conceptuales (ver la figura 3.1).

- Un elemento de **aprendizaje**, el cual es responsable de desarrollar mejoras.
- Un elemento de **desempeño**, responsable de seleccionar las acciones externas con base en las percepciones.
- Un elemento de **crítica**, que indica al elemento de aprendizaje que tan bien esta realizando el aprendizaje el agente, con respecto a un nivel de actuación fijo. El elemento de aprendizaje utiliza una retroalimentación de la crítica que recibe la actuación del agente y con base en esto determina como debe el elemento de desarrollo ser modificado con el fin de hacerlo mejor en un futuro.
- Un **generador de problemas**, que es responsable de sugerir acciones que le conduzcan a experiencias nuevas e informativas.

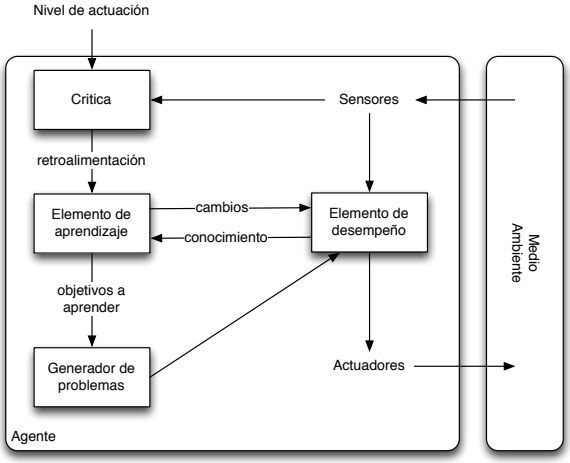


Figura 3.1 Arquitectura abstracta de un agente que aprende. Adaptada de (Russell & Norvig, 2003).

Lo interesante es que mientras el elemento de desempeño sigue su camino, puede continuar llevando a cabo las acciones que sean mejores, de acuerdo con su conocimiento. Pero si el agente esta dispuesto a explorar un poco, y llevar acabo algunas acciones que no sean totalmente óptimas a corto plazo, puede descubrir acciones mejores a largo plazo. El trabajo del generador de problemas es sugerir estas acciones exploratorias.

Entiéndase aquí por aprendizaje, la adquisición de una definición general a partir de una serie de ejemplos de entrenamiento etiquetados como negativos o positivos, es decir, inducir una función a partir de ejemplos de entrenamiento.

El estudio y modelado computacional de los procesos de aprendizaje en sus múltiples manifestaciones constituye el tema de estudio del **Aprendizaje Automático** *(Machine Learning)*, el cual tiene como objetivo desarrollar sistemas que mejoren su funcionamiento para realizar una tarea de forma automática partiendo de su experiencia adquirida. Una forma de abordar el aprendizaje es considerarlo un problema de búsqueda, de esta manera se trata de encontrar dentro de un espacio de hipótesis posibles aquella que mejor corresponda a los datos observados.

Un concepto importante en aprendizaje es el de **función objetivo**, que se entiende como la definición del tipo de conocimiento que debe ser aprendido. Este conocimiento puede referirse por ejemplo, a la evaluación de una tarea representativa dentro de un dominio particular. Formalmente, la función objetivo se define como:

$$F : E \rightarrow A$$

Donde E es el conjunto de estados posibles en el ambiente y A es el conjunto de acciones o valores asignados a las acciones.

Ejemplo 6 *En un tablero de ajedrez,* E *podrá representar el conjunto de posiciones (estados), y* A *la evaluación en números reales de los movimientos posibles dadas las posiciones.*

A partir de lo anterior, el aprendizaje automático puede entenderse como la búsqueda de la descripción de una función objetivo con el fin de utilizar algún algoritmo para computarla. Al diseñar un sistema de aprendizaje se deben tomar en cuenta varios elementos: el tipo de experiencia de entrenamiento, la medida de desempeño, la función objetivo y su representación, así como el algoritmo para aproximarla. Considerando el tipo de experiencia utilizada, el aprendizaje automático puede dividirse en tres tipos principales (Russell & Norvig, 2003), de los cuales, solo el primero es considerado como objeto de estudio en esta investigación.

- *Supervisado*: Consiste en aprender una función a partir de ejemplos de entradas y salidas, es decir, las clases son definidas previamente y con base en ellas se **clasifican** los datos.
- *No supervisado*: Consiste en aprender patrones de entradas cuando no hay valores de salida especificados. Las clases se infieren de los datos, creando **grupos** diferenciados.
- *Por refuerzo*: El aprendizaje se basa en la evaluación de un refuerzo o **recompensa** para el conjunto de acciones realizadas.

Un concepto central en el aprendizaje es la **inducción**, la cual consiste en aprender funciones generales a partir de ejemplos particulares para obtener un modelo de aprendizaje.

Definición 8 (Hipótesis del aprendizaje inductivo) *Cualquier estimación (hipótesis) que aproxime la función objetivo a partir de un conjunto suficientemente grande de ejemplos de entrenamiento, puede aproximar también la función objetivo para ejemplos no observados.*

De esta forma, un algoritmo puede aprender a realizar una tarea, por ejemplo clasificar datos, si obtiene una buena aproximación de la función objetivo tomando en cuenta solo los ejemplos de entrenamiento. En lógica proposicional, la hipótesis aprendida es comúnmente expresada mediante una disyunción de conjunciones proposicionales, las cuales pueden ser representadas mediante árboles de decisión.

El concepto de aprendizaje es central en del campo de la IA: un sistema que aprende es usualmente considerado como inteligente y viceversa, un sistema considerado como inteligente, es normalmente considerado capaz de aprender. Dentro del contexto de agencia se entiende por aprendizaje la adquisición de conocimientos y habilidades por un agente, al cual llamamos agente aprendiz, bajo los siguientes términos (Sen & Weiß, 1999):

1. Esta adquisición es conducida por el propio agente.
2. El resultado de esta adquisición de conocimiento es incorporado por el agente aprendiz en sus actividades futuras.
3. Esta incorporación de conocimientos y habilidades lleva al agente a lograr un mejor rendimiento.

Principalmente se pueden distinguir dos tipos de aprendizaje con base en la ubicuidad de su ejecución:

- **Aprendizaje Centralizado**: Este tipo de aprendizaje se caracteriza porque todo el proceso implicado en la adquisición de conocimiento es ejecutado por un solo agente. Aquí no es necesaria la interacción social con otros agentes. Es posible que el agente se encuentre situado en un SMA, sin embargo, el proceso de aprendizaje se lleva a cabo como si éste estuviera solo.
- **Aprendizaje Distribuido**: También se le conoce como aprendizaje interactivo, o aprendizaje social. Aquí, varios agentes se encuentran implicados en el proceso de aprendizaje. Puede ser visto como un grupo de agentes tratando de resolver el mismo problema a manera de equipo. Este tipo de aprendizaje se basa principalmente en las capacidades particulares que tiene cada agente.

Existe otra clasificación de aprendizaje basada en la forma en que modela a su entorno social y el comportamiento de otros agentes (Guerra-Hernández et al., 2004a), la cual corresponde de cierto modo con la escala de la intencionalidad de Dennett (1987):

- **Nivel 0**: El caso en el que existe un solo agente, el verdadero caso aislado de aprendizaje. Éste puede ser visto como un caso especial de nivel 1.
- **Nivel 1**: Los agentes actúan y aprenden de la interacción directa con su entorno, sin estar explícitamente conscientes de otros agentes en el SMA. De todos modos, los cambios que otros agentes producen en el ambiente pueden ser percibidos por el agente aprendiz.
- **Nivel 2**: Los agentes actúan y aprenden de la interacción directa con otros agentes mediante el intercambio de mensajes. El intercambio de ejemplos de entrenamiento en los procesos de aprendizaje también se considera dentro de este nivel. Este es el caso Intencional BDI de aprendizaje, donde se combina el aprendizaje Intencional con el estado mental BDI de los agentes y la comunicación a través de actos de habla.
- **Nivel 3**: Los agentes aprenden de la observación de las acciones de otros agentes. Se trata de un tipo diferente de conciencia de la de nivel 2. Los agentes no sólo son conscientes de la presencia de otros agentes, sino que también son conscientes de sus competencias

3.1.1. Agentes Intencionales que aprenden

Nuestro uso del término **aprendizaje Intencional** está estrictamente circunscrito a la teoría de racionalidad práctica de Bratman (1987), donde los planes están predefinidos y el objetivo de los procesos de aprendizaje es la razón para adoptar las intenciones, es decir, el contexto de sus planes.

Guerra-Hernández et al. (2008b) proponen una semántica operacional[1] para definir agentes capaces de aprender Intencionalmente. Las transiciones se definen en términos de reglas semánticas con la forma:

$$(\textbf{rule id}) \quad \frac{cond}{C \rightarrow C'}$$

donde $C = \langle ag, C, M, T, s \rangle$ es una configuración que puede transformarse en una nueva configuración C', si *cond* se cumple. Para efectos de describir las reglas semánticas adoptamos la siguiente notación:

[1] Con el propósito de mantener las reglas semánticas coherentes con la definición de la semántica operacional de Jason (apéndice A.2), se emplea la misma notación empleada en (Bordini et al., 2002; Moreira & Bordini, 2003; Moreira et al., 2003).

- Para hacer referencia al componente E (eventos) de una circunstancia C, escribimos C_E. De manera similar accedemos a los demás componentes de una configuración.
- Para indicar que no hay ninguna intención siendo considerada en la ejecución del agente, se emplea $T_\iota = \emptyset$. De forma similar para T_ε, T_ρ y demás registros de una configuración.
- Se usa $i, i', ...$ para denotar las intenciones, e $i[p]$ señala la intención que tiene el plan p en su tope.
- Asumimos la existencia de las funciones de selección: S_E para los eventos, S_{Ap} para seleccionar un plan aplicable, S_I para intenciones, y S_M para los mensajes en el buzón.

Para hacer más clara la descripción de las reglas, primero se definen algunas funciones sintácticas. Un plan p tiene la forma $te : ct \leftarrow h$. Usaremos $Head(p)$ para denotar $te : ct$; y $Body(p)$ para apuntar a h. Así mismo, $Label(p) = label$ donde $label$ es una literal que identifica al plan. $TE(p) = te$ y $Ctxt(p) = ct$. Finalmente, $Plan(label) = p$.

Ejemplo 7 *Supóngase el siguiente plan:*

```
@put_label
+!put(X,Y) : true <-  move(X,Y).
```

entonces:

Head(p) = *+!put(X,Y) : true*
Body(p) = *move(X,Y).*
Label(p) = *@put_label*
TE(p) = *+!put(X,Y)*
Ctxt(p) = *true*
Plan(@put_label) = *+!put(X,Y) : true <- move(X,Y).*

Un agente es capaz de recordar, con alguna extensión, las creencias que soportan la adopción de un plan como una intención, así como el resultado de su ejecución, por ejemplo, si la ejecución de un plan falla o tiene éxito. Así es como un agente puede ir recolectando ejemplos de entrenamiento mientras realiza su ejecución normal. Dado un plan p el conjunto de entrenamiento E está dado por:

$$E(p) = \{example(L, Bs, C) \mid Beliefs \models example(L, Bs, C) \wedge L = Label(p)\}$$

donde L es la etiqueta del plan que se está ejecutando, Bs es el conjunto unión del deseo actual del agente y el conjunto de creencias del agente cuando éste seleccionó el plan p como parte de una intención (o el conjunto de creencias del agente cuando ha fallado la ejecución de un plan) y $C \in \{success, failure\}$ es la clase del ejemplo de entrenamiento.

A continuación, se describen las reglas semánticas adaptadas de (Guerra-Hernández et al., 2008b) para incorporar aprendizaje intencional en Jason. Primero, se definen las reglas para recolectar los ejemplos de entrenamiento de aquellos planes supervisados. Una vez que se conoce el resultado de la ejecución del plan, y éste es satisfactorio, se agrega una creencia de la forma $example(L, Bs, success)$.

$$(\mathbf{RecEx_{succ}}) \quad \frac{T_\iota = i[p], Body(p) = \{\}, TE(p) = +!at, ag_{bs} \models at}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag', C, M, T, ProcMsg\rangle}$$

s.t. $L = Label(p) \wedge I = TE(p) \wedge Bs = I \cup ag'_{bs}, ag'_{bs} \models example(L, Bs, success)$.

Las situaciones de fallo incluyen:

1. No hay planes relevantes o aplicables para una sub-meta dada:

$$(\mathbf{RecEx_{failR}}) \quad \frac{T_\varepsilon = \langle +!at, i[p]\rangle, RelPlans(ag_{ps}, +!at) = \{\}}{\langle ag, C, M, T, RelPl\rangle \rightarrow \langle ag', C, M, T, AppPl\rangle}$$

s.t. $L = Label(p) \wedge I = TE(p) \wedge Bs = I \cup ag'_{bs}, ag'_{bs} \not\models executing(L), ag'_{bs} \models example(L, Bs, fail), C'_E = C_E \cup \{\langle +fail(l), i[p]\rangle\}$.

$$(\mathbf{RecEx_{failAp}}) \quad \frac{T_\varepsilon = \langle +!at, i[p]\rangle, AppPlans(ag_{bs}, T_R) = \{\}}{\langle ag, C, M, T, ApplPl\rangle \rightarrow \langle ag', C, M, T, SelInt\rangle}$$

s.t. $L = Label(p) \wedge I = TE(p) \wedge Bs = I \cup ag'_{bs}, ag'_{bs} \not\models executing(L), ag'_{bs} \models example(L, Bs, fail), C'_E = C_E \cup \{\langle +fail(l), i[p]\rangle\}$.

2. Una meta de verificación falla:

$$(\mathbf{RecEx_{failTestGl}}) \quad \frac{T_\iota = i[p], p = te : ctxt \leftarrow ?at; h, Test(ag_{bs}, at) = \{\}}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag', C, M, T, ClrInt\rangle}$$

s.t. $L = Label(p) \wedge I = TE(p) \wedge Bs = I \cup ag'_{bs}, ag'_{bs} \not\models executing(L), ag'_{bs} \models example(L, Bs, fail), C'_E = C_E \cup \{\langle +fail(l), i[p]\rangle\}$.

3. Desde una perspectiva de metas declarativas, un plan diseñado para lograr *at* falla si la ejecución del plan finaliza y el agente no cree *at* (Hübner et al., 2006):

$$(\textbf{RecEx}_{\textbf{fail}}) \quad \frac{T_\iota = i[p], Body(p) = \{\}, TE(p) = +!at, ag_{bs} \not\models at}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag', C, M, T, ClrInt\rangle}$$

s.t. $L = Label(p) \wedge I = TE(p) \wedge Bs = I \cup ag'_{bs}, ag'_{bs} \not\models executing(L), ag'_{bs} \models example(L, Bs, fail), C'_E = C_E \cup \{\langle +fail(l), i[p]\rangle\}$.

Cabe hacer mención que la librería JILDT da únicamente soporte al tercer caso de fallo, cuando un agente finaliza un plan diseñado para alcanzar *at* y no cree *at* al final de la ejecución. Cuando surge una falla en la ejecución de un plan, el agente pone en marcha un proceso de aprendizaje con los ejemplos relacionados con el plan:

$$(\textbf{Learn}_{\textbf{succ}}) \quad \frac{T_\varepsilon = \langle +fail(l), i[p]\rangle, NewCtxt \neq \{\}}{\langle ag, C, M, T, SelEv\rangle \rightarrow \langle ag', C, M, T', RelPl\rangle}$$

s.t. $NewCtxt = Tilde(p, E(p)), ag'_{bs} \not\models E(p), ag'_{ps} = ag_{ps} \setminus p, ag'_{ps} = ag_{ps} \cup TE(p) : NewCtxt \leftarrow Body(p)$

Estas reglas extienden la semántica operacional de Jason para incorporar aprendizaje intencional en Jason. Se considera como trabajo futuro de esta investigación, dar soporte al aprendizaje social en SMA, extendiendo e implementando el resto de los protocolos presentados por Guerra-Hernández et al. (2008b).

3.2. Inducción de árboles de decisión: ID3 y C4.5

La representación por medio de **árboles de decisión** resulta muy natural para los seres humanos. Realizar inducción de árboles de decisión es uno de los métodos más sencillos y exitosos para construir algoritmos de aprendizaje.Un árbol de decisión toma como entrada un ejemplo (un conjunto de atributos que describen un objeto o una situación dada), y devuelve una decisión sobre su pertenencia a una clase determinada. El resultado del algoritmo de aprendizaje es una hipótesis inducida del conjunto de entrenamiento.

Para abordar la representación de la hipótesis, conviene apoyarse en la tarea de **clasificación** debido a que en este trabajo se emplearan árboles lógicos de decisión para representarla, y estos han sido utilizados como clasificadores.

La tarea de clasificación consiste en tomar una decisión de pertenencia con respecto a una situación determinada, teniendo como base la información disponible. En otras palabras, un procedimiento de clasificación consiste en la construcción de un mecanismo aplicable a una secuencia continua de casos, que determina la pertenencia de estos a una clase predefinida, basándose en sus características o atributos. Para poder realizar la clasificación, los árboles de decisión cuentan con la siguiente topología (ver la figura 3.2):

- Un **nodo hoja**, que contiene la salida (el nombre de la clase).
- Un **nodo interno** o nodo de decisión, que contiene la prueba para un atributo y un conjunto de ramas para cada valor posible, las cuales conducen a otro árbol de decisión (que toma en cuenta sólo los atributos restantes).

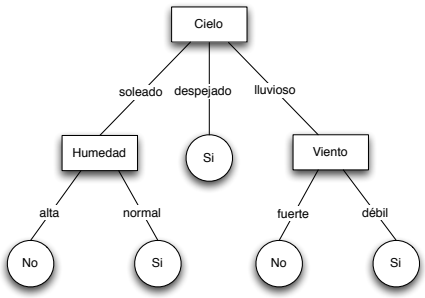


Figura 3.2 Árbol de decisión adaptado de Quinlan (1986).

Los árboles de decisión clasifican los ejemplos al realizar una serie de pruebas sobre los valores de sus atributos. Su mecanismo de operación es el siguiente: se toma un atributo del ejemplo en cada vez, de acuerdo con el valor que ostente el atributo probado, el algoritmo opta por la rama que corresponda a su valor y evalúa el siguiente atributo. La clasificación se realiza repitiendo recursivamente estas pruebas sobre los atributos hasta encontrar un nodo respuesta que indica la clase a la que pertenece el atributo.

Ejemplo 8 *Considérese el ejemplo:* $\langle$***cielo = soleado, temperatura = calor, humedad = alta, viento = débil***$\rangle$ *y el árbol presentado en la figura 3.2. Para clasificar el ejemplo se seguiría una rama del árbol, lo que representa la siguiente sucesión de pruebas (ver la figura 3.3).*

1. *Valor de cielo : soleado*
2. *Valor de humedad : alta*
3. *Conclusión : NO*

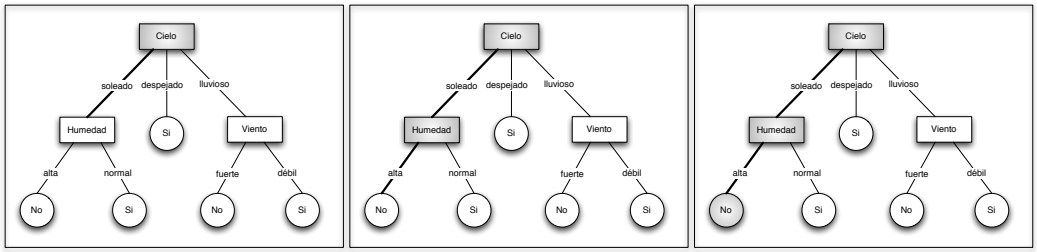


Figura 3.3 Secuencia de pasos para clasificar ⟨cielo = soleado, temp. = calor, hum.= alta, viento = débil⟩.

Por lo tanto, la respuesta para este conjunto de atributos, es la clase NO. De igual manera, cualquier otra evidencia futura podrá ser clasificada de acuerdo con las pruebas de sus atributos establecidas en este árbol.

Un algoritmo clásico para aprender arboles de decisión es el **algoritmo ID3**, propuesto por Quinlan (1986), el cual utiliza una medida de información para guiar la búsqueda en el espacio de árboles de decisión posibles. El Algoritmo 3 muestra una versión de ID3 según lo expuesto en Russell & Norvig (2003).

Algoritmo 3 Algoritmo ID3 adaptado de Tan et al. (2006)

```
1: procedure ID3(E,F)                          ▷ E is a set of examples, F a set of attributes
2:     if stopping_cond(E,F) then              ▷ E.g., All the examples belong to the same class
3:         T ← leaf(classify(E))
4:     else
5:         A ← find_best_split(E,F)
6:         T ← A
7:         V ← { values of attribute A}
8:         for all v ∈ V do
9:             E_v ← {e|v = A and e ∈ E}
10:            T.child ← ID3(E_v,F)
11:        end for
12:    end if
13:    return T                                ▷ The built tree
14: end procedure
```

ID3 infiere el árbol de decisión formándolo desde la raíz hasta las hojas, seleccionando los atributos que mejor separen los datos para incorporarlos a los nodos de prueba. De esta forma, el atributo seleccionado parte los datos en subconjuntos, por cada valor posible del atributo en cuestión, creando un subarbol para cada valor.

El algoritmo determina cual es el atributo que aporta mayor información de acuerdo con una medida, por ejemplo la información esperada o la ganancia de información, y lo sitúa en la raíz del árbol, iniciando el procedimiento de manera recursiva en cada uno de los subárboles. La idea es ir encontrando aquellos atributos que dividan mejor los datos con el objeto de generar un árbol de pocos niveles que sitúe en los primeros nodos aquellos atributos que maximicen (o minimicen) la medida utilizada. Los criterios para decidir el beneficio de un atributo, se determinan mediante una métrica, por ejemplo, **entropía** o la **ganancia de Información** *(Information Gain)* esperada.

Definición 9 (Entropía) *Siendo* k *el número de diferentes etiquetas de clase, m_i el número de valores en el i-ésimo intervalo de una partición, y m_{ij} el número de valores de la clase* j *en el intervalo* i*. Entonces la entropía e_i del intervalo i-ésimo esta dado por la ecuación:*

$$H(i) = -\sum_{i=1}^{k} p_{ij} \log_2 p_{ij} \tag{3.1}$$

donde $p_{ij} = m_{ij}/m_i$ es la probabilidad de la clase j *en el i-ésimo intervalo.*

Por su parte, la ganancia de información es un criterio que puede usarse para determinar la bondad de una partición. Se expresa de la siguiente manera:

$$\Delta_{info}(E,A) = H(E) - \sum_{j=1}^{k} \frac{N(v_j)}{N} H(v_j) \tag{3.2}$$

Donde H(E) es el grado de impureza (entropía) de un nodo dado. N es el número total de registros en los ejemplos E; k es el número de atributos y $N(v_j)$ es el número de registros asociados con el nodo v_j.

Una característica de esta métrica es su tendencia a favorecer atributos con muchos valores, de tal forma que éstos predominan sobre otros con pocos valores, pero cuyo beneficio es mayor. Para evitar esto, se emplea la métrica *Gain Ratio* que evalúa la entropía del conjunto de entrenamiento con respecto a los valores de un atributo. *Gain Ratio* se calcula a partir de la **Ganancia Máxima**:

$$GM = -\sum_{E_i \in E} \frac{E_i}{E} \log \frac{E_i}{E} \tag{3.3}$$

Entonces, el coeficiente ***Gain Ratio*** es el cociente de la Ganancia de información y la Ganancia Máxima:

$$GR = \frac{\Delta_{info}}{GM} \tag{3.4}$$

Años después de ID3 surge el algoritmo C4.5 (Quinlan, 1993), que al igual que ID3 es recursivo y se basa en la estrategia *divide y vencerás*. El núcleo del algoritmo es el mismo que en el ID3 con algunas mejoras.

- Incorporación de atributos tanto discretos como continuos.
- Utilización de la métrica *gain ratio*.
- Método de post-poda, para evitar sobreajuste.
- Método probabilístico para solucionar el problema de los atributos con valor desconocido.

3.3. Programación lógica inductiva

La Hipótesis del Aprendizaje Inductivo (Definición 8) mostrada en la sección 3.1 postula que se puede aprender una función objetivo a partir de datos observados, de manera que un sistema puede contender exitosamente con nuevas evidencias (no observadas), utilizando la función objetivo aproximada en el proceso de aprendizaje. No obstante, conviene resaltar el hecho de que el aprendizaje no parte siempre de cero, sino que con frecuencia se tiene un **conocimiento general (CG)**, el cual es relevante para la tarea que se requiere aprender. Retomando esta idea dentro del marco de la lógica, S. Muggleton & de Raedt (1994) introducen un nuevo campo denominado **Programación Lógica Inductiva (PLI)**, definida como la intersección entre el Aprendizaje Automático y la Programación Lógica (que conjunta la expresión de los datos en lógica de primer orden y métodos de inferencia).

En primer lugar, hay que señalar que el uso de lógica de primer orden confiere las siguientes ventajas (Nenhuys-Chen & de Wolf, 1997):

- Disponibilidad de un conjunto de conceptos, técnicas y resultados que han sido bien estudiados y entendidos.
- Establecer una uniformidad en la representación (en un lenguaje clausal), tanto para el CG, como de la hipótesis por aprender (también denominada teoría inducida) y del conjunto de entrenamiento.
- Facilidad para interpretar y entender la hipótesis inducida por el sistema de aprendizaje.

El conjunto de entrenamiento se divide en dos partes, ejemplos positivos E^+ y negativos E^-. Una característica de los algoritmos de PLI es la corrección de la teoría inducida (Σ). De modo informal, se dice que una teoría aprendida es correcta si es completa (tanto Σ como CG validan todos los ejemplos de E^+), y si es consistente (ni Σ ni CG implican ejemplos de E^-). Con base en lo anterior, se presenta la configuración normal o explicativa de la tarea de PLI (Nenhuys-Chen & de Wolf, 1997), enunciada como se muestra enseguida:

Definición 10 (Configuración normal del problema de PLI.) *A partir de un conjunto finito CG de cláusulas (que puede estar vacío) y un conjunto de ejemplos positivos E^+ y negativos E^-; encontrar una teoría Σ, tal que $\Sigma \cup CG$ sea correcta con respecto a E^+ y E^-.*

Un aspecto que hay que señalar en PLI es que existen conjuntos E^+ y E^- para los cuales no hay una teoría que sea correcta. Esto debido a que:

1. Puede ser que $\Sigma \cup CG$ sea inconsistente con respecto a los ejemplos negativos, por ejemplo cuando un ejemplo es positivo y negativo al mismo tiempo.
2. El problema en cuestión tenga un número infinito de ejemplos, lo que ocasiona que existan más ejemplos que teorías, por lo que no habrá una teoría que cubra todos los ejemplos.

Otra posibilidad se presenta cuando la teoría encontrada no tiene poder predictivo, es decir, sólo se ajusta a los ejemplos pertenecientes a E^+ y a nada más, lo cual es opuesto al cometido del Aprendizaje Automático. Una forma de minimizar este efecto es añadir restricciones a la teoría, lo cual será posible dependiendo de la tarea a resolver, como acotar el numńero de cláusulas que deba contener la teoría con respecto a la cardinalidad del conjunto de ejemplos (Nenhuys-Chen & de Wolf, 1997).

Ahora bien, tomando en cuenta la existencia de una o más teorías correctas para los conjuntos de ejemplos positivos y negativos que se tengan, el aprendizaje consiste en la búsqueda de la teoría correcta de entre el universo de cláusulas permitidas (el espacio de búsqueda). Para realizar esta búsqueda es de gran importancia la existencia de un orden en dicho espacio, pues permite efectuar una revisión sistemática de las cláusulas. Tal ordenamiento puede establecerse, mediante la especificidad de las hipótesis buscadas, esto es, considerando la búsqueda de hipótesis de la más general a la más específica o viceversa. Así, existen dos formas principales de realizar esta búsqueda: Descendente *(Top-Down)* o Ascendente *(Bottom-up)*, seguń comiencen con una teoría demasiado general -que valida cualquier ejemplo-, o demasiado específica -que no valida ningún ejemplo-, respectivamente. La siguiente sección describe la configuración basada en interpretaciones, la cual está estrechamente relacionada con la representación del conjunto de entrenamiento.

3.3.1. Sistemas basados en interpretaciones

La inducción de hipótesis a partir de evidencias ha sido tratada desde dos paradigmas: **la Representación Proposicional** y la **Representación en Primer Orden** (Blockeel, 1998). Estas dos formas de expresar la información determinan las posibilidades del algoritmo de aprendizaje. En este mismo sentido, la representación en primer orden da lugar a dos vertientes dentro de la PLI: **el Aprendizaje Inductivo por Implicación** *(Learning from Entailment)* y el **Aprendizaje Inductivo Basado en Interpretaciones** *(Learning from Interpretations)*.

El cometido de los sistemas en PLI puede verse como la obtención de generalizaciones a partir de un **Conjunto de Entrenamiento (E)** y conocimiento general (CG) relevante para el dominio en el que se realice el aprendizaje. Es precisamente la forma en que se conciben E y CG lo que hace la diferencia entre las dos configuraciones

de aprendizaje mencionadas. El Aprendizaje Inductivo por Implicación (AII) es el paradigma más utilizado en la PLI y se orienta a la obtención de hipótesis partiendo de E y CG. Tomando como base la configuración normal del problema de PLI mostrado en la definición 10, el Aprendizaje Inductivo por Implicación se expresa formalmente como sigue (Blockeel, 1998):

Definición 11 (Aprendizaje Inductivo por Implicación) *A partir de un conjunto de ejemplos positivos E^+, un conjunto de ejemplos negativos E^-, conocimiento general (CG) y un lenguaje de primer orden: $L \subseteq Prolog$: Encontrar una hipótesis $H \subseteq L$, tal que: $\forall e \in E^+ : H \wedge B \models e$ y $\forall e \in E^- : H \wedge B \not\models e$*

Lo importante a resaltar de esta configuración es el uso de la totalidad de los ejemplos E junto con el CG para definir la hipótesis. Sin embargo, no toda la información contenida en E es relevante y no está definida cuál parte si lo es, por lo que es necesario buscarla en todo el universo de datos, lo que origina un alto costo computacional. Desde una perspectiva de implementación, tanto E como CG pueden verse como un sólo programa en Prolog, donde cada ejemplo es un hecho, por lo tanto, se pueden aprender relaciones recursivas.

Por otro lado, el Aprendizaje Inductivo Basado en Interpretaciones (AIBI) es una configuración alternativa, que contrariamente al Aprendizaje Inductivo por Implicación, toma como base la noción de que la información relevante para cada ejemplo está localizada, sólo en una parte de los datos, por lo que no es necesario considerar todo el conjunto. De esta forma, se asume que cada ejemplo es independiente de los demás, y proporciona la información necesaria para aprender un concepto, teniendo como consecuencia la imposibilidad de aprender definiciones recursivas. Esto se conoce como el supuesto de localidad, el cual se enuncia a continuación:

Definición 12 (Supuesto de localidad) *Toda la información relevante para un solo ejemplo está contenida en una pequeña parte de la base de datos.*

En el Aprendizaje Inductivo Basado en Interpretaciones, el conocimiento general se representa como un programa en Prolog y cada ejemplo $e \in E$, es representado por un programa en Prolog separado que incluye una etiqueta $c \in Clases(+,-)$. Cada ejemplo expresa un conjunto de hechos fundamentados, esto es, un modelo mínimo de Herbrand, para el cual se cumple $e \wedge CG$, a esto se le denomina interpretación (Blockeel, 1998). Formalmente, el Aprendizaje Inductivo Basado en Interpretaciones se define de la siguiente manera:

Definición 13 (Aprendizaje Inductivo Basado en Interpretaciones) *A partir de una variable objetivo C, un conjunto E de ejemplos etiquetados con un valor $c \in C$, conocimiento general (CG) y un lenguaje $L \subseteq Prolog$ Encontrar: una hipótesis $H \in L$ tal que para todo ejemplo $(e,c) \in E$:*
$H \wedge e \wedge B \models etiqueta(c)$ *y* $\forall c' \neq c : H \wedge e \wedge B \not\models etiqueta(c)$.

Blockeel (1998) señala las siguientes ventajas de esta configuración de aprendizaje:

- La información contenida en los ejemplos es separada del conocimiento general.
- Explota el supuesto de localidad, la información de los ejemplos es independiente entre sí.
- Consecuencia del punto anterior es considerar que los ejemplos provienen de una población (son una muestra) y no agotan la descripción de ella, por lo que debe tomarse en cuenta ruido y valores faltantes. Esto conduce a las siguientes proposiciones:
  - La descripción de los ejemplos es cerrada, autocontenida.
  - La descripción de la población es abierta, en presencia de más evidencia, se tendrá mayor conocimiento.

En el cuadro 3.1 se sintetizan las diferencias entre los tipos de aprendizaje que se han mencionado hasta el momento: Proposicional, Inductivo por Implicación e Inductivo Basado en Interpretaciones.

Paradigma	**Información**	**Supuesto de localidad**	**Descripcion de** $e \in E$
Proposicional	$\{V_i, ..., V_n\}$ Donde: $V_x = [valor\_de\_atributo, Etiqueta]$	Si	Cerrada
AII	$\{E^+, E^-, ConocimientoBase\}$	No	Abierta
AIBI	$\{\{\{Interpret_1\}, ..., \{Interpret_n\}\}, \{ConocimientoBase\}\}$	Si	Cerrada

Cuadro 3.1 Diferencias en los tipos de aprendizaje, proposicional, por implicación y por interpretaciones (Blockeel, 1998).

En lo que respecta al supuesto de localidad y a la apertura en cuanto a la descripción de la población origen de los datos, el Aprendizaje Inductivo Basado en Interpretaciones puede considerarse como una configuración intermedia entre los aprendizajes proposicional y por implicación, aunque con la limitación de no poder aprender definiciones recursivas, sin embargo, éstas no tienen una presencia preponderante en las aplicaciones prácticas. En consecuencia, utilizarlo pone al alcance del algoritmo de aprendizaje el poder expresivo de la Representación en Primer Orden y la apertura de la información. (Figura 3.4).

3.3.2. Árboles Lógicos de Decisión

En la sección 3.2, se describió el uso de árboles de decisión para expresar información proposicional. En esta sección presenta su versión en primer orden: los **Árboles Lógicos de Decisión (ALD)**.

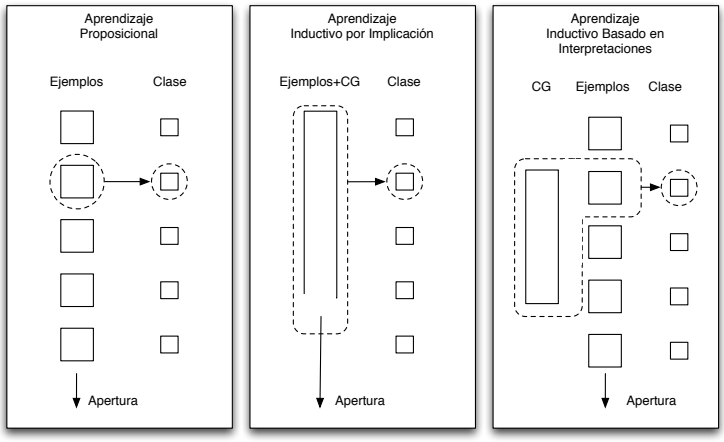


Figura 3.4 Comparación gráfica entre los aprendizajes proposicional, por implicación y basado en interpretaciones, con respecto al supuesto de localidad y la apertura de la descripción de la población de los datos. Adaptado de (Blockeel, 1998).

Los árboles de decisión en primer orden o árboles lógicos de decisión son árboles binarios que representan una conjunción de literales. Cada nodo a su vez es una conjunción de literales y cada rama completa es la conjunción completa de pruebas que hay que realizar para encontrar el valor de la etiqueta. Pueden ser utilizados para realizar clasificación y regresión. Formalmente se definen de la siguiente manera:

Definición 14 (Árboles Lógicos de Decisión) *Un árbol de decisión en primer orden, o árbol lógico de decisión, es un árbol binario constituido por dos elementos: nodos hoja y nodos de prueba, denotados como hojas y nodos internos, respectivamente. Estos mantienen las siguientes particularidades:*

- *Cada nodo contiene una conjunción de literales.*
- *Distintos nodos pueden compartir variables únicamente por la rama izquierda.*

De esta manera, un ALD tiene la siguiente morfología:
$T = hoja(L)$ *Donde L= una etiqueta de respuesta.*
$T = nodo(conj, \{(verdadero, izq)\}, \{(falso, der)\})$

El nodo representa la prueba a realizar sobre los datos. La rama izquierda se sigue cuando la conjunción conj *se hace verdadera. La rama derecha es relevante sólo cuando* conj *falla (por ello no comparte variables con los nodos que la anteceden).*

En la Figura 3.5 se muestra un ALD, el cual equivale a la condición para colocar un bloque X sobre un bloque Y. Si se tiene la intención de poner un bloque X sobre un bloque Y, y ambos bloques están libres, la ejecución es satisfactoria *succ*, en caso contrario la ejecución es fallida $fail$.

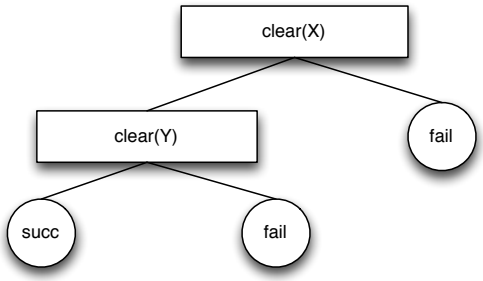
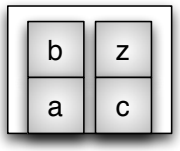


Figura 3.5 Árbol Lógico de Decisión.

Ejemplo 9 *Considérese la intención de poner el bloque* b *sobre el bloque* c*, la siguiente percepción del ambiente:*



y el árbol presentado en la figura 3.5; la sucesión de pruebas indicadas por las literales del árbol sobre el ejemplo sería como se muestra enseguida (ver la figura 3.6):

1. *clear(b)? si, éxito* $\Rightarrow$ *ir por la rama izquierda;*
2. *clear(b)* $\wedge$ *clear(c)? no, fallo* $\Rightarrow$ *ir por la rama derecha;*
3. *leaf(fail)* $\Rightarrow$ *clase :* ***fail***

En este caso, la clase asignada tiene la etiqueta $fail$, y se obtuvo tras probar la conjunción $clear(b) \wedge clear(c)$, donde clear(c) *resultó falsa, de acuerdo a la evidencia.*

De esta manera, al utilizar árboles lógicos de decisión, se consigue aprovechar el poder expresivo de la Representación en primer orden y la facilidad de interpretación de los árboles de decisión.

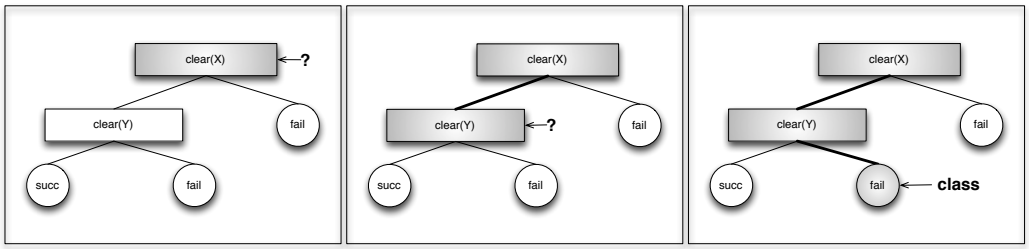


Figura 3.6 Secuencia de pasos para clasificar put(b,c), con la configuración de bloques: ⟨on(b,a), on(a,table), on(c,table), on(z,c)⟩ en un ALD.

Los árboles lógicos de decisión pueden derivarse a partir de un conjunto de entrenamiento, siguiendo un procedimiento de aprendizaje semejante al utilizado para inducir árboles proposicionales, aumentándole las características necesarias para el uso de la representación en Primer Orden. Lo anterior se conoce como un escalamiento del algoritmo de inducción (por ejemplo de ID3). Tal es el caso del algoritmo TILDE (Blockeel et al., 1999), del cual se hablará en el siguiente capítulo.

3.4. Resumen

El aprendizaje, visto desde un contexto de agencia se entiende como la adquisición de conocimientos y habilidades de un agente, en el que dicha adquisición es obtenida por sí mismo y le permite lograr un mejor rendimiento en sus actividades futuras. En términos de Intencionalidad, la noción de aprendizaje Intencional en este trabajo esta fuertemente ligada a la teoría de racionalidad práctica de (Bratman, 1987), donde el objetivo del proceso de aprendizaje esta orientado hacia las razones que debe considerar un agente para adoptar una intención.

La representación por medio de árboles de decisión es uno de los métodos más sencillos y exitosos para construir algoritmos de aprendizaje. Un árbol de decisión toma como entrada un conjunto de atributos que describen un objeto o una situación dada, y devuelve una decisión sobre su pertenencia a una clase determinada. El resultado del algoritmo de aprendizaje es una hipótesis inducida del conjunto de entrenamiento. Sin embargo una representación *atributo-valor*, como es usada en ID3 o C4.5, no es la manera más conveniente de representar el conocimiento en sistemas multiagente BDI. Por ello, se hace uso de los árboles lógicos de decisión, la versión en primer orden de los árboles de decisión.

Los árboles lógicos de decisión son árboles binarios que representan una conjunción de literales. Cada nodo a su vez es una conjunción de literales y cada rama completa es la conjunción completa de pruebas que hay que realizar para encontrar el valor de la etiqueta. El contexto de un plan puede ser representado por una de estas ramas, debido a la naturaleza conjuntiva de éstas; mientras que la disyunción de las ramas de un árbol lógico de decisión, permite definir más de un plan, con contextos diferentes.

Una representación proposicional no es suficiente para sustentar el aprendizaje en sistemas multiagente BDI. Las representaciones de información basados en interpretaciones resultan ser convenientes como soporte de aprendizaje, debido a que estas interpretaciones pueden extraerse del estado mental de los agentes BDI.

Parte II

Desarrollo

Capítulo 4
Análisis y diseño

JILDT (Jason Induction of Logical Decision Trees) es una librería de aprendizaje desarrollada en el lenguaje de programación Java, bajo el paradigma de programación orientada a agentes *AgentSpeak(L)*, a través de su intérprete *Jason*. Esta librería permite sustentar aprendizaje intencional en sistemas multiagentes BDI a través de la inducción de árboles lógicos de decisión, mismos que han sido introducidos en la sección 3.3.2.

Como se mencionó en el capítulo introductorio, el principal objetivo de este proyecto es extender JILDT hacia un nivel de aprendizaje *AgentSpeak(L)*, para ello es necesario diseñar una clase de agente que extienda las capacidades de un agente convencional para ejecutar un mecanismo de aprendizaje después de la ejecución fallida de un plan, y de esta manera mejorar la adaptabilidad a su entorno.

En este capítulo, se presenta un análisis *grosso modo* del modo de operación de la librería de aprendizaje JILDT. La primera sección describe un algoritmo de inducción de árboles lógicos de decisión, el cual ha resultado el mecanismo idóneo para sustentar aprendizaje en sistemas multiagentes BDI. La segunda sección propone el diseño general de una clase de agente aprendiz. Los agentes aprendices modularizan sus creencias, de modo que se puedan separar las creencias relacionadas con el aprendizaje de las creencias relacionadas con el objetivo principal del agente.

4.1. Inducción de Árboles Lógicos de Decisión

La inducción de árboles lógicos de decisión, o TILDE por sus siglas en inglés (*Top-Down Induction of Logical Decision Trees*), es un mecanismo idóneo para sustentar aprendizaje en el contexto de agentes Intencionales BDI, principalmente gracias a que las entradas requeridas para este método son fácilmente obtenidas del estado mental de los agentes, dada la naturaleza en primer orden de sus enunciados; además, cada recorrido del nodo raíz a una

hoja corresponde a una conjunción de literales de primer orden, es decir, el tipo de representación necesaria para definir el contexto de un plan (ver la figura. 4.1 (b)). En sí, un árbol es la disyunción de estas conjunciones, lo que nos permite tener más de un plan con diferentes contextos . Estos árboles se pueden emplear para expresar hipótesis sobre las ejecuciones satisfactorias o fallidas de intenciones.

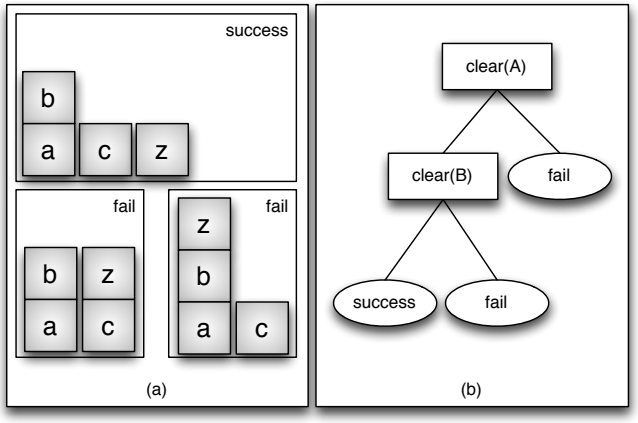


Figura 4.1 (a) Ejemplos de entrenamiento para el mundo de los bloques, cuando un agente intenta colocar el bloque *b* sobre el bloque *c*. (b) Árbol lógico de decisión, formado con los ejemplos a su izquierda.

El sistema ACE/TILDE, desarrollado en la Universidad de Leuven, Bélgica por Blockeel et al. (1999), ejecuta una búsqueda en profundidad y utiliza interpretaciones para expresar el conjunto de entrenamiento y árboles lógicos para representar la hipótesis. Para computar un árbol lógico de decisión se requieren tres archivos de entrada:

- El conocimiento general del agente *(background.bg)*.
- La base de conocimiento, *(kwnoledgeBase.kb)*
- Las configuraciones y el sesgo del lenguaje *(settings.s)*.

El **conocimiento general** (*background file*) es aquel que el agente tiene sobre su dominio de experiencia, y está formado por las reglas que el agente cree. La **base de conocimiento** (*knowledge base file*) se refiere al conjunto de ejemplos de entrenamiento reunidos por el agente, donde cada ejemplo de entrenamiento es conocido como **modelo**. Para fines de esta investigación, cada modelo está compuesto por el conjunto de creencias que el agente tiene al momento de adoptar una intención (para ejemplos etiquetados como satisfactorios -*success*-), o el conjunto de creencias que el agente tiene al momento de fallar la ejecución de un plan (para ejemplos etiquetados como fallidos -*fail*-); una literal indicando el deseo actual del agente; y una etiqueta que indica si la ejecución de la intención fué exitosa o fallida. En el cuadro 4.1 se muestran los modelos correspondientes a los ejemplos

en la figura 4.1(a). La clase de los ejemplos se introduce en la línea 2, y el deseo asociado en la línea 3. El resto del modelo corresponde a las creencias del agente al momento de adoptar la intención o fallar la ejecución de la misma.

```
1 begin(model(1))
2   success.
3   intend(put(b,c)).
4   on(b,a).
5   on(a,table).
6   on(c,table).
7   on(z,table).
8 end(model(1))
```

```
1 begin(model(2))
2   fail.
3   intend(put(b,c)).
4   on(b,a).
5   on(a,table).
6   on(z,c).
7   on(c,table).
8 end(model(2))
```

```
1 begin(model(3))
2   fail.
3   intend(put(b,c)).
4   on(b,a).
5   on(a,table).
6   on(z,b).
7   on(c,table).
8 end(model(3))
```

Cuadro 4.1 Ejemplos de entrenamiento de la figura 4.1(a) como modelos de TILDE. Etiquetas de clase en la línea 2.

Por último, el archivo de configuraciones (*settings file*) permite personalizar el mecanismo de aprendizaje y definir el **sesgo del lenguaje**, el cual especifica las literales que deben ser consideradas como candidatos para ser incluidos en el árbol lógico de decisión. El sesgo del lenguaje se define por directivas *rmode* y directivas *lookahead*. Las directivas **rmode** indican que su argumento debe ser considerado como un candidato para formar parte del árbol, y tienen la forma `rmode(conj)`, donde *conj* tiene aquellos indicadores posibles (definidos por el usuario) para las literales que deben añadirse y las variables que deben ocurrir en ellas. En el cuadro 4.2 se especifican los tipos de *rmode* utilizados en este trabajo.

Tipo	Formato	Descripción
Entrada	rmode(literal(+A))	Las variables de *A* deben ocurrir ya en la conjunción asociada.
Salida	rmode(literal(-A))	Las variables de *A* no ocurren en la conjunción asociada. Se generan variables nuevas.
Entrada / Salida	rmode(literal(+-A))	*A* toma tanto variables que ya ocurren en la conjunción asociada como variables nuevas.

Cuadro 4.2 Descripción de las variables ocurridas en los operadores *rmode*.

Las directivas **lookahead** indican que la conjunción en su argumento debe ser considerada como un candidato también. Esta directiva permite vincular lógicamente las variables en el plan original con las variables de las literales que son candidatas a formar parte del árbol. El cuadro 4.3 muestra un ejemplo del sesgo del lenguaje formado por directivas rmode y lookahead.

```
1 rmode(clear(+V1)).
2 rmode(on(+V1,+V2)).
3 rmode(on(+V2,+V1)).
4 rmode(intend(put(-V1,-V2))).
5 lookahead(intend(put(V1,V2)),clear(v1)).
6 lookahead(intend(put(V1,V2)),clear(v2)).
7 lookahead(intend(put(V1,V2)),on(v1,V2)).
8 lookahead(intend(put(V1,V2)),on(v2,V1)).
```

Cuadro 4.3 Ejemplos de directivas que forman el sesgo de lenguaje.

Ejemplo 10 *Supóngase el sesgo del lenguaje mostrado en el cuadro 4.3, y la consulta inicial $Q = [true]$, se generarán los siguientes candidatos:*

$$\rho(\leftarrow [true]) = \begin{cases} \leftarrow intend(put,(A,B)); \\ \leftarrow intend(put(A,B)),clear(A); \\ \leftarrow intend(put(A,B)),clear(B); \\ \leftarrow intend(put(A,B)),on(A,B); \\ \leftarrow intend(put(A,B)),on(B,A) \end{cases}$$

Estos candidatos son generados debido a que los modos de los enunciados rmode para los funtores clear y on (Cuadro 4.3, líneas 1-3), requieren que existan variables declaradas, las cuales son introducidas por el enunciado rmode(intend(put(-V1,-V2))) y los enunciados lookahead. Supóngase ahora que el candidato que maximizó la métrica de ganancia es [intend(put(A,B)), clear(A)], se generarán los siguientes candidatos:

$$\rho(\leftarrow [true, intend(put,A,B), clear(A)]) = \begin{cases} \leftarrow clear(A); \leftarrow clear(B); \\ \leftarrow on(A,A); \leftarrow on(A,B); \leftarrow on(B,A); \leftarrow on(B,B); \\ \leftarrow intend(put(C,D)); \\ \leftarrow intend(put(C,D)),clear(C); \leftarrow intend(put(A,B)),clear(D); \\ \leftarrow intend(put(C,D)),on(C,D); \leftarrow intend(put(C,D)),on(D,C) \end{cases}$$

Aquí, ya hay variables introducidas en la consulta, por lo que se pueden generar candidatos con los enunciados rmode donde el funtor es on o clear. En caso de que el candidato clear(B), maximice la métrica de ganancia, se puede generar un árbol como el que se muestra es la figura 4.1.

Como salida, el sistema TILDE regresa un archivo con extensión *(*.out)*, el cual contiene entre otra información, un árbol lógico de decisión y su programa Prolog equivalente (ver cuadro 4.4).

```
1  Compact notation of tree:
2
3  intend(put(-A,-B)),clear(B) ?
4  +--yes: [succ] 4.0 [[succ:3.0,fail:1.0]]
5  +--no:  [fail] 2.0 [[succ:0.0,fail:2.0]]
6
7  Equivalent prolog program:
8
9  class([succ]) :- intend(put(A,B)),clear(B), !.
10 % 3.0/4.0=0.75
11 class([fail]).
12 % 2.0/2.0=1.0
```

Cuadro 4.4 Parte del archivo .out, resultado de la ejecución del sistema ACE/TILDE.

4.1.1. Algoritmo de Inducción

Los árboles lógicos de decisión se computan recursivamente, como lo hacen el algoritmo ID3 (Quinlan, 1986) y C4.5 (Quinlan, 1993), con la diferencia de que los nodos de un árbol ID3 o C4.5 representan los valores que pueden tomar los atributos, mientras que los nodos de un árbol lógico de decisión son conjunciones lógicas. El algoritmo 4 describe como son computados estos árboles lógicos de decisión:

1. Dada una consulta inicial, $Q = true$ y un conjunto de ejemplos etiquetados E, se computa un conjunto de candidatos $\rho(Q)$ con base en Q y el sesgo del lenguaje (*language bias*). El candidato que maximice una métrica de ganancia, por ejemplo *gain ratio* (Eq. 4.4) es seleccionado como nodo del árbol (línea 2).
2. El procedimiento se detiene cuando un criterio de parada es alcanzado (línea 4), por ejemplo, el coeficiente *gain ratio* no es mejorado por el candidato.
3. Si este es el caso, se regresa una hoja con la etiqueta del valor de clase que más se repita en el conjunto de ejemplos. Como trabajo futuro, queda la posibilidad de explorar nuevo métodos para asignar la etiqueta de clase a un nodo hoja.
4. De otro modo, se computa un nodo interno, en el que el contenido del nodo interno $Conj$ se forma con el candidato que maximice su métrica de ganancia (línea 2). Q_b es la unión entre la consulta inicial Q y $Conj$ (línea 3). El conjunto de ejemplos E se divide (líneas 7-8) en aquellos ejemplos que satisfacen Q_b, denotados por E_1 y aquellos que no satisfacen Q_b, denotados por E_2; para continuar, el procedimiento es ejecutado recursivamente para construir las ramas izquierda y derecha del nodo interno (líneas 9-11). El nodo izquierdo se construye a partir de los ejemplos de entrenamiento que satisfacen Q_b, y la consulta Q_b; el nodo derecho se construye a partir de los ejemplos de entrenamiento que no satisfacen Q_b, y la consulta inicial Q. Al final, el procedimiento regresa el árbol construido T.

Algoritmo 4 Inducción de Árboles Lógicos de Decisión.

```
1:  procedure BUILDTREE(E,Q)                          ▷ E is a set of examples, Q a query
2:      ← Conj := best(ρ(← Q))                        ▷ best max information gain
3:      ← Q_b := Q ∪ Conj
4:      if stopCriteria(← Q_b)) then                  ▷ E.g., No gain ratio obtained
5:          T := leaf(majority_class(E))
6:      else
7:          E_1 := {e ∈ E | e ∧ B ⊨ Q_b}
8:          E_2 := {e ∈ E | e ∧ B ⊭ Q_b}
9:          Left := buildTree(E_1, Q_b)
10:         Right := buildTree(E_2, Q)
11:         T := innerNode(Conj, Left, Right)
12:     end if
13:     return T;                                     ▷ The built tree
14: end procedure
```

El mejor candidato se selecciona (Algoritmo 5) de la siguiente manera: Cada candidato $r_i \in \rho(\leftarrow Q)$ induce una partición en los ejemplos en E; se computa una medida de calidad usando la métrica *Gain Ratio* (Eq. 4.4) (Quinlan, 1993). Para esto, una matriz *counter* (ver figura 4.2) almacena el número de ejemplos satisfactorios o fallidos para cada clase $c_i \in C$ (líneas 6-9). Se calcula su valor *Gain Ratio*. El candidato seleccionado es aquel que maximiza la métrica *Gain Ratio*(línea 13-17).

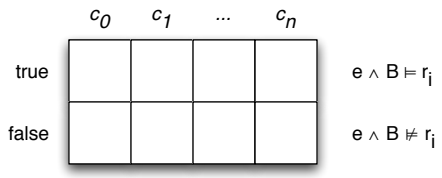


Figura 4.2 Matriz *counter* que almacena el número de ejemplos satisfactorios o fallidos para cada clase c.

Gain Ratio es una métrica de información basada en entropía, la cual es una medida de incertidumbre asociada a un evento; la entropía de un conjunto de ejemplos E está dada por la ecuación 4.1 (Shannon, 1948).

$$s(E) = -\sum_{i=1}^{k} p(c_i, E) \log_2 p(c_i, E) \tag{4.1}$$

donde k es el número de clases $c_i \in C$ y $p(c_i, E)$ es la proporción de ejemplos en E que pertenecen a la clase c_i.

Algoritmo 5 Selección del mejor candidato

```
1:  procedure BEST(E,R)                          ▷ E is a set of examples, R = ρ(← Q) refinements of Q
2:      Max := 0; Best := true;                  ▷ Initialize variables
3:      for all r ∈ R do
4:          counter := {0,0}                     ▷ Initialize counter matrix
5:          for all e ∈ E do
6:              if e ∧ B ⊨ r_i then
7:                  counter[0][class(e)]++;
8:              else
9:                  counter[1][class(e)]++;
10:             end if
11:         end for
12:         Gr := gainRatio(E, counter);
13:         if Gr > Max then
14:             Max = Gr; Best = r;
15:         end if
16:     end for
17:     return Best;
18: end procedure
```

Como estamos interesados en el refinamiento de candidatos r_c que reducen la entropía de los ejemplos, se introduce el cálculo de la métrica de ganancia de información, la cual es computada en la ecuación 4.2.

$$infoGain(E, counter) = s(E) - \sum_{c \in C} \frac{\sum_{v \in \{success, fail\}} counter[v][c]}{|E|} s(E_c) \tag{4.2}$$

donde *counter* es la matriz computada en el algoritmo 5 y $E_{r_c} \subseteq E$ es la partición por clase de los ejemplos en E. La ganancia máxima que un candidato r_i puede obtener está dada por la ecuación 4.3.

$$maxGain(E) = -\sum_{E_{r_i} \subseteq E} \frac{|E_{r_i}|}{|E|} \log_2 \frac{|E_{r_i}|}{|E|} \tag{4.3}$$

La métrica *Gain Ratio* penaliza a los atributos que generan muchas particiones, por ejemplo atributos de tipo *ID*, y se computa por la ecuación 4.4.

$$gainRatio(E, counter) = \frac{infoGain(E, counter)}{maxGain(E)} \tag{4.4}$$

Ejemplo 11 *Supóngase que se quiere calcular el valor de Gain Ratio de* $Q_i = [intend(put(X,Y)), clear(X)]$ *con los ejemplos presentados en el cuadro 4.5.*

```
begin(model(1)).
success.
intend(put(b,c)).
on(z,table).
on(b,a).
on(c,table).
on(a,table).
end(model(1)).
```

```
begin(model(2)).
success.
intend(put(b,c)).
on(b,a).
on(z,table).
on(c,table).
on(a,table).
end(model(2)).
```

```
begin(model(3)).
success.
intend(put(b,c)).
on(b,a).
on(z,table).
on(c,table).
on(a,table).
end(model(3)).
```

```
begin(model(4)).
fail.
intend(put(b,c)).
on(z,b).
on(b,a).
on(c,table).
on(a,table).
end(model(4)).
```

```
begin(model(5)).
success.
intend(put(b,c)).
on(z,table).
on(b,a).
on(c,table).
on(a,table).
end(model(5)).
```

```
begin(model(6)).
fail.
intend(put(b,c)).
on(z,c).
on(b,a).
on(c,table).
on(a,table).
end(model(6)).
```

```
begin(model(7)).
success.
intend(put(b,c)).
on(z,table).
on(c,table).
on(b,a).
on(a,table).
end(model(7)).
```

```
begin(model(8)).
fail.
intend(put(b,c)).
on(z,c).
on(c,table).
on(b,a).
on(a,table).
end(model(8)).
```

Cuadro 4.5 Ejemplo de modelos para un agente aprendiz.

La matriz Counter *quedará formada de la siguiente manera:*

	succ	***fail***	***Total***
true	*5*	*1*	*6*
false	*0*	*2*	*2*
Total	*5*	*3*	*8*

Primero, se calcula la entropía del conjunto de ejemplos empleando la ecuación 4.1. Se busca la incertidumbre basándose en cuantos ejemplos satisfacen Q_i *(6) y cuantos no (2).*

$$\begin{aligned} s(E) &= -(\tfrac{6}{8}\log_2\tfrac{6}{8}) - (\tfrac{2}{8}\log_2\tfrac{2}{8}) \\ &= -(0{,}75\log_2 0{,}75) - (0{,}25\log_2 0{,}25) \\ &= -(0{,}75 \times -0{,}415037) - (0{,}25 \times -2) \\ &= 0{,}811277 \end{aligned}$$

Después, se calcula la ganancia de información usando la ecuación 4.2, tomando como referencia el total de ejemplos de la clase succ *(5) y la clase* fail *(3).*

$$\begin{aligned} infoGain(E,counter) &= 0{,}811277 - (\tfrac{5}{8}(-(\tfrac{5}{5}\log_2\tfrac{5}{5}) - (\tfrac{0}{5}\log_2\tfrac{0}{5})) + \tfrac{3}{8}(-(\tfrac{1}{3}\log_2\tfrac{1}{3}) - (\tfrac{2}{3}\log_2\tfrac{2}{3}))) \\ &= 0{,}811277 - (\tfrac{5}{8}(0-0) + \tfrac{3}{8}(-(0{,}333333 \times -1{,}584962) - (0{,}666666 \times -0{,}584962))) \\ &= 0{,}811277 - (\tfrac{3}{8}(-(-0{,}528320) - (-0{,}389974))) \\ &= 0{,}811277 - (\tfrac{3}{8}(0{,}528320 + 0{,}389974)) \\ &= 0{,}466917 \end{aligned}$$

La ganancia máxima se calcula usando la ecuación 4.3:

$$\begin{aligned} maxGain(E) &= -(\tfrac{6}{8}\log_2\tfrac{6}{8}) - (\tfrac{2}{8}\log_2\tfrac{2}{8}) \\ &= -(0{,}75\log_2 0{,}75) - (0{,}25\log_2 0{,}25) \\ &= -(0{,}75 \times -0{,}415037) - (0{,}25 \times -2) \\ &= 0{,}811277 \end{aligned}$$

Por último, el coeficiente Gain Ratio está dado por la ecuación 4.4.

$$\begin{aligned} gainRatio(E,counter) &= \tfrac{0{,}466917}{0{,}811277} \\ &= 0{,}575533 \end{aligned}$$

En la siguiente sección se explica el diseño general de una clase de agente aprendiz *(Learner)*, el cual extiende las capacidades de un agente por defecto de Jason para incorporar el uso de árbolos lógicos de decisión como mecanismo de aprendizaje; además del modo en que esta clase de agente representa las entradas necesarias para ejecutar el algormitmo mostrado en esta sección.

4.2. Agente aprendiz

Como se describe anteriormente en el capítulo 2.4.1, y en Bordini et al. (2007), desde el punto de vista de un intérprete *AgentSpeak(L)*, un agente está compuesto por un conjunto de creencias, un conjunto de planes, un conjunto de eventos, un conjunto de intenciones, funciones usadas en el ciclo de razonamiento y una instancia de la clase *Circumstance*, que incluye una representación de eventos pendientes, intenciones y otras estructuras necesarias durante la interpretación de un agente *AgentSpeak(L)*. Por defecto, la implementación de estas funciones está codificada en una clase llamada *Agent*, la cual, se personalizará para extender sus funciones básicas, de modo que agentes definidos como aprendices pueda ejecutar un mecanismo de aprendizaje después de la ejecución fallida de un plan.

La figura 4.3 presenta el diseño básico de una clase de agente que implementará aprendizaje intencional. La clase de agente *Learner* heredará los métodos de la clase *Agent*. Una vez que se haya implementado una clase de agente aprendiz intencional, se podrán definir nuevas clases de agentes que tengan la capacidad de aprender, y que además tengan un comportamiento adicional, tal es el caso del agente *SingleMindedLearner*, que como se mencionó en el capítulo introductorio, forma el caso de estudio de esta investigación. En este caso, el aprendizaje no será utilizado únicamente para redefinir el contexto de un plan, si no que también permitirá crear políticas de reconsideración, es decir, las reglas que el agente debe saber para abandonar una intención cuando considere racional hacerlo.

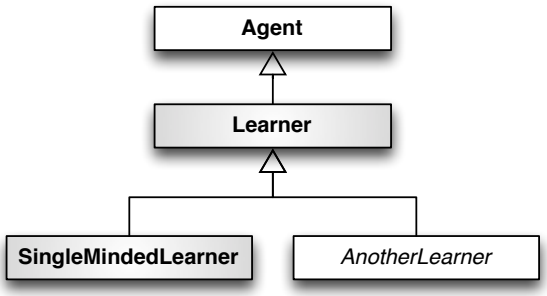


Figura 4.3 Diagrama de clase de un agente *Learner*

La clase **Learner**, permitirá definir agentes capaces de modificar el contexto de sus planes toda vez que haya ocurrido un fallo en la ejecución de un plan. De este modo, este tipo de agentes podrá aprender *a posteriori* nuevas razones para adoptar una intención futura cada vez que haya fallado en la ejecución de un plan. El comportamiento de esta clase de agente, será guiado por las siguientes reglas:

1. Si existe un plan aplicable, y el contexto del plan es consecuencia lógica del conjunto de creencias del agente, entonces ejecuta el plan seleccionado. Si la ejecución es satisfactoria, se agregará un ejemplo de entrenamiento etiquetado como *success*.
2. Si al momento de ejecutar el plan seleccionado ocurre un fallo, se agregará un ejemplo de entrenamiento etiquetado como *fail*, para después ejecutar un plan de aprendizaje; en caso de haber aprendido un contexto diferente, éste es modificado en el plan que produjo el fallo.

El mecanismo anterior se ejemplifica en la figura 4.4. Supóngase que dentro del ciclo de razonamiento del agente, éste está intentado colocar el bloque b sobre el bloque c. En el momento en que la ejecución del plan falla, se dispara el evento **-!put(b,c)** , el cual activa el plan **+!learning(put)**.

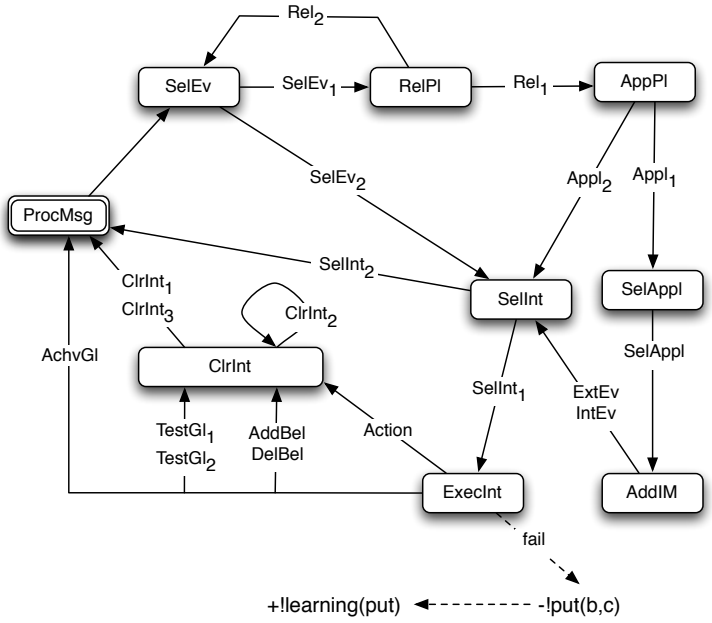


Figura 4.4 Modo operacional de un agente tipo *Learner*.

Obsérvese que en esta clase de agente no está siendo modificado el ciclo de razonamiento del agente, sino que el proceso de aprendizaje se llevará a cabo por un mecanismo de extensión de planes. Gracias a que Jason es un intérprete de *AgentSpeak(L)* programado en Java, los planes originales de los agentes pueden ser extendidos de modo que éstos puedan ejecutar acciones vinculadas con el aprendizaje, como recolectar ejemplos de entrenamiento, construir árboles lógicos de decisión y en el caso de esta investigación redefinir contextos o aprender reglas de abandono. La implementación del mecanismo de extension de planes es explicado con mayor detalle en la sección 5.2. Un evento disparador *(Trigger Event)* puede tener la forma $+!g$, $+?g$ y $+g$ dependiendo si se ejecuta una meta de logro, una meta de verificación o la adición de una creencia; o $-!g$, $-?g$ y $-g$ si se ejecutan planes de fallo o eliminación de creencias. En la librería JILDT solo se extenderán los planes de logro $+!g$ y su respectivo fallo $-!g$. No es necesario extender todos los planes de un agente, éste podrá definir que planes extenderá y que planes dejará sin modificar (ver figura 4.5).

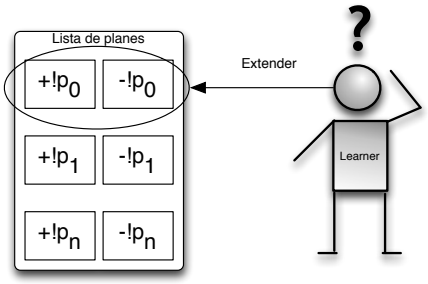


Figura 4.5 Un agente aprendiz podrá seleccionar que planes extenderá para aprender sobre ellos.

4.2.1. Modularidad de creencias

En la sección 4.1 se mencionó que se requieren tres entradas para ejecutar el algoritmo de inducción: el conocimiento general, la base de conocimiento y el sesgo del lenguaje. Uno de los objetivos de este trabajo es representar estas entradas como literales de primer orden para descartar el uso de archivos externos representándolas, que como se ha mencionado anteriormente, es la forma de hacerlo en la versión preliminar de JILDT.

Las reglas que forman el **conocimiento general** del agente se representan como tales dentro del estado mental del agente, sin embargo, los agentes aprendices deberán contar con una lista que apunte a estas reglas para facilitar la formación de interpretaciones, al momento de computar consecuencia lógica en la construcción de un árbol lógico de decisión. Los ejemplos de entrenamiento que forman la **base de conocimiento**, se representan con una literal del tipo *jildt_example/3*, donde el primer argumento es la etiqueta del plan que está siendo ejecutado al momento de recolectar un ejemplo etiquetado como satisfactorio, o la etiqueta del plan que arrojo el plan de fallo, en el caso de ejemplos etiquetados como fallidos; el segundo argumento es el conjunto de percepciones del agente cuando adoptó o falló una intención. El primer elemento de esta lista es una literal que asocia la intención actual del agente con la percepción del agente; por último, el tercer argumento define la clase a la que pertenece el ejemplo: *success* en el caso de ejemplos satisfactorios y *fail* en el caso de ejemplos fallidos. El cuadro 4.6 muestra la representación en primer orden de los modelos presentados en la figura 4.1.

```
1 jildt_example(put_label, [jildt_intend(put(b,c)), on(b,a), on(a,table), on(c,table), on(z,table)],success)
2 jildt_example(put_label, [jildt_intend(put(b,c)), on(b,a), on(a,table), on(z,c), on(c,table)],fail)
3 jildt_example(put_label, [jildt_intend(put(b,c)), on(b,a), on(a,table), on(z,b), on(c,table)],fail)
```

Cuadro 4.6 Representación en primer orden de los modelos presentados en la figura 4.1.

Para el caso del **sesgo del lenguaje** y las **configuraciones** se emplean dos literales: *jildt_rm/1* y *jildt_settings/2*. La literal *jildt_rm/1* define una directiva *rmode*, tal como se definieron en la sección 4.1. Los operadores descritos en el cuadro 4.2 trabajan de manera similar: el operador **+** define variables que deben ocurrir ya en la conjunción asociada, el operador **-** define variables nuevas, es decir, que no ocurren en la conjunción asociada. Debido a que Jason solo permite el uso de letras y números como términos (además de la variable anónima _), los simbolos **+** y **-** no pueden escribirse como tales, por lo que se definen como cadenas. Esto es, el equivalente del enunciado *rmode(on(+V1,+V2))* en JILDT es *jildt_rm(on("+","+"))*. Para el caso de constantes, solamente se considerará el valor declarado por el término constante. La figura 4.6 muestra los candidatos generados (derecha) a partir del sesgo del lenguaje (izquierda) y una consulta inicial $Q = intend(put(X,Y))$.

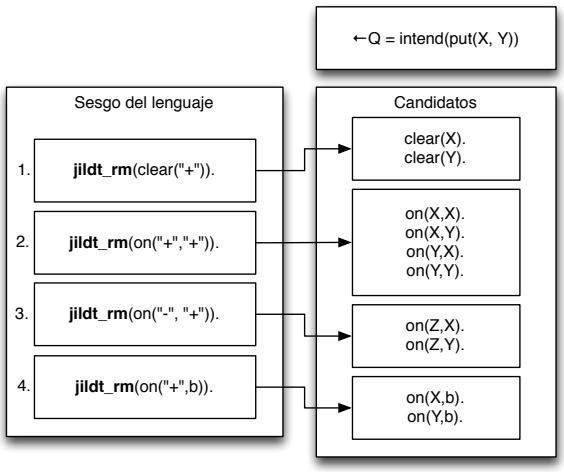


Figura 4.6 Derecha: Candidatos formados a partir del sesgo de lenguaje (izquierda) y una consulta inicial $Q = intend(put(X,Y))$.

El uso de las directivas *lookahead* es omitido en esta versión de JILDT. Estas construcciones eran utilizadas para vincular lógicamente las variables en el plan original con las variables de las literales que son candidatas a formar parte del árbol, sin embargo suponen un costo de computo elevado, tanto para generar las directivas como para formar los candidatos a partir de éstas. Como solución alterna, se redefine la consulta inicial al ejecutar la construcción de un árbol lógico de decisión, esto es, la consulta inicial $Q = [true]$ es reemplazada por $Q = [intend/1]$, donde la literal $intend/1$ asocia la intención actual del agente y esta formada por el evento disparador del plan que se esta ejecutando actualmente. La figura 4.6 presenta una consulta inicial de este tipo, para el plan $+!put(X,Y) : clear(X) < -move(X,Y)$.

La literal *jildt_settings/2* define las posibles configuraciones utilizadas en el proceso de aprendizaje. El cuadro 4.7 muestra las configuraciones que se pueden definir por el usuario para personalizar el proceso de aprendizaje.

Configuración	Valores	Descripción
trace	{true, false}	Indica si deben o no desplegarse mensajes de depuración en consola.
biasMode	{automatic, manual}	Indica el modo en que se computa el sesgo del lenguaje. El término *automatic* obliga al agente a computar todas las combinaciones posibles de directivas *jildt_rm*. Utilizando el término *manual* se emplean únicamente aquellas directivas definidas por el usuario.
learningPlansSrc	$< PATH >$	Indica el origen de los planes de aprendizaje. Por defecto, los planes de aprendizaje están definidos en el archivo learningPlans.asl.
excludeBels	$< [Functors] >$	Indica qué creencias debe descartar el agente para formar los ejemplos de entrenamiento.
inductionLevel	{java, agentSpeak}	Ejecuta el algoritmo de inducción desde una acción interna implementada en *Java* o a través de un conjunto de planes implementados en AgentSpeak(L).

Cuadro 4.7 Configuraciones posibles en el proceso de aprendizaje.

Un agente *Learner* extiende la representación del estado mental de un agente por defecto de Jason. Además de la base de creencias habitual con la que cuenta un agente por defecto, un agente *Learner* contará con una base de creencias de aprendizaje *(LearningBB)* para creencias relacionadas con el proceso de aprendizaje, esto es, las creencias que inician con el prefijo *jildt*. Por otra parte, cuenta con una lista de reglas que definen el conocimiento general *(background)* del agente. Este nuevo estado mental (Ver la figura 4.7) proporciona una verdadera representación modular e intencional de los ejemplos de entrenamiento, el sesgo de lenguaje y el conocimiento general, además de que supone una mejora a la eficiencia de la librería en comparación con versiones anteriores de JILDT basados en archivos externos.

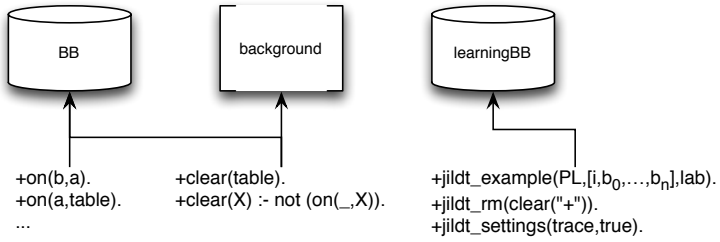


Figura 4.7 Distribución de las creencias de un agente *Learner*

La manipulación de los ejemplos de entrenamiento puede llegar a ser complicada dado que las literales *jildt_example/3* pueden llegar a ser bastante extensas. Es por eso, que se ha diseñado una base de creencias personalizada para almacenar estas literales, la cual llamamos *LearningBeliefBase*. A diferencia de una base de

creencias convencional, las literales en *LearningBeliefBase* tienen asignado un identificador numérico. De esta manera, la lista de ejemplos de entrenamiento mostrados en el cuadro 4.6 puede representarse como *[1,2,3]*. Además de esta característica, la base de creencias *LearningBelifBase* añade una anotación de repetición para cada literal de ejemplo. Esta caracteristica nos sirve como criterio de desempate al momento de seleccionar candidatos al construir un árbol lógico de decisión. La figura 4.8 muestra el diseño general de la base de creencias de aprendizaje.

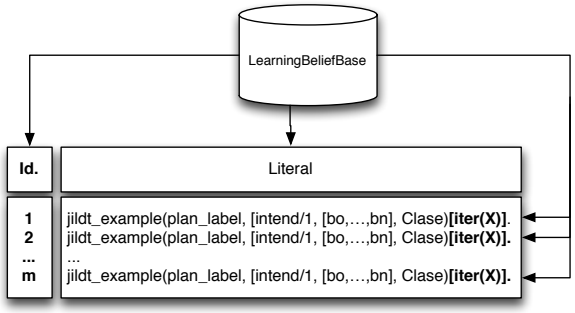


Figura 4.8 Diseño de la base de creencias de aprendizaje *LearningBeliefBase*.

La salida del algoritmo de aprendizaje también se ha representado en un formato que un agente *Learner* pueda entender, en este caso usando listas de literales. El primer elemento es una literal que representa la etiqueta del nodo. El segundo y tercer elementos son listas que representan los árboles izquierdo y derecho, respectivamente. En caso de ser un nodo hoja, se reemplaza la lista por el término *success* o *fail*, dependiendo si se encuentra al lado izquierdo o derecho del árbol. La figura 4.9 muestra la representación como lista del árbol que se muestra en la figura 4.1(b).

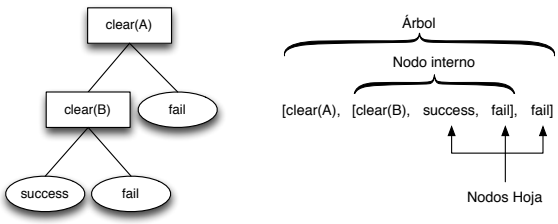


Figura 4.9 Representación de un árbol lógico de decisión como una lista de literales.

4.3. Resumen

La inducción de árboles lógicos de decisión resulta ser un mecanismo idóneo para sustentar aprendizaje en los sistemas multiagentes BDI, esto gracias a que la expresividad en primer orden de los enunciados que conforman las creencias del agente permite formar ejemplos de entrenamiento adecuados al algoritmo TILDE. Otra ventaja que se ha mencionado, es que cada recorrido del nodo raíz a una hoja corresponde a una conjunción de literales de primer orden, lo que es equivalente al tipo de representación empleada para definir el contexto de un plan.

Las entradas necesarias para ejecutar el algoritmo de aprendizaje toman una nueva representación en este trabajo. En versiones preeliminares, se hace uso de ficheros externos para representar el conocimiento general, los ejemplos de entrenamiento, las configuraciones y sesgo del lenguaje. En este capítulo se presenta una representación alternativa de estas entradas, las cuales son definidas en literales de primer orden y modifican el estado mental de los agentes, lo cual permite dar soporte a las representaciones previamente mencionadas. A diferencia de un agente convencional, un agente aprendiz *Learner* cuenta con dos bases de creencias, una de ellas personalizada para almacenar las creencias que están vinculadas directamente con el aprendizaje.

El siguiente capítulo presenta de manera mas detallada la implementación tanto del algoritmo de aprendizaje, como de la clase de agente *Learner*, que como se mencionó en este capítulo, extiende sus capacidades originales para ejecutar aprendizaje, a través de un mecanismo de extension de planes.

Capítulo 5
Implementación

Como se ha mencionado anteriormente, Jason es intérprete de *AgentSpeak(L)* basado en Java. Gracias a su naturaleza orientada a objetos es posible aumentar las capacidades y funcionalidades de un agente por defecto para sustentar aprendizaje intencional en sistemas multiagentes BDI. En su mayoría la librería JILDT está formada por clases que extienden algunas clases definidas en Jason.

En este capítulo se describe a detalle la implementación de JILDT. La primera sección describe las acciones internas y funciones matemáticas que se implementaron para ejecutar el proceso de aprendizaje, que van desde la recolección de ejemplos de entrenamiento, hasta la construcción de árboles lógicos de decisión. Las acciones internas se encuentran divididas en dos paquetes: el paquete *jildt* y el paquete *jildt.tilde* mientras que las funciones matemáticas se encuentran en el paquete *jildt.tilde.math*. La segunda sección presenta el mecanismo utilizado para extender los planes originales de un agente a través de la directiva de pre-procesamiento *LearnablePlans*. En la misma sección, se presentan los planes de aprendizaje y los planes que implementan el algoritmo TILDE visto en la sección 4.1. El algoritmo de inducción de árboles lógicos de decisión se implementó en dos niveles de programación: Primero, un nivel Java diseñado para mejorar el desempeño computacional en agentes definidos para aprender sin alguna interación social; el segundo define un nivel *AgentSpeak(L)* más flexible, el cual abre las puertas a la definición de sistemas multiagentes que aprendan colectivamente. La tercera sección describe la arquitectura de la clase de agente aprendiz *(Learner)*, y cómo puede ser extendida para formar una clase de agente aprendiz más específica, un agente que cuenta con una estrategia de compromiso racional *(singleMinded-Learner)*, que como se mencionó en el capítulo introductorio, es caso de estudio en este proyecto. Por último, la cuarta sección presenta otras clases y funcionalidades implementadas en la librería JILDT.

La librería JILDT se encuentra disponible en la web[1], en la que se puede descargar la última version de JILDT y consultar la documentación de la interfaz de programación de aplicaciones *(API Documentation)*.

[1] Disponible en http://jildt.sourceforge.net/

5.1. Acciones internas y funciones matemáticas

A pesar de contar con un amplio conjunto de acciones internas, *Jason* no provee funciones que den soporte al aprendizaje intencional en los agentes. Por tal motivo, la librería JILDT extiende las capacidades de los agentes a través de acciones internas personalizadas para dar soporte al aprendizaje intencional. Dichas acciones internas están desarrolladas en Java, y heredan sus métodos de la clase *DefaultInternalAction*, que a su vez implementa la interfaz *InternalAction*. Estas acciones internas sobrescriben el método *execute*, el cual es llamado por el intérprete del agente para ejecutar la acción interna. El primer argumento de este método es el **sistema de transición**, el cual contiene toda la información sobre el estado actual del agente; el segundo argumento es el **unificador** actualmente determinado por la ejecución del plan donde la acción interna apareció, o la comprobación de si el plan es aplicable, esto dependiendo de si la acción interna que está siendo ejecutada apareció en el cuerpo o el contexto de un plan; el tercer argumento es un arreglo de **términos** y contiene los argumentos enviados a las acciones internas por el usuario en el código *AgentSpeak(L)* que ejecuta la acción interna. Las acciones internas regresan valores booleanos, *true* en caso de que la ejecución de la acción interna sea satisfactoria, y *false* en caso de que falle. Si la función tuviese que regresar un valor, lo hace mediante la unificación con los términos que recibe en su lista de argumentos.

Las acciones internas implementadas en la librería JILDT se encuentran en dos paquetes: el paquete *jildt* y el paquete *jildt.tilde*. Dado que las acciones internas forman parte de paquetes en Java, a cada una de éstas se debe anteponer el prefijo *jildt* o *jildt.tilde*, según sea el caso.

5.1.1. Paquete jildt

Las acciones internas que conforman el paquete *jildt* permiten ejecutar funciones generales relacionadas con el aprendizaje, como obtener información necesaria para el ejecutar el algoritmo de aprendizaje, o modificar el comportamiento del agente a partir del conocimiento adquirido después de realizar el proceso de aprendizaje. La figura 5.3 presenta el diagrama de clases[2] de las acciones internas implementadas en *jildt*. Después de esto, se describe la funcionalidad de cada una de estas acciones.

[2] En adelante, se emplea una notación UML estándar, donde los atributos y métodos públicos se denotan con el símbolo $+$, los privados se denotan por $-$ y los protegidos con #. Los métodos subrayados refieren métodos estáticos. Las relaciones de herencia de clase e implementación de interfaz se denotan con una flecha con línea continua y línea punteada respectivamente, dirigidas hacia la clase de la cual se hereda o la interfaz que se implementará. Las clases sombreadas representan aquellas que están incluidas dentro de la librería JILDT.

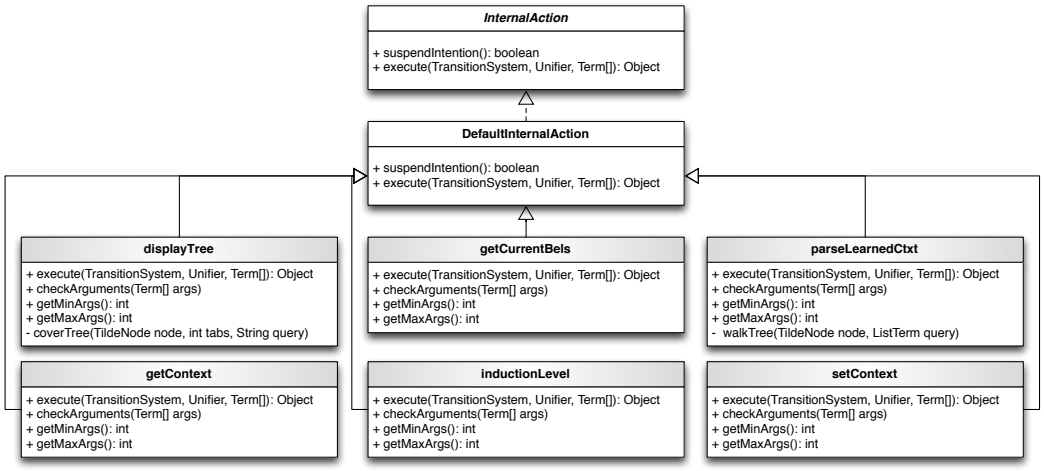


Figura 5.1 Diagrama de clase de las acciones internas en JILDT.

- *jildt.displayTree(Tree, Mode)*. Despliega en pantalla una visualización del árbol declarado en la variable *Tree*. El árbol puede mostrarse en consola o en una ventana externa la variable *Mode* unifica con *console* o *gui*, respectivamente. La opción *both* despliega el árbol de las dos formas. La figura 5.2 ejemplifica estas configuraciones.

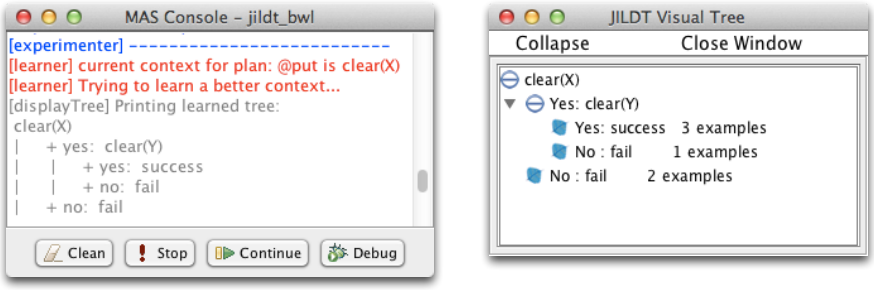


Figura 5.2 Visualización de un árbol lógico de decisión en consola (izquierda) y en una interfaz gráfica de usuario (derecha).

- *jildt.getContext(Plan, Ctxt)*. Obtiene el contexto del plan en el cual su etiqueta unifica con el valor de la variable *Plan*. El resultado es devuelto como una instancia de la clase *LogicalFormula* en la variable *Ctxt*. Cuando un plan tiene un contexto vacío, el resultado es el término *true*.
- *jildt.setContext(Plan, Ctxt)*. De manera inversa a la acción interna *jildt.getContext*, esta acción interna permite modificar el contexto del plan cuya etiqueta unifique con la variable *Plan*, por un nuevo contexto *Ctxt*.

- *jildt.getCurrentBels(Bs)*. Esta acción interna permite recuperar la lista con las creencias actuales del agente, las cuales son unificadas con la variable *Bs*. Únicamente recupera las creencias del agente relacionadas con su problemática original (B), y excluye las reglas (que forman parte del conocimiento general, *CG*), las creencias que están relacionadas con el aprendizaje (aquellas que inician con el prefijo *jildt*), y las creencias que se definen como excluibles en la literal *jildt_settings(excludeBels,[*$f_0, ..., f_n$*])*.

$$Bs = [b_0, b_1, ...b_n] \, donde \begin{cases} b_i \in B \quad \& \quad b_i \notin CG \\ b_i \notin \{jildt\_example, jildt\_intend, jildt\_rm, jildt\_settings\} \\ b_i \notin [f_0, ..., f_n] \end{cases}$$

- *jildt.inductionLevel(Lvl)*. Permite verificar el mecanismo de inducción configurado. El aprendizaje puede ejecutarse en un nivel de programación Java o *AgentSpeak(L)*. A nivel Java, el proceso de aprendizaje se ejecuta a través de una acción interna, la cual define los métodos necesarios para construir un árbol lógico de decisión. El nivel *AgentSpeak(L)* ejecuta el proceso de aprendizaje a través de un conjunto de planes que ejecutan diversas acciones internas definidas en el paquete *jildt.tilde*. El mecanismo de inducción se puede configurar usando la literal *jildt_settings(inductionLevel, Level)*, donde la variable *Level* unifica con el término *java* o *agentSpeak*, según sea el caso. Por defecto, un agente aprendiz ejecuta el aprendizaje en un nivel Java.
- *jildt.parseLearnedCtxt(Tree, LC)*. Obtiene el contexto aprendido de un árbol definido en la variable *Tree*. El resultado es la lista con todas los caminos que llevan desde el nodo raíz hasta un nodo hoja etiquetado como *success*. Cada camino es representado como una lista de literales.

5.1.2. Paquete jildt.tilde

Las acciones internas que conforman el paquete *jildt.tilde* permiten ejecutar funciones directamente relacionadas con la construcción de árboles lógicos de decisión. La figura 5.3 presenta el diagrama de clases de las acciones internas implementadas en el paquete *jildt.tilde*, seguido de la descripción de sus funcionalidades.

- *execTilde(Plan, Tree)*. Ejecuta el algoritmo TILDE para construir un árbol lógico de decisión, implementando los algoritmos 4 y 5 descritos en la sección 4.1. La variable *Plan* indica la etiqueta del plan para el cual se ejecutará el algoritmo de aprendizaje. La variable *Tree* unifica con el árbol aprendido.
- *findExamples(Plan, Exs)*. Busca y recolecta los ejemplos de entrenamiento que corresponden a la etiqueta de plan definida en la variable *Plan*. Los ejemplos de entrenamiento son almacenados en una base de creencias personalizada que indiza las literales contenidas en ésta. Como resultado, *Exs* unifica con una lista de índices, que son utilizados para recuperar los ejemplos de entrenamiento como literales.

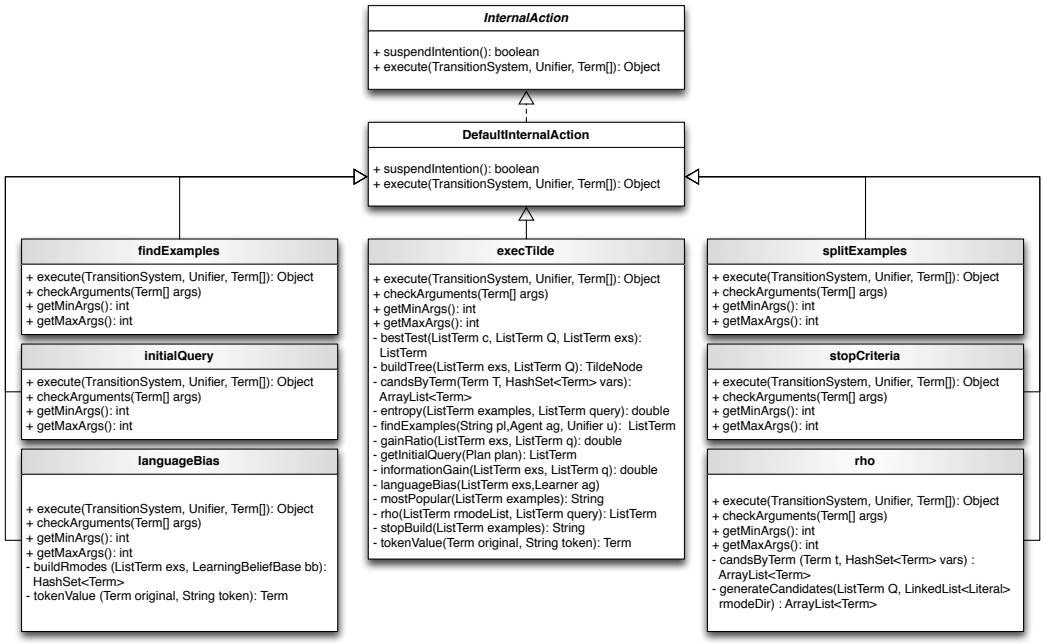


Figura 5.3 Diagrama de clase de las acciones internas en JILDT.TILDE.

En esta primera versión se utilizan los índices para facilitar el manejo de los ejemplos de entrenamiento entre los planes que hacen uso de éstos, ya que enviar listas de literales *jildt_example/3* puede llegar a ser muy complejo, debido a la longitud de este tipo de literales. Sin embargo, como trabajo futuro se pretende dejar la opción abierta a utilizar la representación que mejor se adecue al problema (índices o literales).

- *initialQuery(Plan, Qi)*. Construye la consulta inicial Q_i para construir un árbol lógico de decisión. La consulta inicial es una lista de un elemento, donde el único elemento es una literal *jildt_intend/1*, cuyo único argumento está formado por el evento disparador de un plan etiquetado como *Plan*. Esta estructura vincula lógicamente las variables del plan con las del árbol por construir.
- *languageBias(Exs, Lb)*. Permite computar el sesgo del lenguaje a través de los ejemplos definidos en la variable *Exs*. Como se mencionó en la sección 4.1, el sesgo del lenguaje es la definición de qué literales van a ser consideradas como candidatos para ser incluidos en el árbol lógico de decisión.

  Un agente aprendiz cuenta con dos opciones para manipular el sesgo del lenguaje: automática y manual. En el primer caso, se computan todas las posibles combinaciones de directivas *jildt_rm* que se puedan formar, mientras que en el segundo caso, las directivas no son computadas y se usan únicamente aquellas definidas el usuario. Para especificar el modo en que se manipulará el sesgo del lenguaje, se utiliza la literal *jildt_settings(biasMode, Mode)*. La variable *Mode* unifica con *automatic* o *manual*, según sea el caso.

Como resultado, *Lb* unifica con una lista de literales *jildt_rm/1*, sin modificar el estado mental del agente, esto es, las literales obtenidas no son añadidas como creencias, por lo que se sugiere se realice esta acción al obtener el resultado.

- *rho(Q, Cns)*. Genera la refinación de candidatos para la consulta inicial *Q*. Los candidatos son generados a partir del sesgo del lenguaje que el agente cree, por ello es necesario añadir las literales *jildt_rm* computadas por la acción interna *jildt.tilde.languageBias/2*, o en su defecto definirlas manualmente. El procedimiento para generar los candidatos, es el mismo que se presenta en el ejemplo 10, en la sección 4.1.
- *splitExamples(Exs, Qb, Sp)*. Divide la lista de ejemplos de entrenamiento definida en *Exs*, entre aquellos que satisfacen la consulta en *Qb*, y aquellos que no. El resultado es una lista de dos elementos, donde el primer elemento es una lista con los ejemplos que satisfacen la consulta *Qb* y el segundo elemento es una lista con los elementos que no la satisfacen. Un uso alternativo de esta acción interna es *jildt.tilde.splitExamples(Exs, Qb, [Esu, Ense])*, donde las variables *Esu* y *Ensu* unifican con los ejemplos que satisfacen la consulta y los que no, respectivamente.
- *stopCriteria(Exs, percentage(P), Class)*. Comprueba si el conjunto de ejemplos *Exs* cumple un criterio para detener la construcción del árbol. El mecanismo elegido es verificar que un porcentaje de los ejemplos pertenece a una clase. La literal *percentage(P)* define el porcentaje de ejemplos necesarios para satisfacer el criterio de parada. La variable *Class* unifica con la etiqueta de clase a la que pertenece la mayoría de los ejemplos.

5.1.3. Paquete jildt.tilde.math

Además de las acciones internas, la librería JILDT contiene funciones matemáticas, las cuales implementan las ecuaciones presentadas en la sección 4.1.1 y se encuentran en el paquete *jildt.tilde.math*. Estas funciones matemáticas heredan los métodos de la clase *DefaultArithFunction*, la cual a su vez implementa la interfaz *ArithFunction*.

De manera similar a como las acciones internas sobrescriben el método *execute*, las funciones matemáticas sobrescriben el método *evaluate*, el cual es llamado por el intérprete del agente para ejecutar la acción interna. El primer argumento de este método es el **sistema de transición**, que contiene la información sobre el estado actual del agente; el segundo argumento es el arreglo de **términos** que contiene los argumentos enviados a la función matemática desde el código *AgentSpeak(L)* que la ejecuta.

A diferencia de las acciones internas, las funciones matemáticas regresan valores reales *(Doubles)*. Esto último presenta una ventaja en este proyecto, ya que permite ejecutar una función matemática desde el código en Java de otra. Por ejemplo, la función de ganancia de información (Eq. 4.2) requiere ejecutar mas de una vez

la función de entropía (Eq. 4.1). Utilizando acciones internas, el cálculo de entropía tendría que computarse antes de ejecutar la acción de ganancia de información. Para aprovechar aún más esta ventaja, las funciones matemáticas implementadas en la librería JILDT implementan un método estático llamado *calculate*, el cual ejecuta el método *evaluate* de la función, y facilita su ejecución desde cualquier otra clase implementada en Java.

En código *AgentSpeak(L)*, el resultado de una función matemática debe asignarse a una variable, por ejemplo, $GR = jildt.math.gainRatio(Exs, Qb)$. Para poder utilizar las funciones matemáticas, éstas deben definirse al principio del código del agente con el enunciado *register_function("pack.function")*, sin embargo, un agente aprendiz implementa una método que se encarga de definir las funciones del paquete *jildt.tilde.math*, por lo que el usuario se deslinda de definir las funciones del paquete *jildt.tilde.math*. La figura 5.4 muestra el diagrama de clases de las funciones matemáticas implementadas.

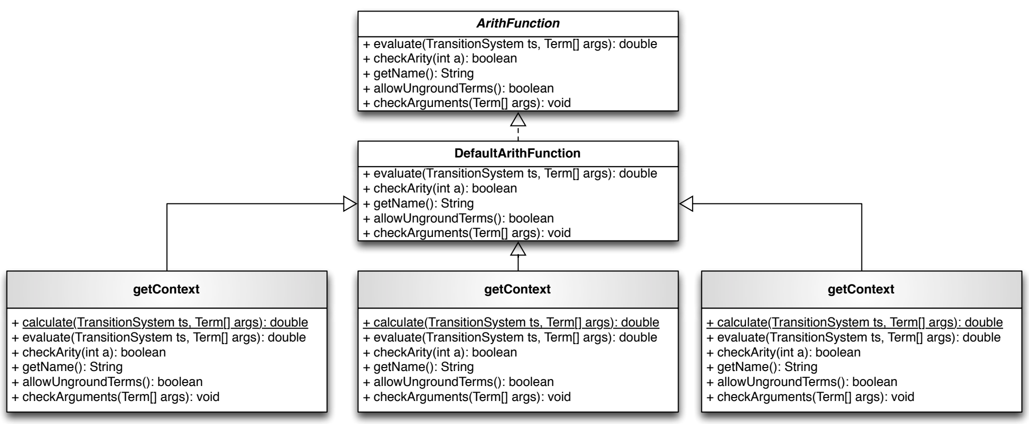


Figura 5.4 Diagrama de clase de las funciones matemáticas en *jildt.tilde.math*.

5.2. Extensión de planes

Como se mencionó en la sección 4.2, un agente aprendiz ejecuta el proceso de aprendizaje a través de un mecanismo de extensión de planes. Gracias a que *Jason* es un intérprete de *AgentSpeak(L)* programado en Java, los planes originales de los agentes pueden ser extendidos de modo que puedan ejecutar acciones relacionadas con el proceso de aprendizaje. La extensión de planes se lleva a cabo desde una directiva de pre-procesamiento llamada *LearnablePlans*, que se encuentra en del paquete *jildt* e implementa la interfaz *Directive* (Ver figura 5.5).

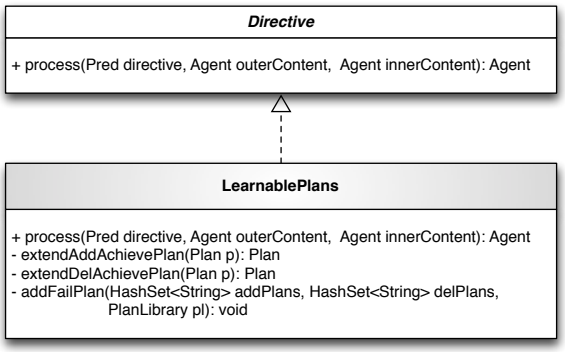


Figura 5.5 Diagrama de clase de la directiva de pre-procesamiento *jildt.LearnablePlans*.

Las directivas de pre-procesamiento se utilizan para pasar algunas instrucciones al intérprete que no están relacionados con la semántica del lenguaje, sino que son meramente sintácticas. La directiva *LearnablePlans* es usada para modificar únicamente los planes definidos para lograr una meta *(Achive goals (!))* que se encuentran definidos dentro del alcance de la directiva de pre-procesamiento *LearnablePlans*. El cuadro 5.1 muestra la sintáxis de esta directiva de pre-procesamiento[3].

```
{begin learnablePlans}
   +!p0;
   -!p1;
   ...
   +!pn;
{end}
```

Cuadro 5.1 Sintaxis del uso de la directiva de pre-procesamiento *jildt.LearnablePlans*

Para ejemplificar en qué consiste la extensión de planes, se introduce un programa de agente para el mundo de los bloques adaptado de la implementación de Bordini et al. (2007), el cual se muestra en el cuadro 5.2.

[3] El nombre de la clase es *LearnablePlans*, sin embargo, para utilizar esta directiva dentro del código *AgentSpeak(L)* es necesario definirla en el archivo de configuración del SMA *(*.mas2j)* con la sentencia **directives: learnablePlans = jildt.LearnablePlans**, por ello, en adelante se refiere al uso de esta directiva como *learnablePlans*.

```
1  /* Initial beliefs and rules */
2  clear(X) :- not(on(_,X)).
3  clear(table).
4
5  /* Plans */
6  {begin learnablePlans}
7  @put_succCase
8  +!put(X,Y) : true <-
9    move(X,Y);
10   .print("Yeah, I did the task succesfully");
11   .send(experimenter,tell,experiment(done)).
12
13 @put_failCase
14 -!put(X,Y) : true <-
15   .send(experimenter,tell,experiment(done)).
16 {end}
17
18 @put_failCaseNonApplicable
19 -!put(X,Y)[error(Error)] : .member(Error,[no_applicable,no_relevant,no_option,wrong_arguments,unknown]) <-
20   .print("Plan +!put produced an irrelevant failure.");
21   .send(experimenter,tell,[non_applicable(put),experiment(done)]).
```

Cuadro 5.2 Programa de agente para el mundo de los bloques, basado y simplificado de su implementación en Bordini et al. (2007).

En este ejemplo, solo los planes etiquetados como `@put_succCase`(líneas 7-11) y `@put_failCase`(líneas 13-15) pueden ser extendidos, ya que se encuentran dentro del alcance de la directiva `learnablePlans`(líneas 6-16), mientras que el plan etiquetado como `@put_failCaseNonApplicable`(líneas 18-21) no se extiende por estar fuera del alcance de la directiva de pre-procesamiento.

Dos tipos de extensiones han sido implementadas, una para los planes que se ejecutan al agregar una meta de logro *(+!)* y otra para los planes que se ejecutan cuando fallan estos planes *(-!)*. En el primer caso, el plan original es extendido (ver el cuadro 5.3) en tres partes: la primera parte, o preprocesamiento (líneas 3-6), consiste en obtener información relevante sobre el mundo actual del agente, antes de ejecutar el plan original. Primero se obtiene el deseo actual del agente, usando la acción interna `.desire(Des)`, después se obtienen las creencias del agente al momento de iniciar la ejecución del plan, usando la acción interna `jildt.getCurrentBeliefs(Bs)`. El deseo y las creencias actuales del agente formarán parte del ejemplo de entrenamiento y son concatenadas en una lista. Una vez que se obtiene el estado BDI del agente se agrega la creencia `jildt_tagPlan(put_succCase)` para indicar cual es el plan que se está intentando ejecutar. La segunda parte introduce el cuerpo original del plan (líneas 7-9). Por último, la tercera parte (líneas 10-11), que se ejecuta únicamente si el plan original concluye con éxito, se elimina la creencia `jildt_tagPlan(put_succCase)`, y se agrega un ejemplo de entrenamiento etiquetado como *success*, indicando que una vez que se intentó ejecutar el plan *put_succCase*, teniendo el deseo *Des* y conociendo las creencias *Bs*, la ejecución del plan fue satisfactoria (*success*).

```
1  @put_succCase
2  +!put(X,Y) : true <-
3    .desire(Des);
4    jildt.getCurrentBels(Bs);
5    .concat([intend(Des)], Bs, Train);
6    +jildt_tagPlan(put_succCase);
7    move(X,Y);
8    .print("Yeah, I did the task succesfully");
9    .send(experimenter,tell,experiment(done)).
10   -jildt_tagPlan(put_succCase);
11   +jildt_example(put_succCase, Train, success).
```

Cuadro 5.3 Extensión del plan *put_succCase* del cuadro 5.2.

En el caso de los planes que se ejecutan al fallar un plan de logro *(-!)*, la extensión se realiza de la siguiente manera (ver el cuadro 5.4): al contexto del plan original se añade una consulta para ejecutar este plan únicamente cuando el error se produjo por un fallo en una acción interna `(ia_failed)`, en una acción del plan `(action_failed)`, en una consulta `(ask_failed)` o en una restricción `(constraint_failed)`. En el cuerpo del plan, se obtienen el deseo actual del agente y las creencias del agente al momento de iniciar la ejecución del plan, para formar la lista de creencias que formarán parte del ejemplo de entrenamiento (líneas 3-5). Después de esto, se consulta qué plan estaba siendo ejecutado cuando se produjo el error(línea 7) y se ejecuta un plan de aprendizaje (línea 8), del cual aprenderá si es posible, nuevas razones para adoptar el plan etiquetado como *Tag*. Por último, al finalizar el proceso de aprendizaje, elimina la creencia `jildt_tagPlan(Tag)` (línea 9) y ejecuta las acciones que conformaban el plan de fallo original (línea 10).

```
1  @put_failCase
2  -!put(X,Y)[error(Error)] : .member(Error,[ia_failed, action_failed, ask_failed, constraint_failed]) <-
3    .desire(Des);
4    jildt.getCurrentBels(Bs);
5    .concat([intend(Des)], Bs, Train);
6    ?jildt_tagPlan(Tag);
7    +jildt_example(Tag, Train, fail);
8    !learning(Tag, Tree);
9    -jildt_tagPlan(Tag);
10   .send(experimenter,tell,experiment(done)).
```

Cuadro 5.4 Extensión del plan *put_failCase* del cuadro 5.2.

5.2.1. Planes de aprendizaje

Como se muestra en el cuadro 5.4, un plan que responde a un evento de fallo *(-!)* intenta ejecutar el plan de aprendizaje `!learning(Tag, Tree)` para aprender un nuevo contexto para el plan que estaba siendo ejecutado al momento de producirse el fallo, definido en la variable *Tag*. A continuación se explica el funcionamiento de los planes implementados para ejecutar el proceso de aprendizaje. Los planes de aprendizaje se encuentran definidos en el archivo de agente *learningPlans.asl*, en el paquete *jildt*. Sin embargo, puede definirse el origen de estos planes usando la literal *jildt_settings(learningPlansSrc, <PATH>)* en el programa de agente.

El cuadro 5.5 muestra el plan de aprendizaje ejecutado desde un plan de fallo extendido. Primero se ejecuta el plan `executeTilde(P, Tree)` (línea 4) para construir el árbol lógico de decisión relacionado con el fallo producido por el plan etiquetado como *P*. El árbol computado es unificado con la variable *Tree*. Una vez computado el árbol lógico de decisión, se visualiza usando la acción interna `jildt.displayTree(Tree, both)`. Por último se obtiene un nuevo contexto a partir de las ramas del árbol computado que llevan a un nodo hoja etiquetado como *success*, y se sustituye el contexto del plan que provocó el fallo con el contexto aprendido.

```
1 @learning
2 +!learning(P,Tree): true <-
3   .print("Trying to learn a better context...");
4   !executeTilde(P,Tree);
5   jildt.displayTree(Tree, both);
6   jildt.parseLearnedCtxt(Tree, LC);
7   .print("Learned context for ",P," is ", LC);
8   jildt.setContext(P, LC).
```

Cuadro 5.5 Plan de aprendizaje.

El plan `executeTilde(P,Tree)` (línea 4) construye un árbol lógico de decisión y puede ejecutarse en dos niveles distintos de programación: Como una acción interna programada en Java y como un conjunto de planes programados en *Jason/AgentSpeak(L)*. Para configurar el nivel de inducción, se declara una literal *jildt_settings(inductionLevel, Lvl)*, donde la variable *Lvl* unifica con *java* o *agentSpeak*, según sea el caso. Por defecto, y por motivos de eficiencia, un agente aprendiz ejecuta el algoritmo de aprendizaje en un nivel Java.

El cuadro 5.6 muestra los planes definidos para la construcción de un árbol lógico de decisión. El plan etiquetado como `@execTilde_agSpeak`(líneas 1-7) ejecuta un plan para inducir la construcción del árbol si el si el nivel de inducción es *agentSpeak*, en caso contrario, se ejecuta el plan etiquetado como `@execTilde_java`(líneas 9-11), el cual construye el árbol desde la acción interna `jildt.tilde.execTilde`.

```
1  @execTilde_agSpeak
2  +!executeTilde(P, Tree) : jildt.inductionLevel(agentSpeak) <-
3    jildt.tilde.findExamples(P, Exs);
4    jildt.tilde.languageBias(Exs, Lb);
5    for (.member(Rm,Lb)){+Rm;};
6    jildt.tilde.initialQuery(P, Qi);
7    !buildTree(Exs, Qi, Tree).
8
9  @execTilde_java
10 +!executeTilde(P, Tree) : jildt.inductionLevel(java) <-
11   jildt.tilde.execTilde(P,Tree).
```

Cuadro 5.6 Planes de inducción de TILDE

5.2.2. Construcción de Árboles Lógicos de Decisión enAgentSpeak(L)

A continuación se explica cómo se construyen árboles lógicos de decisión desde un nivel de programación *AgentSpeak(L)*, partiendo del plan etiquetado como *@execTilde_agSpeak* en el cuadro 5.6. Primero, se buscan los ejemplos de entrenamiento relacionados con el plan *P*(línea 3). Una vez que se obtienen los ejemplos de entrenamiento, se genera y agrega el sesgo del lenguaje (líneas 4-5). El siguiente paso es formar la consulta inicial *Qi* (línea 6), para empezar a construir el árbol lógico de decisión con las entradas generadas anteriormente.

La construcción del árbol se realiza recursivamente usando los planes mostrados en el cuadro 5.7. El plan `@buildTree_stop` detiene la construcción del árbol cuando un criterio de paro se cumple. En este caso, la variable *Tree* unifica con una lista que representa un nodo hoja, con su respectiva etiqueta de clase y el total de ejemplos que pertenecen a ésta (línea 4). En caso de no cumplirse el criterio de paro, se ejecuta el plan `@buildTree_recursive`. Primero genera los candidatos a formar parte del árbol (línea 8) y selecciona el mejor de ellos a través del plan `bestTest(Exs, Q, Cns, Bcn)` (Cuadro 5.8). Una vez que se obtiene el mejor candidato, se construye una nueva consulta *Qb* agregando el mejor candidato a la consulta inicial (línea 10) y se dividen los ejemplos de entrenamiento entre aquellos que satisfacen *Qb* y aquellos que no (línea 11). Dos árboles se construyen y formaran los nodos izquierdo y derecho del árbol (líneas 12-13). El nodo *izquierdo* se forma a partir de los ejemplos que satisfacen la consulta *Qb* y se envía como consulta inicial a *Qb*; el nodo *derecho* se forma a partir de los ejemplos que no satisfacen *Qb* y recibe como consulta inicial a *Q*. La variable *Tree* unifica con una lista formada por la literal del nodo y sus nodos izquierdo y derecho.

El plan *bestTest* computa el coeficiente *Gain Ratio* para todos los candidatos recibidos en la variable *Cns*. La literal *jildt_bestTest* permite almacenar temporalmente la información sobre el mejor candidato durante la búsqueda. La variable *Bcn* unifica con el candidato que maximice su valor de *Gain Ratio*.

```
@buildTree_stop
+!buildTree(Exs, _,Tree): jildt.tilde.stopCriteria(Exs, percentage(100), Class) <-
  .length(Exs, L);
  Tree = [Class, L].

@buildTree_recursive
+!buildTree(Exs, Q,Tree): true <-
  jildt.tilde.rho(Q, Cns);
  !bestTest(Exs, Q, Cns, Bcn);
  .concat(Q, [Bcn], Qb);
  jildt.tilde.splitExamples(Exs, Qb, [ExsLeft, ExsRight]);
  !buildTree(ExsLeft, Qb, Left);
  !buildTree(ExsRight, Q, Right);
  Tree = [Bcn, Left, Right].
```

Cuadro 5.7 Planes para construir un árbol lógico de decisión.

```
@bestTest
+!bestTest(Exs, Q, Cns, Bcn): true <-
  +jildt_bestTest(true, 0);
  for (.member(Cn,Cns)){
    .concat(Q, [Cn], Qb);
    GR = jildt.math.gainRatio(Exs,Qb);
    ?jildt_bestTest(Temp, Max);
    if (GR > Max) {
      -jildt_bestTest(Temp, Max);
      +jildt_bestTest(Cn, GR);
    };
  };
  ?jildt_bestTest(Bcn, Max);
  -jildt_bestTest(Bcn, Max).
```

Cuadro 5.8 Plan de selección del mejor candidato.

5.3. Clase de agente *Learner*

Bordini et al. (2007) describen desde el punto de vista de un interprete *AgentSpeak(L)* extendido a un **agente** como un conjunto de creencias (*Bs*), un conjunto de planes (*Ps*), algunas funciones de selección definidas por el usuario y la función de confianza (una relación "socialmente aceptable" para los mensajes recibidos), funciones usadas en el ciclo de razonamiento (por ejemplo, la función de actualización de creencias (*BUF)* y la función de revisión de creencias (*BRF*)), y una instancia de la clase *Circumstance*, la cual incluye eventos pendientes, intenciones y otras estructuras necesarias durante la interpretación de un agente *AgentSpeak(L)*. La implementación por defecto de estas funciones está codificada en una clase llamada *Agent*, la cual ha sido personalizada extendiendo sus funciones básicas para aprender nuevas razones para adoptar planes toda vez que haya fallado en la ejecución de éstos y se encuentra en el paquete *jildt.agent*. La figura 5.6 muestra el diagrama de clases de un agente aprendiz *Learner*. Por razones de presentación, solo se muestran los atributos que son propios de la

clase de agente *Learner* y los métodos que fueron sobrescritos, sobrecargados o que son propios de esta clase de agente (a excepción de los *getters* y *setters*).

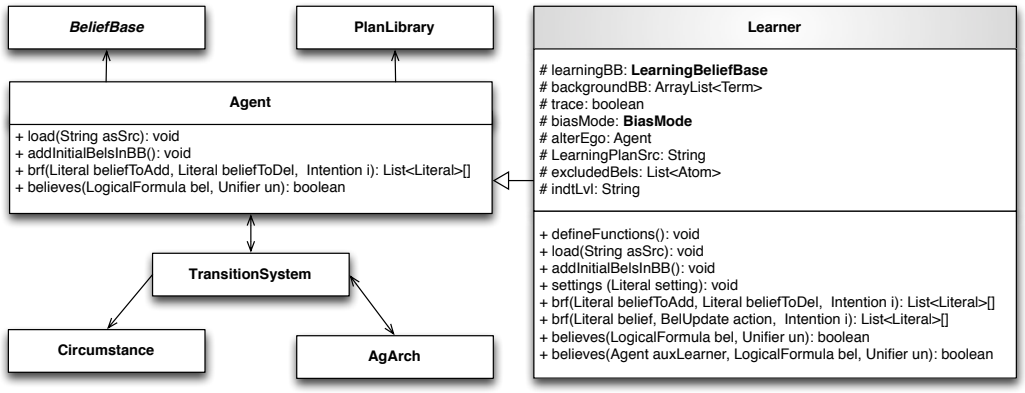


Figura 5.6 Diagrama de clase de un agente aprendiz *Learner*, Adaptado de Bordini et al. (2007)

La clase de agente *Learner* especifica tres tipos de datos enumerados: *BelUpdate*, *BiasMode* y *SettingsAtt*, los cuales se explicarán a lo largo de esta sección. A diferencia de un agente convencional, un agente aprendiz está compuesto por atributos que son usados en el proceso de aprendizaje y se enlistan a continuación:

- *trace*. Variable *booleana* que indica si se pueden imprimir mensajes en pantalla desde las acciones internas que ejecute el agente. Puede ser configurada agregando la creencia *jildt_settings(trace,Val)*, donde $Val \in \{true, false\}$. Por defecto, el valor de este atributo es *false*.
- *excludedBels*. Es una lista de átomos $(List < Atom >)$ que almacena los funtores[4] de las creencias que no formaran parte de los ejemplos de entrenamiento. Se configura desde el programa de agente agregando la creencia *jildt_settings(excludeBels,Func)*, donde *Func* es una lista de funtores. Por defecto, la lista es vacía.
- *indtLvl*. Una variable de tipo cadena que almacena el nivel de programación en el que se ejecutará la inducción de árboles lógicos de decisión. Se configura agregando la creencia *jildt_settings(inductionLevel, Lvl)*, donde $Lvl \in \{java, agentSpeak\}$. Por defecto, el nivel de inducción es *java*.
- *LearningPlanSrc*. Este atributo define la ruta del archivo origen donde se definen los planes de aprendizaje. Se puede configurar agregando la creencia *jildt_settings(learningPlansSrc, Src)*, donde *Src* es la ruta del archivo. Por defecto, los planes de aprendizaje son cargados del archivo *jildt/learningPlans.asl*.

[4] Un *funtor* es el átomo que relaciona los *n* términos que forman una literal. Por ejemplo, el *funtor* de la literal *on(b,a)* es *on* e indica que la relación entre sus términos es que el bloque *b* está sobre el bloque *a*.

- *biasMode*. Variable del tipo *Learner.BiasMode* que indica el modo en que el agente formará y utilizará el sesgo del lenguaje. Puede ser configurado añadiendo la creencia *jildt_settings(biasMode, BM)*, donde $BM \in \{manual, automatic\}$. Por defecto, el valor de este atributo es *Learner.BiasMode.automatic*.
- *learningBB*. Este atributo es una instancia de una Base de Creencias personalizada en la libreria JILDT, la cual se describe mas adelante en la sección 5.3.1. Esta base de creencias es pieza fundamental en el diseño de un agente aprendiz, ya que permite separar las creencias del agente relacionadas con el proceso de aprendizaje, de las creencias que el agente tiene de su entorno y problemática para la que fue implementado.
- *backgroundBB*. Un lista dinámica de términos $(ArrayList < Term >)$ que apunta a las reglas del agente. Como se mencionó en la seccioón 3.3.1, la representación de los ejemplos de entrenamiento para construir árboles lógicos de decisión se basa en interpretaciones. A estas interpretaciones es unido el conocimiento general del agente, por lo que tener una lista que apunte directamente a este conocimiento mejora la eficiencia del agente, evitando una búsqueda exhaustiva de las reglas en la base de creencias del agente.
- *alterEgo*. Una instancia de la clase *Agent* que apunta a la base de creencias de aprendizaje *learningBB*. El objetivo de este atributo es auxiliar en la búsqueda de una literal de aprendizaje, ya que por defecto, los métodos en un agente común de Jason buscan las creencias únicamente en la base de creencias implementada por defecto (Ver figura 5.7).

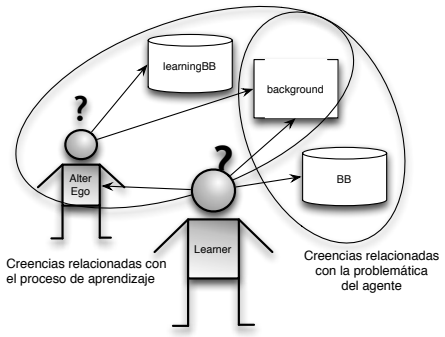


Figura 5.7 Modo de operación de un agente aprendiz con respecto a la consulta de sus creencias.

La clase de agente *Learner* hereda la mayoría de los métodos definidos en la clase *Agent*, pero también sobrescribe algunos otros. El método **load** es sobrescrito para cargar los planes de aprendizaje, una vez se haya ejecutado el método *load(String asSrc)* de la clase *Agent*; el método **addInitialBelsInBB()** es sobrescrito para agregar las reglas que forman el conocimiento general del agente a la lista *backgroundBB*. Esta asociación se hace únicamente al agregar las creencias iniciales, ya que los agentes Jason no agregan reglas dinámicamente.

Dos métodos fueron sobrescritos y sobrecargados para manejar las creencias relacionadas con el aprendizaje. El método de revisión de creencias **brf** se sobrescribe para preguntar si la creencia que se agregará o eliminará es una creencia relacionada con el aprendizaje. Si así es, ejecuta el método sobrecargado `brf(Literal belief, BelUpdate action, Intention i)`, pasándo como argumento una variable del tipo enumerado *Learner.BelUpdate*, que indica si la variable se agregará o eliminará de la base de creencias de aprendizaje; de no tratarse de una creencia de aprendizaje, se ejecuta el método `brf(Literal beliefToAdd, Literal beliefToDel, Intention i)` definido en la clase *Agent*. El otro método sobrescrito y sobrecargado es el método **believes**, el cual pregunta si una creencia es consecuencia lógica de las creencias del agente. Si se pregunta por una creencia de aprendizaje, se ejecuta el método sobrecargado `believes(Agent auxLearner, LogicalFormula bel, Unifier un)`, el cual recibe como argumento la variable *AlterEgo*, que apunta a la base de creencias de aprendizaje, que es donde busca la creencia consultada; en caso de no tratarse de una creencia de aprendizaje, se ejecuta el método *believes(LogicalFormula bel, Unifier un)* implementado en la clase *Agent*.

El método **defineFunctions()** define las funciones matemáticas implementadas en *jildt.tilde.math* y es ejecutado al inicio del método `load`. El método **settings (Literal setting)** guarda la configuracion recibida en su argumento. Se ejecuta desde el método `brf` cada vez que una literal del tipo *jildt_settings/2* es añadida. El tipo de dato enumerado *Learner.SettingsAtt* es usado en este método para asociar el tipo de configuración en su argumento con el atributo correspondiente en la clase de agente *Learner*. Por último, se implementan los métodos *setters* y *getters* correspondientes a cada atributo.

5.3.1. LearningBeliefBase

Como se mencionó anteriormente, las creencias del agente relacionadas con el aprendizaje son separadas de las creencias que el agente tiene sobre la problemática para la cual fue diseñado, para ello se extiende el estado mental de un agente *Learner* para incluir una base de creencias de aprendizaje. Esta base de creencias incluye principalmente ejemplos de entrenamiento *(jildt_example/3)*, directivas que forman el sesgo del lenguaje *(jildt_rm/1)* y configuraciones *(jildt_setings/2)*. Para el caso de los ejemplos de entrenamiento, el uso de las literales puede llegar a ser complicado, ya que el tamaño de estas puede ser bastante amplio. Para ello se ha implementado la clase *LearningBeliefBase*, la cual define una base de creencias personalizada, que hereda sus métodos de la clase *DefaultBeliefBase* que a su vez implementa la interfaz *BeliefBase*. La imagen 5.8 muestra el diagrama de la clase *LearningBeliefBase* y sus clases auxiliares *BelieveEntry* y *LiteralWrapper*, mismas que se encuentran en el paquete *jildt.bb*.

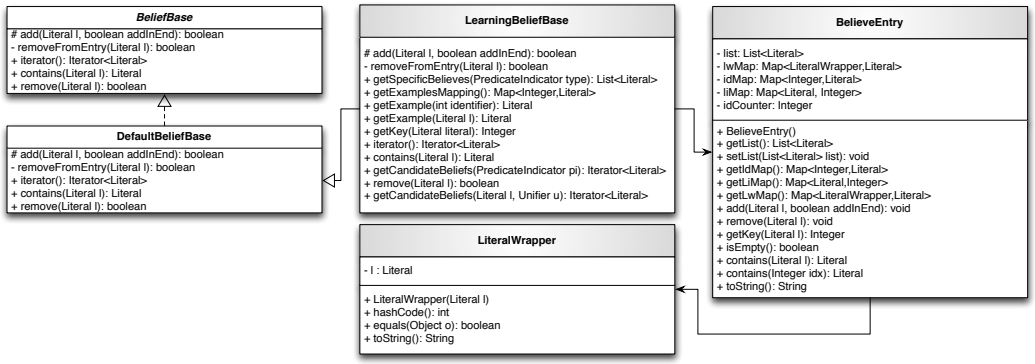


Figura 5.8 Diagrama de clases de *LearningBeliefBase* y sus clases auxiliares.

Una base de creencias contiene una instancia de la clase *Map* que mapea un predicado[5] con una instancia de la clase *BelieveEntry* (ver figura 5.9). Esta clase define dos tipos de mapeos, uno entre la *Literal* y su índice, y otro entre el índice y la *Literal*. Los índices son relativos, a cada predicado en la base de creencias corresponde un mapeo de índices diferente. Además, se agrega una anotación que indica cuantas veces se ha agregado la misma creencia. Esta última característica nos abre la puerta criterios probabilísticos de desempate al obtener el mejor candidato en la construcción de un árbol lógico de decisión, y forma parte de los trabajos a futuro de este proyecto. La clase *LiteralWrapper* únicamente sirve como auxiliar para buscar una *Literal* a partir de otra.

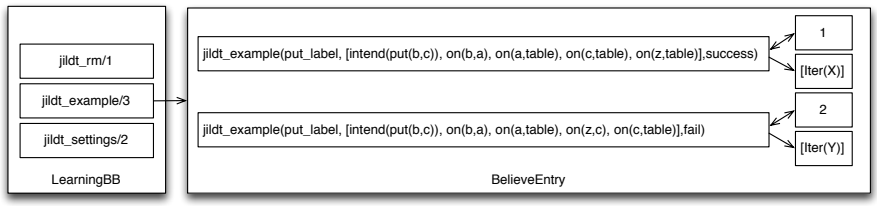


Figura 5.9 Composición de una entrada dentro de una base de creencias de aprendizaje.

[5] Un *predicado* es una estructura formada por el funtor y la aridad de una literal, por ejemplo, el predicado de *on(X,Y)* es *on/2*.

5.3.2. Clase de agente SingleMindedLearner

Se ha implementado la clase de agente **SingleMindedLearner**, la cual extiende la funcionalidad de un agente *Learner* para adoptar una estrategia de compromiso racional *(SingleMinded commitment)* a partir del aprendizaje obtenido a través de la inducción de árboles lógicos de decisión. Esta implementación forma el caso de estudio de este proyecto[6].

La clase de agentes *Learner* permite definir agentes capaces de redefinir el contexto de sus planes toda vez que haya ocurrido un fallo en la ejecución de un plan. De este modo, este tipo de agentes aprende a reconsiderar sus acciones antes de adoptar una intención futura, cada vez que haya fallado en la ejecución de un plan. De modo muy similar, los agentes definidos como instancias de la clase **SingleMindedLearner** son capaces, además de redefinir el contexto de sus planes, de adquirir conocimiento general que expresa cuándo es racional abandonar una intención una vez que percibe que no podrá finalizarla con éxito. Esta clase de agente implementa una estrategia de compromiso racional, como la descrita en la página 29, mediante reglas de abandono, donde el cuerpo de estas reglas se obtiene de las ramas del árbol lógico de decisión que llevan a un nodo hoja etiquetado como *fail*.

El comportamiento de un agente tipo *singleMindedLearner* se rige por las siguientes reglas:

1. Si existe un plan aplicable, y el contexto del plan es consecuencia lógica del conjunto de creencias del agente, entonces ejecuta el plan seleccionado, y si la ejecución es satisfactoria, agrega un ejemplo de entrenamiento etiquetado como *success*.
2. Si al momento de ejecutar el plan seleccionado ocurre un fallo, entonces se ejecuta un plan de aprendizaje, y en caso de haber aprendido un contexto diferente, lo redefine y agrega al menos una regla de abandono que indica cuándo es racional abandonar una intención. Una regla de abandono esta formada de la siguiente manera: La cabeza es una literal *drop(I)*, donde *I* unifica con la intención actual del agente; el cuerpo está formado por la consulta *.intend(I)* y la conjunción de literales que forman el camino recorrido desde el nodo raíz de un árbol hasta un nodo hoja etiquetado como *fail*. El cuadro 5.9 muestra las reglas de abandono obtenidas a partir del árbol lógico de decisión mostrado en la figura 4.1.

```
1 drop(put(A,B)) :- .intend(put(X,Y)) & clear(A) & not(clear(B)).
2 drop(put(A,B)) :- .intend(put(X,Y)) & not (clear(A)).
```

Cuadro 5.9 Reglas de abandono formadas a partir del árbol mostrado en la figura 4.1.

[6] Al no ser parte del núcleo de la librería JILDT, la clase de agente y las funcionalidades implementadas se encuentran definidas en un paquete llamado *sml*, incluido en la distribución de JILDT.

3. Si posteriormente, al momento de tratar de ejecutar una intención *I*, el agente percibe que la literal *drop(I)* es consecuencia lógica de su conjunto de creencias, entonces dispara el evento *+dropIntention(I)*, que ejecuta el plan etiquetado como `@dropPlan` (Ver cuadro 5.10). Este plan fuerza al agente a abandonar la intención *I*, usando la acción interna `.drop_intention(I)` (línea 4) que es parte de las acciones internas incluidas en la distribución de *Jason*.

```
1 @dropPlan
2 +dropIntention(I) : true <-
3   .print("Wow!! I'm sorry, I have to abandon my intention");
4   .drop_intention(I).
```

Cuadro 5.10 Plan de abandono de intenciones en el agente *singleMindedLearner*.

El mecanismo anterior se ejemplifica en la figura 5.10. En este caso, el ciclo de razonamiento es modificado para percibir cuando es racional abandonar una intención ($SelInt_3$).

$$(\textbf{SelInt}_3) \quad \frac{Ag_{bs} \models drop(I)}{\langle ag, C, M, T, SelInt \rangle \rightarrow \langle ag, C', M, T, ClrInt \rangle}$$
$$\text{donde: } \mathcal{C}'_{\mathcal{I}} = \mathcal{C}'_{\mathcal{I}} / \{\mathcal{I}\}$$

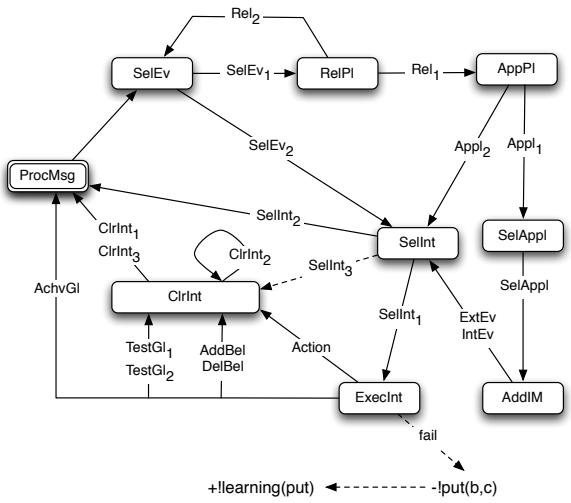


Figura 5.10 Modo operacional de un agente tipo *singleMindedLearner*.

Los planes de aprendizaje de un agente definido como instancia de la clase *SingleMindedLearner* se encuentran en el archivo *smLearnPlans.asl* en el paquete *sml*. Esta configuración es posible definiendo la literal *jildt_settings(learningPlansSrc, "../sml/smLearnPlans.asl")* en el programa de agente. El cuadro 5.11 muestra el plan de aprendizaje modificado para un agente *SingleMindedLearner*, el cual agrega la acción interna `sml.addDropRule(P,Tree)` (línea 9), implementada para agregar las reglas de abandono. La variable *P* indica el plan para el que se ejecuto el proceso de aprendizaje, y la variable *Tree* es el árbol computado. El resto de los métodos descritos en la sección 5.2 se mantienen iguales.

```
1 @learning
2 +!learning(P,Tree): true <-
3   .print("Trying to learn a better context...");
4   !executeTilde(P,Tree);
5   jildt.parseLearnedCtxt(Tree,LC);
6   jildt.displayTree(Tree, gui);
7   .print("Learned context for ",P," is ", LC);
8   jildt.setContext(P, LC);
9   sml.addDropRule(P,Tree).
```

Cuadro 5.11 Plan de aprendizaje para un agente *SingleMindedLearner*.

5.4. Otras clases y funciones

Además de las clases que se han mencionado a lo largo de este capítulo, la librería JILDT cuenta con otras clases. El paquete *jildt*, aparte de las acciones internas y el programa de agente que define los planes aprendizaje, implementa dos clases: *Functions* y *TildeNode*(Ver figura 5.11).

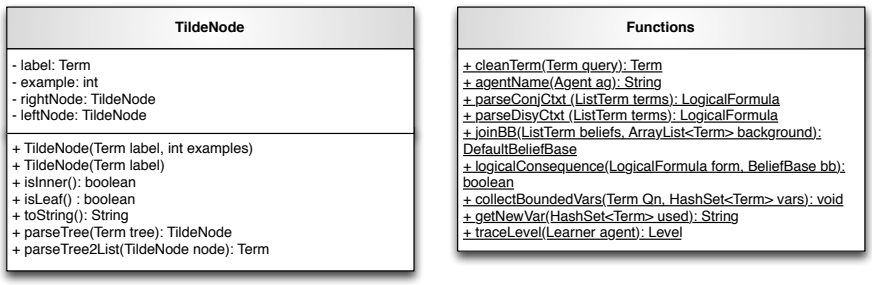


Figura 5.11 Diagrama de clases de *TildeNode* y *Functions*.

La clase **Functions** contiene funciones estáticas de uso general, las cuales son utilizadas en más de una acción interna, de ahı' su implementación como métodos estáticos. La clase **TildeNode** implementa un *nodo* de árbol. Los métodos definidos en esta clase permiten consultar si un nodo es un nodo interno *(inner node)* o un nodo hoja *(leaf node)*. Dos métodos estáticos permiten convertir una instancia de la clase *TildeNode* en una instancia de la clase *Term* y viceversa, para garantizar una buena integración entre la representación de un árbol en Java y su representación en *AgentSpeak(L)*.

Por último, el paquete *jildt.gui* implementa una clase de interfaz gráfica de usuario, que funge como herramienta de visualización y despliega una ventana que muestra el árbol computado como el presentado en la figura 5.2 (derecha).

5.5. Resumen

JILDT es una librería programada en Java, bajo el paradigma de programación orientada a agentes que implementa Aprendizaje Intencional en agentes definidos como instancias de la clase *Learner*. Esto es posible gracias a un mecanismo en el que los planes originales son extendidos, para poder generar ejemplos de entrenamiento con base en el estado mental de los agentes y lidiar con ejecuciones fallidas ejecutando el proceso de aprendizaje. Las capacidades de los agentes son extendidas a través de nuevas acciones internas y funciones matemáticas que permiten conocer el estado BDI del agente y realizar acciones relacionadas al proceso de aprendizaje a través de un algoritmo de inducción de árboles lógicos de decisión, el cual es implementado en dos niveles de programación: *Java* y *AgentSpeak(L)*. El primero diseñado para ejecuciones eficientes en aprendizaje sin interacción social; el segundo abre las puertas a un aprendizaje colectivo.

Dos clases de agente personalizadas se presentaron en este capitulo, mismas que extienden las funciones básicas de un agente por defecto de *Jason*, para sustentar Aprendizaje Intencional e implementar una estrategia de compromiso racional *(SingleMinded)*. En el siguiente capítulo, se realizan algunos experimentos para probar la capacidad de aprendizaje de estas clases de agente.

Capítulo 6
Experimentos

En este capítulo se presentan los experimentos ejecutados para demostrar la funcionalidad de la libreria JILDT como soporte de Aprendizaje Intencional. Éstos, son realizados en el conocido **mundo de los bloques**, el cual es descrito en la primera sección de este capítulo. La segunda sección presenta un experimento que servirá para demostrar que la racionalidad de los agentes puede aumentar gracias al mecanismo de Aprendizaje Intencional sustentado por JILDT. Tres agentes son implementados para medir su racionalidad en este experimento: un agente por defecto de Jason, el cual no cuenta con un mecanismo de aprendizaje; un agente definido como instancia de la clase *Learner*, el cual cuenta con un mecanismo de aprendizaje y es capaz de redefinir el contexto de sus planes; y un agente definido como instancia de la clase *SingleMindedLearner*, el cual adquiere una estrategia de compromiso racional *Single Minded commitment*, y es capaz de aprender sus políticas de abandono.

Como se ha mencionado anteriormente, este proyecto es la extension de una trabajo preliminar, presentado en (González-Alarcón, 2010). La representación de las entradas del algoritmo de aprendizaje, y la implementación de éste mismo han sido objetivo principal de este proyecto, por ello, la tercera sección presenta un par de comparaciones de eficiencia. Tres aspectos han sido seleccionados para comparar eficiencia: el tiempo de ejecución del plan de aprendizaje, el consumo de memoria RAM y el almacenamiento en disco duro. Primero, se compara la nueva versión de JILDT contra su versión preliminar, ya que ambas ejecutan el algoritmo de aprendizaje como acción interna de Jason, con la diferencia de que la primera cuenta con una representación en primer orden de las entradas del algoritmo y la otra almacena estas entradas en ficheros; después de esto, se hace una comparación de eficiencia entre la ejecución del algoritmo de aprendizaje en un nivel de programación Java y un nivel de programación *AgentSpeak(L)*, ambas forman parte de la versión de JILDT reportada en este trabajo.

Los experimentos se ejecutaron en un servidor Intel I7-3930K con 6 núcleos, 48 GB de memoria RAM DDR3 y un disco duro de 1 TB. El sistema operativo es openSUSE 12.1 y la versión de Java es OpenJDK 64 bits 1.6.0_24.

6.1. Mundo de los bloques

El mundo de los bloques es uno de los dominios más empleado en planificación. Consiste en un conjunto de bloques dispuestos sobre una mesa o encima de otros bloques. Los bloques pueden ser apilados, pero únicamente se pueden apilar uno sobre otro. Hay un brazo mecánico[1] que puede levantar un bloque a la vez y dejarlo encima de la mesa o de otro bloque. El objetivo es construir una o mas pilas de bloques. Empleamos $on(X,Y)$ para indicar que el bloque X se encuentra sobre el objeto Y, y $clear(X)$ para indicar que el objeto X no tiene un bloque encima. La acción para mover el bloque X sobre el objeto Y se representa por $put(X,Y)$. La figura 6.1 muestra el ambiente simulado en Jason del mundo de los bloques, adaptado y simplificado de Bordini et al. (2007).

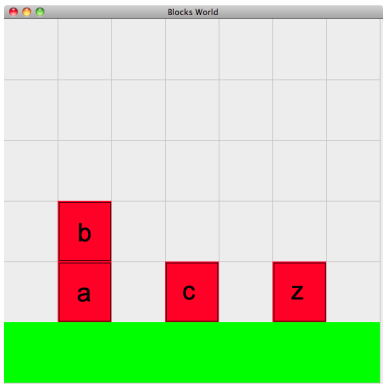


Figura 6.1 Mundo de los bloques simulado en Jason. Adaptado de Bordini et al. (2007)

6.2. Comportamiento racional

Se ha diseñado este experimento para comparar el comportamiento de un agente por defecto Jason *(default)*, un agente aprendiz *(learner)*, instancia de la clase *Learner* y un agente con estrategia de compromiso racional *(singleMinded)*, instancia de la clase *SingleMindedLearner*. El agente *learner* es capaz de aprender las razones para adoptar un plan, mientras que el agente *singleMinded* aprende las razones para adoptar un plan y políticas para abandonar intenciones. El agente *default* no cuenta con capacidades de aprendizaje.

[1] La abstracción del mundo de los bloques supone la existencia de este brazo que mueve los bloques, sin embargo, en la implementación obviamos la existencia de este brazo mediante la acción de mover un bloque sobre otro objeto.

Por cuestiones de simplicidad, estos tres agentes definen el plan $put(X,Y)$ como se muestra en el cuadro 6.1. Todos creen ciegamente que pueden poner cualquier bloque en cualquier lugar(línea 11), y que ésta es su única competencia. Además perciben del ambiente mostrado en la figura 6.1: $on(b,a)$, $on(a,table)$, $on(c,table)$ y $on(z,table)$. El conocimiento general de estos agentes está dado por la creencia $clear/1$(líneas 2-3): la mesa siempre está libre, al igual que cualquier objeto que no tenga un bloque sobre él.

```
1  // Beliefs
2  clear(X) :- not(on(_,X)).
3  clear(table).
4  // Perceptions of the environment
5  on(b,a).
6  on(a,table).
7  on(c,table).
8  on(z,table).
9  // Plans
10 @put
11 +!put(X,Y) : true <- move(X,Y).
```

Cuadro 6.1 Un agente simplificado del mundo de los bloques.

El experimento se ejecuta como se ilustra en la figura 6.2. El agente experimentador *experimenter* pide a cualquiera de los otros agentes que coloquen el bloque b sobre el bloque c, pero con cierta probabilidad, éste introduce ruido en el experimento $p(N)$, colocando el bloque z sobre el bloque b, o sobre el bloque c. También hay una probabilidad de latencia $p(L)$, para introducir el ruido, es decir, el experimentador podría poner el bloque z antes o después de solicitar a otro agente poner el bloque b sobre el bloque c. Esto significa que los otros agentes pueden percibir el ruido antes o al mismo tiempo de adoptar la intención de poner b en c. Cuando el agente *experimenter* coloca el bloque z sobre el bloque c, permite a los agentes aprendices aprender que para poner un bloque sobre otro, éste último debe estar libre. En la práctica un agente debería saber que ambos bloques deben estar libres, por ello, cuando el agente *experimenter* mueve el bloque z sobre el bloque c, un agente aprendiz aprende también que el bloque que pretende mover debe estar libre. La probabilidad de opción de ruido $p(O)$ es de 50 %, y permanece igual en todos los experimentos, ya que su variación no afecta directamente a la racionalidad del agente, y el agente debe conocer ambas opciones de configuración para aprender ambas condiciones.

En el cuadro 6.2 se muestra el código del SMA en el que corre el experimento. La comunicación se da únicamente entre el agente *experimenter* y cualquiera de los otros agentes, es decir, *default*, *learner* y *singleMinded* no tienen interacción entre ellos; además, el agente *experimenter* pide poner el bloque b sobre el bloque c únicamente a un agente a la vez.

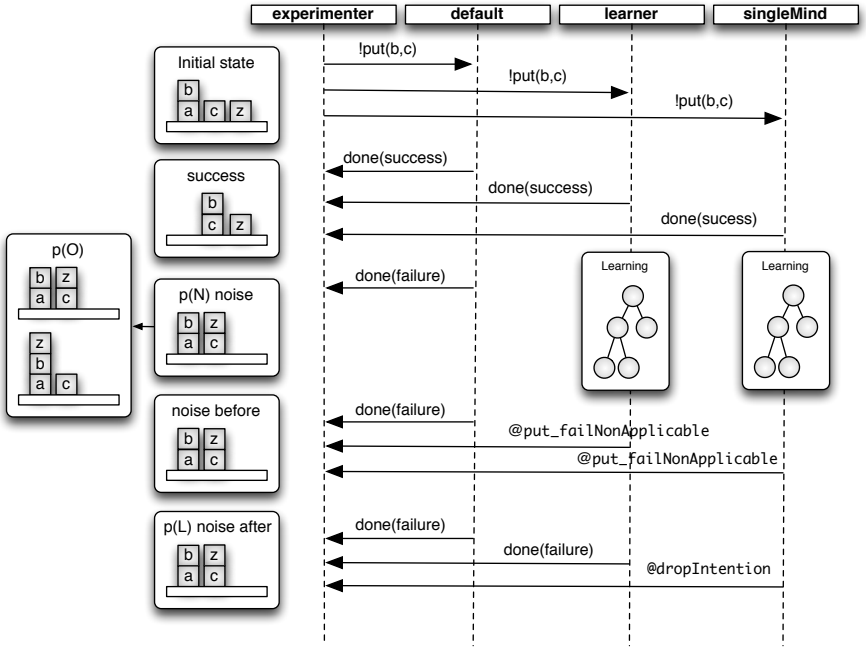


Figura 6.2 Procedimiento experimental en el mundo de los bloques.

```
MAS jildt_bwl {
   infrastructure: Centralised

   environment: BlocksEnv.BlocksWorld(1,100,yes)

   agents:
       experimenter;
       learner agentClass jildt.agent.Learner;
       singleMinded agentClass sml.singleMindedLearner;
       default;

   directives: learnablePlans=jildt.LearnablePlans;
   aslSourcePath: "src/asl";
}
```

Cuadro 6.2 Código del MAS para el mundo de los bloques.

En el cuadro 6.3 se muestra una parte de los resultados numéricos, los cuales son el promedio de 10 repeticiones, donde cada repetición ejecuta 100 veces el procedimiento descrito anteriormente en la figura 6.2 para los agentes *default*, *learner* y *singleMinded*. Los resultados que se muestran corresponden a una probabilidad de latencia de 50 % y probabilidades de ruido $p(N)$ que varían y toman los valores (10 %, 30 %, 50 %, 70 %, y 90 %).

Dos columnas corresponden al tipo de comportamiento mostrado en el experimento: *racional* e *irracional*. El desempeño del agente se considera más o menos racional de la siguiente manera:

- Abandonar la intención debido a que ocurrió un error en la ejecución, ya sea *antes* o *después* de adoptar la intención de ejecutar el plan se considera un comportamiento **irracional**.
- *Rechazar* adoptar una intención porque el plan no es aplicable; *abandonarla* cuando existe una razón para creer que ésta fallará; y lograr poner *b* en *c* con *éxito* se consideran comportamientos **racionales**.

Agente	**p(N)**	**Irracional**			**Racional**			
		Después	**Antes**	**Total**	**Rechazo**	**Abandono**	**Éxito**	**Total**
default	10	5.1	5.1	10.2	0	0	89.8	89.8
learner	10	4.8	1.1	5.9	3.7	0	90.4	94.1
singleMinded	10	0.3	0.8	1.1	3.7	4.9	90.3	98.9
default	30	14.3	15	29.3	0	0	70.7	70.7
learner	30	14.3	1.1	15.4	16.6	0	68	84.6
singleMinded	30	0.6	1.4	2	12.7	14.4	70.9	98
default	50	24.1	24.5	48.6	0	0	51.4	51.4
learner	50	22.3	1.1	23.4	25.1	0	51.5	76.6
singleMinded	50	1.9	0.7	2.6	23.7	22.8	50.9	97.4
default	70	39.7	35.5	75.2	0	0	24.8	24.8
learner	70	35.1	1.6	36.7	33.2	0	30.1	63.3
singleMinded	70	1.2	1.1	2.3	34.9	33.6	29.2	97.7
default	90	44.4	46.4	90.8	0	0	9.2	9.2
learner	90	46.1	5.5	51.6	38	0	10.4	48.4
singleMinded	90	3.6	4.6	8.2	42.2	41	8.6	91.8

Cuadro 6.3 Resultados experimentales para una probabilidad de latencia $P(L) = 0{,}5$ y diferetentes probabilidades de ruido $P(N)$.

Los valores más bajos configuran entornos menos dinámicos, libre de sorpresas, lo que da como resultado un entorno efectivamente observable. Como se puede observar, el agente *default* muestra un comportamiento racional con valores bajos de ruido, pero ésta empieza a decrementar comforme la probabilidad de que aparezca ruido crece. De forma similar, la racionalidad del agente *learner* disminuye conforme aumenta la probabilidad de ruido, pero con un comportamiento más racional que el agente *default*, debido a la adquisición de conocimiento obtenida; el agente *singleMinded* muestra un comportamiento irracional muy bajo en todas las posibles combinaciones de valores. La figura 6.3 resume el resultado de los experimentos ejecutados, donde las probabilidades de ruido y la latencia varían y toman los valores de $\{90\%, 70\%, 50\%, 30\%, 10\%\}$. Como era de esperar el desempeño del agente *default* es inversamente proporcional a la probabilidad de ruido, independientemente de la probabilidad de latencia.

El agente *learner* reduce la irracionalidad cuando el ruido aparece antes de la adopción del plan como intención, porque eventualmente aprende que para poner un bloque X sobre un bloque Y, tanto el bloque X como el bloque Y deben estar libres.

```
+!put(X,Y) : clear(X) & clear(Y) <- move(X,Y).
```

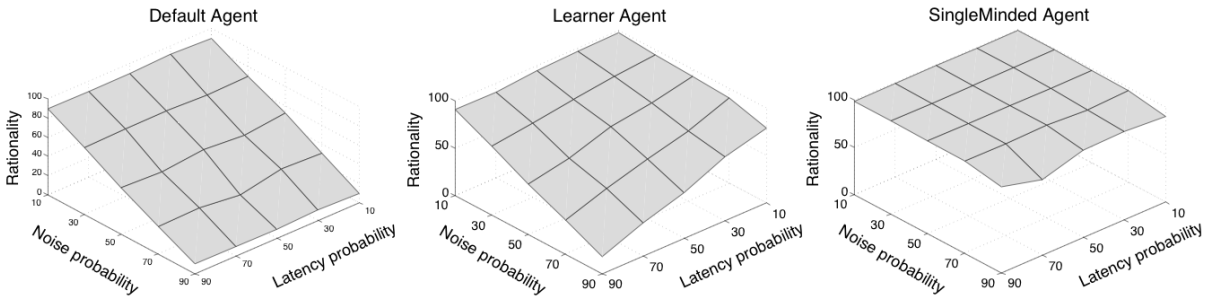


Figura 6.3 Resultados experimentales de desempeño. Izquierda: Agente *default*. Centro: Agente *learner*. Derecha: Agente *singleMinded*.

De esta manera, el agente *learner* puede rechazar la intención de poner b en c si percibe que b, c, o incluso ambas no están libres. Por lo tanto, para probabilidades bajas de latencia su comportamiento es mejor que el del agente *default*, pero por supuesto, su rendimiento decae conforme la probabilidad de latencia va aumentando, y más importante aún: no hay nada que hacer si percibe el ruido después de haber adoptado la intención. Por otro lado, el agente *singleMinded*, además de aprender a rechazar intenciones sin futuro exitoso, aprende las siguientes reglas para abandonar la intención cuando el bloque X o el bloque Y no están libre.

```
drop(put(X,Y)) :- .intend(put(X,Y)) & not (clear(X)).
drop(put(X,Y)) :- .intend(put(X,Y)) & clear(X) & not (clear(Y)).
```

Estas reglas indican al agente *singleMinded* que debe abandonar la intención de poner un bloque X sobre un bloque Y, siempre que lo esté intentando y perciba que el bloque X no está libre (línea 1); o siempre que lo esteé intentando y el bloque Y no se encuentre libre, a pesar de estar libre X. Estas reglas, como se ha mencionado en el capítulo anterior, se construyen a partir del árbol aprendido, y representan las ramas que van desde el nodo raíz hasta un nodo hoja etiquetado como *fail*.

Cada vez que un agente definido como *singleMindedLearner* va a ejecutar una intención, primero verifica que no haya razones para abandonar la intención existente, de lo contrario la intención es abandonada. Así, cuando

el agente *singleMinded* ya tiene la intención de poner *b* en *c*, y el experimentador coloca el bloque *z* en *b* o en el bloque *c*, el agente *singleMinded* racionalmente abandona su intención. De hecho, el agente singleMinded sólo falla cuando está listo para ejecutar la acción *move* y aparece el ruido sin que el agente conozca aún alguna política de abandono.

El cuadro 6.4 muestra parte de la salida del experimento con los agentes *experimenter* y *singleMinded*, donde se pueden ver los mensajes al momento de aprender un un nuevo contexto, y aquellos mensajes desplegados cuando se abandona o rechaza una intención.

```
[experimenter] ====================================================
[experimenter] Experiment Number: 10/100
[experimenter] Noise : on(z,c)
[experimenter] Latency: Before
[experimenter] --------------------------
[singleM] current context for plan: @put is clear(X)
[singleM] Trying to learn a better context...
[singleM] Learned context for put is [[clear(X),clear(Y)]]
[experimenter] Agent singleMinded has failed
[experimenter] ====================================================
[experimenter] Experiment Number: 11/100
[experimenter] Experiment without noise
[experimenter] --------------------------
[singleM] current context for plan: @put is (clear(X) & clear(Y))
[singleM] Yeah, I did the task succesfully
[experimenter] Agent singleMinded has succeed.
[experimenter] ====================================================
[experimenter] Experiment Number: 12/100
[experimenter] Noise : on(z,c)
[experimenter] Latency: Before
[experimenter] --------------------------
[singleM] Plan +!put produced an irrelevant failure.
[experimenter] Agent singleMinded has failed because it didn't take the intention
[experimenter] ====================================================
[experimenter] Experiment Number: 13/100
[experimenter] Noise : on(z,b)
[experimenter] Latency: After
[experimenter] --------------------------
[singleM] current context for plan: @put is (clear(Y) & clear(X))
[singleM] Wow!! I'm sorry, I have to abandon my intention
[experimenter] Agent singleMinded has failed because it abandoned the intention
[experimenter] ====================================================
```

Cuadro 6.4 Salida de la ejecución del experimento con el agente *singleMinded*

En la página 13 se menciona que la racionalidad de un agente depende de cuatro factores, los cuales pueden ser aplicados a las clases de agente *Learner* y *SingleMindedLearner*. La **medida de desempeño** está dada por la suma de las ejecuciones que el agente logró concluir satisfactoriamente, más la cantidad de veces que rechazó una intención dado que consideró que no era posible concluirla, más las veces que abandonó una intención cuando detectó que no era posible concluirla; ambos agentes son capaces de **percibir** su entorno antes de adoptar una intención, incluso cuando éste se encuentre en movimiento; el **conocimiento del medio** depende de la implementación del agente, como ejemplo, el agente del mundo de los bloques mencionado en la página 32 tiene el conocimiento de que un objeto está libre, si no existe algún bloque sobre éste; las **habilidades** del agente son adaptativas en el sentido de que ambos agentes van aprendiendo cuando es posible ejecutar una acción y cuando no lo es.

Considerando la medida de desempeño mencionada anteriormente, se puede concluir que, un agente de la clase *Learner* es más racional que un agente *default* de *Jason* (ver la figura 6.4), ya que un agente *default* no es capaz de redefinir los contextos de sus planes y rechazar intenciones; sin embargo, un agente de la clase *SingleMindedLearner* es aún más racional que un agente de la clase *Learner*, ya que el primero es capaz de abandonar una intención cuando percibe que no podrá finalizarla con éxito.

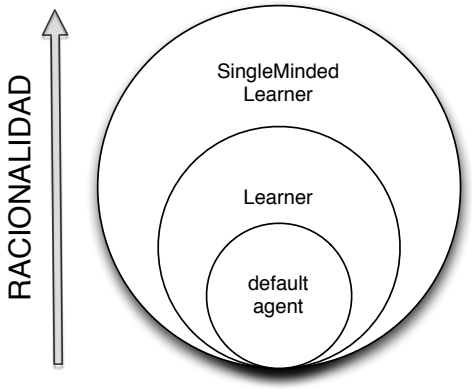


Figura 6.4 Niveles de racionalidad de agentes *default*, *Learner* y *SingleMindedLearner*.

6.3. Eficiencia

Esta sección presenta dos experimentos realizados para probar la eficiencia de JILDT. Primero se presenta una comparación del tiempo de ejecución, consumo de memoria RAM y consumo de disco duro entre la versión preliminar de JILDT y la versión a nivel Java reportada en este proyecto. En ambas versiones se ejecuta el algoritmo de aprendizaje como una acción interna de Jason, con la diferencia de que en la primera, las entradas del algoritmo de aprendizaje están almacenadas en ficheros, mientras que la segunda cuenta con una representación en primer orden de las entradas del algoritmo, además de una clase de agente mas robusta.

El experimento se ejecuta de la misma manera a como se presenta en la figura 6.2. En este caso, únicamente se trabaja con los agentes *experimenter* y *learner*. El agente *default* es descartado, por no contar con un mecanismo de aprendizaje, mientras que el agente *singleMinded* se descarta debido a que es bastante adaptable a su entorno, y por lo tanto no se ejecutaría el proceso de aprendizaje las veces necesarias para medir los recursos empleados durante el proceso de aprendizaje. El agente *learner* es idóneo para realizar este tipo de experimento, ya que a pesar de redefinir el contexto de sus planes, aún esta en posibilidades de ejecutar el proceso de aprendizaje, siempre que aumente la probabilidad de que el ruido aparezca después de haber adoptado una intención.

Se realizaron 10 experimentos de 100 ejecuciones cada uno, para valores $p(N) = 50\,\%$ y $p(L) = 25\,\%, 50\,\%$ y $75\,\%$. El cuadro 6.5 muestra los resultados obtenidos: *N* es el numero de ocasiones en que se ejecutó el proceso de aprendizaje en cada experimento. El tiempo es medido en milisegundos y el consumo de memoria RAM y de disco duro en Kilobytes.

	JILDT Preliminar				**JILDT Java**			
p(L)	**N**	**TIEMPO (Ms)**	**RAM (Kb)**	**HDD (Kb)**	**N**	**TIEMPO (Ms)**	**RAM (Kb)**	**HDD (Kb)**
0.25	14	72.64	6521.58	40.29	13	58.82	6338.52	0.00
0.50	24	63.71	4180.02	40.16	26	55.70	4089.37	0.00
0.75	40	64.68	2639.73	38.82	38	48.46	2978.88	0.00

Cuadro 6.5 Resultados obtenidos comparando la versión preliminar de JILDT contra la nueva versión de JILDT en un nivel de programación Java.

Las figuras 6.5 y 6.6 muestran una representación gráfica de los resultados obtenidos al hacer esta primera comparación. Como se muestra en la gráfica, el número de ocasiones en que se ejecuta el proceso de aprendizaje va aumentando conforme aumenta la probabilidad de latencia. Esto se debe, como se mencionó anteriormente a que el agente *learner*, a pesar de redefinir el contexto de sus planes sigue fallando cuando se encuentra con problemas una vez adoptada una intención.

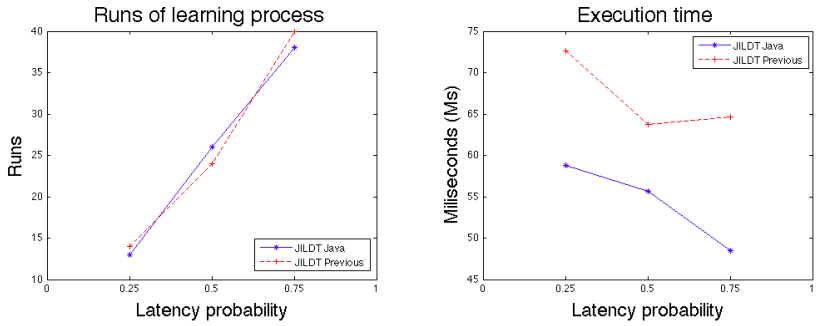


Figura 6.5 Comparación de ejecuciones del proceso de aprendizaje y de tiempo de ejecución entre la versión preliminar de JILDT y la nueva versión de JILDT en un nivel de programación Java.

El tiempo de ejecución se reduce en la versión de JILDT reportada en este trabajo. Esto se debe principalmente a que se reduce el tiempo de acceso a los archivos que formaban las entradas del algoritmo de aprendizaje, ya que estas ahora forman parte del estado mental del agente. En el caso de la versión preliminar de JILDT, su tiempo de ejecución se mantiene mas o menos constante cuando la probabilidad de latencia es alta, pero aumenta cuando ésta es baja; en el caso de JILDT Java se reduce el tiempo de ejecución conforme aumenta la probabilidad de latencia. Esto responde a que a pesar de que el proceso de aprendizaje se ejecute en más ocasiones, esta información se sigue manteniendo en memoria RAM. Es por ello que el uso de memoria RAM es muy parecido en ambos casos (ver figura 6.6). A pesar de consumir prácticamente la misma cantidad de memoria RAM, esta nueva versión de JILDT tiene la ventaja de no consumir espacio en el Disco Duro, a diferencia de su version preliminar.

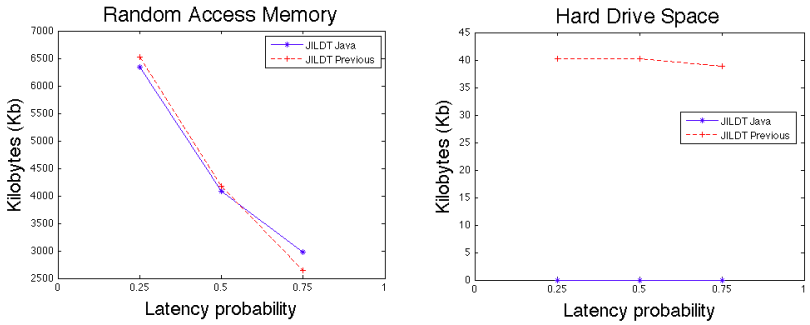


Figura 6.6 Comparación del consumo de memoria RAM y de espacio en disco duro entre la versión preliminar de JILDT y la nueva versión de JILDT en un nivel de programación Java.

La segunda comparación se hace en relación con el tiempo de ejecución del proceso de aprendizaje y el consumo de memoria RAM entre los niveles de programación Java y *AgentSpeak(L)* de la nueva versión de JILDT. Las probabilidades y ejecuciones son las mismas que el experimento anterior. El consumo de Disco Duro no es medido, ya que ninguno de estos niveles hace uso de archivos. El cuadro 6.6 muestra los resultados obtenidos.

p(L)	Java		*AgentSpeak(L)*	
	TIEMPO (Ms)	**RAM (Kb)**	**TIEMPO (Ms)**	**RAM (Kb)**
0.25	58.82	6338.52	73.99	7171.85
0.50	55.70	4089.37	64.46	4438.21
0.75	48.46	2978.88	57.88	3664.67

Cuadro 6.6 Resultados obtenidos comparando la versión preliminar de JILDT contra la nueva versión de JILDT en un nivel de programación Java.

La figura 6.7 muestra los resultados obtenidos. Como era de esperarse, el nivel de programación Java es más eficiente en tiempo de ejecución y consumo de memoria RAM. Como se mencionó en el capítulo 5, el nivel Java está diseñado para mejorar el desempeño computacional en agentes que se definen para aprender sin necesidad de tener alguna interación social. Sin embargo, el caso del nivel *AgentSpeak(L)*, a pesar de ser menos eficiente define un nivel más flexible, el cual abre las puertas a la definición de sistemas multiagentes que aprendan colectivamente, objetivo que corresponde a los trabajos futuros de este proyecto en su fase doctoral.

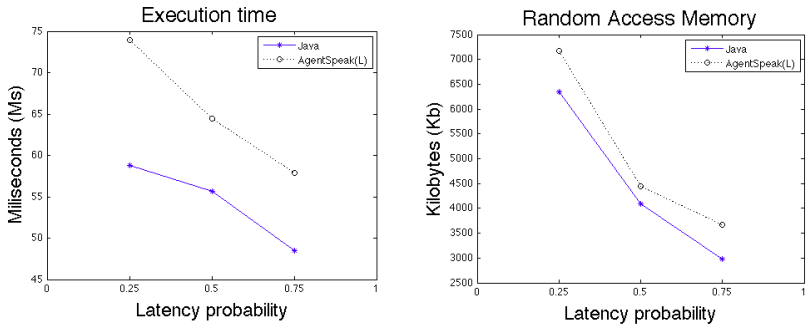


Figura 6.7 Comparación del tiempo de ejecución y consumo de memoria RAM entre las versiones Java y *AgentSpeak(L)* de JILDT

Los resultados presentados en la figura 6.8 muestran que el nivel de programación Java es más eficiente en tiempo de ejecución, mientras que el nivel de programación *AgentSpeak(L)* es el que más consume, tanto tiempo como memoria RAM. Éste es el coste que se paga por la flexibilidad social que ofrece este último nivel de programación.

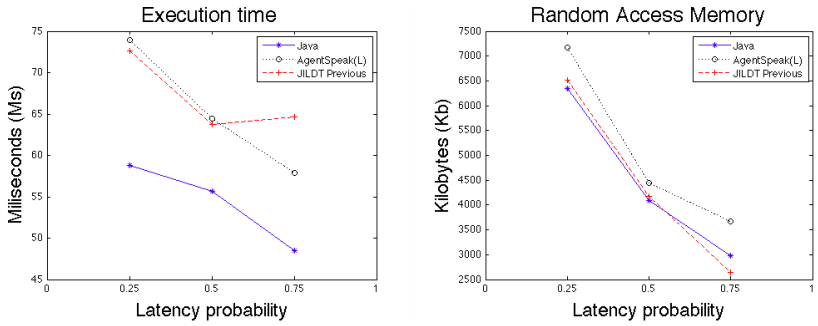


Figura 6.8 Comparación del tiempo de ejecución y consumo de memoria RAM entre la versión preliminar de JILDT y la nueva versión de JILDT en sus dos niveles de programación.

Tanto la versión en Java, como la versión *AgentSpeak(L)* de la nueva distribución de JILDT, tienen la ventaja de no emplear archivos, mismos que en problemas más complejos pueden llegar a ocupar mucho espacio. Esta es la mejora ofrecida al representar en primer orden las entradas del algoritmo de aprendizaje.

6.4. Resumen

Este capítulo presenta tres experimentos. El primero de ellos realiza una comparativa del comportamiento racional entre un agente *default* de Jason, un agente que aprende intencionalmente *(learner)* y un agente con una estrategia de compromiso racional *(singleMinded)*. Los resultados demuestran que hay un incremento de racionalidad desde el agente *default* hacia el agente *singleMinded* mostrando una mejoría en el nivel de adaptabilidad al ambiente.

El segundo y tercer experimento se enfocaron en hacer una evaluación al desempeño de la versión preliminar de JILDT mostrada en (González-Alarcón, 2010) y la nueva distribución de JILDT presentada en este trabajo, la cual cuenta con dos niveles de programación: Java y *AgentSpeak(L)*. Los resultados demostraron que una representación en primer orden de las entradas del algoritmo de aprendizaje mejoran el tiempo de ejecución, y el uso del Disco Duro en el nivel de programación Java, mientras que en el nivel de programación *AgentSpeak(L)* solo mejora el uso del Disco Duro. A pesar de esto, este último nivel de programación es más flexible y abre las puertas a un nivel de aprendizaje social, en el que los planes de aprendizaje son personalizados por cada agente. En la siguiente fase del proyecto, es decir, la fase doctoral, se buscará demostrar esta última sentencia.

Capítulo 7
Conclusiones

La librería JILDT *(Jason Induction of Logical Decision Trees)* proporciona las extensiones necesarias para Jason, el bien conocido intérprete de *AgentSpeak(L)*, para definir agentes que aprenden intencionalmente. Ésto es posible gracias a las bondades que ofrece el modelo BDI de agentes. El estado mental de los agentes permite formar ejemplos de entrenamiento expresados en lógica de primer orden, los cuales son necesarios para sustentar aprendizaje a través de la inducción de árboles lógicos de decisión. Este mecanismo de aprendizaje resultó ser idóneo gracias a que la naturaleza conjuntiva de sus ramas permite formar nuevos contextos, mismos que pueden ser modificados en planes que no hayan podido sido ejecutados satisfactoriamente.

A partir de la clase de agente aprendiz *(Learner)* implementada en JILDT, se ha podido implementar una clase de agente con compromiso racional *(SingleMindedLearner)*, la cual modifica su sistema de transición para habilitar abandono de intenciones. Básicamente, cada vez que el sistema está en en el paso *execInt* y la creencia `drop(I)` es consecuencia lógica de su conjunto de creencias y reglas, entonces se ejecuta una plan de abandono, donde elimina la intención que estaba tratando de ejecutar, y que, gracias al aprendizaje de nuevas reglas de abandono, ha podido razonar que no es posible llevar a cabo. Ahora es posible pensar en una semántica operacional formal para *AgentSpeak(L)* de compromiso basado en la reconsideración basada en políticas y el Aprendizaje Intencional.

Esta nueva versión de JILDT mejora las siguientes características, en relación con su versión preliminar presentada en (González-Alarcón, 2010).

- Las entradas del algoritmo de aprendizaje son representados como literales de primer orden, lo que homogeneiza el proceso de aprendizaje con el comportamiento habitual de un agente. Esta representación reduce el tiempo de ejecución, el consumo de memoria RAM y el espacio en Disco Duro al momento de ejecutar el proceso de aprendizaje.

- La clase de agente aprendiz define atributos relacionados directamente con el aprendizaje, entre ellos una lista que apunta a sus propias reglas para separar el conocimiento general de su conjunto de creencias, y atributos que almacenan las configuraciones del proceso de aprendizaje.
- JILDT incluye una base de creencias de aprendizaje que almacena aquellas creencias vinculadas directamente con el aprendizaje, como lo son ejemplos de entrenamiento *(jildt_example/3)*, las directivas *rmode* que definen el sesgo del lenguaje *(jildt_rm/1)* y las configuraciones definidas en el archivo de agente *(jildt_settings/2)*. Las entradas en esta base de creencias son indexadas e incluyen una anotación de iteración, que permite conocer el número de veces que se ha agregado la literal a la base de creencias.
- El conjunto de acciones internas ha sido enriquecido para ejecutar el aprendizaje en dos niveles de programación: Java y *AgentSpeak(L)*. La primera está diseñada para mejorar el desempeño de agentes que aprenden sin alguna interacción social; la segunda, por su parte, presenta una alternativa flexible que abre las puertas a la implementación de agentes que aprenden interactuando en algún entorno social.
- La percepción de las creencias del agente para formar ejemplos de entrenamiento de fallo es mejorada también. En vez de que estos ejemplos estén formados por las creencias que el agente tenia al adoptar la intención, se forman por las creencias que el agente percibe al momento de ejecutar el plan de fallo. Ésto último permite tener una noción más precisa del conocimiento del agente sobre su entorno al presentarse algún problema.
- Actualmente, una distribución de JILDT está disponible en `http://jildt.sourceforge.net/`, la cual incluye el ejemplo del mundo de los bloques y estrategia de compromiso racional, presentados en este trabajo.

Los resultados experimentales son muy prometedores. En comparación con los experimentos sobre compromiso mostrados por Kinny (1991), se observa que agentes definidos como instancias de las clases *Learner* y *SingleMindedLearner* son adaptables: eran confiados con respecto al plan *put*, y luego adoptaron una estrategia cautelosa después de haber tenido problemas con la ejecución de su plan. El uso de Aprendizaje Intencional proporciona la convergencia con el nivel adecuado de **confianza-cautela**, basado en la experiencia de los agentes. Los agentes adoptan una actitud confiada hacia los planes de éxito, y una actitud cautelosa hacia los planes que fallan.

En cuanto al consumo de recursos, se ha demostrado una mejoría en el tiempo de ejecución, el consumo de memoria RAM, y como era de esperarse en el espacio en Disco Duro. A pesar de que el consumo de memoria RAM es muy parecido entre la versión prelimiar de JILDT y la nueva versión de JILDT en un nivel de programación Java. El desempeño de ésta última puede mejorar conforme se implementen ejemplos mas complejos.

7.1. Trabajos futuros

Los trabajos futuros de este proyecto forman parte de la fase doctoral del mismo. El principal de éstos es extender la librería JILDT a un nivel de aprendizaje con base en su nivel de conocimiento social en un entorno multiagente, como se describe en la página 41. Esto, nos lleva a un aprendizaje distribuido, colectivo y social. Los protocolos de aprendizaje están previamente formalizados en (Guerra-Hernández et al., 2004a).

Entre los trabajos futuros a corto plazo se busca mejorar el mecanismo de extensión de planes, a través del uso de meta-eventos. Los meta-eventos permiten ejecutar acciones dependiendo el cambio de estado de una meta. Estos estados son: *created*, cuando una meta ha sido creada; *finished*, cuando se ha logrado terminar la ejecución de una meta; *failed*, cuando un fallo ha ocurrido en la ejecución de un plan; *suspended*, la meta está en suspenso (por ejemplo, con las acciones `.suspend` o `.wait`); y *resumed*, cuando se ha reanudado la ejecución de un plan.

Otra mejora de implementación incluye la selección de literales relacionadas con un plan, es decir, seleccionar únicamente las directivas rmode que tengan cierta relevancia con la intención que se esta intentando llevar a cabo, por ejemplo, en el mundo de los bloques, una creencia que especifique la fecha sería irrelevante para sustentar el aprendizaje.

Aún cuando el algoritmo TILDE implementado en JILDT tiene una buena representación sobre datos discretos, nada se ha hecho aún para manejar datos continuos, por lo que otro trabajo futuro consiste en implementar un mecanismo de discretización en primer orden, muy parecido al que usa C4.5.

En el capítulo 3 se habló de aprendizaje mediante Arboles de Decisión, los cuales parten de un conjunto completo de datos, es decir, un conjunto estático. Sin embargo, cuando se requiere que un sistema lleve a cabo tareas que necesitan aprender de forma serial o dinámica, por ejemplo, cuando los ejemplos de entrenamiento se presentan de manera secuencial, como un flujo, o están distribuidos en varios depósitos y no como un conjunto de tamaño fijo, se necesita revisar la hipótesis aprendida en presencia de nuevos ejemplos, en lugar de rehacerla cada vez que se tienen datos nuevos. Para ello, se pretende extender TILDE hacia un algoritmo incremental de aprendizaje (Utgoff, 1988).

Referencias

Austin, J. (1975). *How to Do Things with Words*. Harvard, MA., USA: Harvard University Press, second edition.

Bellifemine, F., Caire, G., & Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. England: John Wiley & Sons, Ltd.

Blockeel, H. (1998). *Top-down induction of first order logical decision trees*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

Blockeel, H. & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1–2), 285–297.

Blockeel, H., Raedt, L. D., Jacobs, N., & Demoen, B. (1999). Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1), 59–93.

Bordini, R. H., Bazzan, A. L. C., de O. Jannone, R., Basso, D. M., Vicari, R. M., & Lesser, V. R. (2002). Agentspeak(xl): efficient intention selection in BDI agents via decision-theoretic task scheduling. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems* (pp. 1294–1302). New York, NY, USA: ACM.

Bordini, R. H., Dastani, M., Dix, J., & Seghrouchni, A. E. F. (2005). *Multi-Agent Programming: Languages, Platforms and Applications*. Springer Science-Business Media Inc.

Bordini, R. H., Fisher, M., Pardavila, C., Visser, W., & Wooldridge, M. (2003a). Model checking multi-agent programs with casp. In *CAV* (pp. 110–113).

Bordini, R. H., Fisher, M., Pardavila, C., & Wooldridge, M. (2003b). Model checking AgentSpeak. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (pp. 409–416). New York, NY, USA: ACM Press.

Bordini, R. H. & Hübner, J. F. (2006). Bdi agent programming in agentspeak using jason. In F. Toni & P. Torroni (Eds.), *Proceedings of the Sixth International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VI), London, UK, 27-29 June, 2005, Revised Selected and Invited Papers*, volume 3900 of *Lecture Notes in Computer Science* (pp. 143–164). Berlin: Springer-Verlag.

Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). *Programming Multi-Agent Systems in Agent-Speak using Jason*. John Wiley & Sons Ltd.

Bordini, R. H. & Moreira, Á. F. (2004). Proving BDI properties of agent-oriented programming languages. *Annals of Mathematics and Artificial Intelligence*, 42, 197–226.

Bordini, R. H., Okuyama, F. Y., de Oliveira, D., Drehmer, G., & Krafta, R. C. (2004). The mas-soc approach to multi-agent based simulation. In G. Lindermann & et al. (Eds.), *RASTA 2002*, volume 2934 of *Lecture Notes in Artificial Intelligence* (pp. 70–91). Berlin Heidelberg: Springer-Verlag.

Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. Cambridge, MA., USA, and London, England: Harvard University Press.

Bratman, M. E., Pollak, M. E., & Israel, D. J. (1988). Plans and resource-bounded practical reasoning. *Computer Intelligence*, 4(4), 349–355.

Brentano, F. (1973). *Psychology from an Empirical Standpoint*. London: Routledge, second edition.

Busetta, P., Ronnquist, R., Hodgson, A., & Lucas, A. (1999). Jack intelligent agents - components for intelligent agents in java.

Covrigaru, A. & Lindsay, R. (1991). Deterministic autonomous systems. *AI Magazine*, Fall, 110–117.

Dennett, D. (1987). *The Intentional Stance*. Cambridge, MA., USA: MIT Press.

Dennett, D. C. (1971). Intentional systems. *The Journal of Philosophy*, 68(4), 87–106.

d'Inverno, M., Kinny, D., Luck, M., & Wooldridge, M. (1998). A formal specification of dmars. In M. Singh, A. Rao, & M. Wooldridge (Eds.), *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*, volume 1365 of *Lecture Notes in Artificial Intelligence* (pp. 155–176). Berlin-Heidelberg, Germany: Springer Verlag.

Ferber, J. (1995). *Les Systèmes Multi-Agents: vers une intelligence collective*. Paris, France: InterEditions.

Ferrater Mora, J. (2001). *Diccionario de filosofía*. España: Ariel.

Finin et al., T. (1992). *An overview of KQML: A Knowledge Query and Manipulation Language*. Technical report, University of Maryland, CS Department.

Foner, L. (1993). *What's an agent, anyway? A sociological case study*. Technical Report Agents Memo 93-01, MIT Media Lab, Cambridge, MA., USA.

Franklin, S. & Graesser, A. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. P. Muller, M. Wooldridge, & N. R. Jennings (Eds.), *Intelligent Agents III*, number 1193 in Lecture Notes in Artificial Intelligence (pp. 21–36). Berlin, Germany: Springer-Verlag.

Georgeff, M. & Ingrad, F. (1989). Decision-making in an embedded reasoning system. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89* (pp. 972–978). Detroit, MI., USA.

González-Alarcón, C. A. (2010). Jason induction of logical decision trees (jildt):una librería de aprendizaje y su aplicación en compromiso. Master's thesis, Maestría en Inteligencia Artificial. Universidad Veracruzana.

Guerra-Hernández, A. (2010). *Notas del curso Sistemas Multiagentes*. Technical report, Maestria en Inteligencia Artificial. Universidad Veracruzana.

Guerra-Hernández, A., Castro-Manzano, J. M., & El-Fallah-Seghrouchni, A. (2009). CTL AgentSpeak(L): a Specification Language for Agent Programs. *Journal of Algorithms*, (64), 31–40.

Guerra-Hernández, A., El-Fallah-Seghrouchni, A., & Soldano, H. (2004a). Distributed learning in BDI Multi-agent Systems. In R. Baeza-Yates, M. J.L., & E. Chávez (Eds.), *Fifth Mexican International Conference on Computer Science* (pp. 225–232). USA: Sociedad Mexicana de Ciencias de la Computación (SMCC) IEEE Computer Society.

Guerra-Hernández, A., El-Fallah-Seghrouchni, A., & Soldano, H. (2004b). Learning in BDI Multi-agent Systems. In J. Dix & J. Leite (Eds.), *Computational Logic in Multi-Agent Systems: 4th International Workshop, CLIMA IV, Fort Lauderdale, FL, USA, January 6–7, 2004, Revised and Selected Papers*, volume 3259 of *Lecture Notes in Computer Science* (pp. 218–233). Berlin Heidelberg: Springer-Verlag.

Guerra-Hernández, A., González-Alarcón, C., & El FallahSeghrouchni, A. (2010a). Jason induction of logical decision trees: A learning library and its application to commitment. In G. Sidorov, A. Hernández Aguirre, & C. Reyes García (Eds.), *Advances in Artificial Intelligence*, volume 6437 of *Lecture Notes in Computer Science* (pp. 374–385). Springer Berlin / Heidelberg.

Guerra-Hernández, A., González-Alarcón, C., & El FallahSeghrouchni, A. (2010b). Jason induction of logical decision trees (jildt): A learning library and its application to commitment. EUMAS 2010.

Guerra-Hernández, A., Mondragón-Becerra, R., & Cruz-Ramírez, N. (2008a). Explorations of the BDI multi-agent support for the knowledge discovery in databases process. *Research in Computing Science*, 39, 221–238.

Guerra-Hernández, A., Ortiz-Hernández, G., & Luna-Ramírez, W. A. (2008b). Jason smiles: incremental BDI MAS learning. In *MICAI 2007: Sixth Mexican International Conference on Artificial Intelligence, Special Session* (pp. 61–70). Los Alamitos: IEEE Computer Society CPS.

Huber, M. J. (1999). JAM: a BDI-theoretic mobile agent architecture. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents* (pp. 236–243). New York, NY, USA: ACM Press.

Hübner, J. F., Bordini, R. H., & Wooldridge, M. (2006). Programming declarative goals using plan patterns. In *proceedings DALT 2006* (pp. 123–140).

Kinny, D. N. (1991). Commitment and effectiveness of situated agents. In *In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91* (pp. 82–88).

Lee et al., J. (1994). UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS)* (pp. 842–849). Houston, Texas.

Lyons, W. (1995). *Approaches to intentionality*. Oxford, New York, USA: Oxford University Press Inc.

McCarthy, J. (1979). *Ascribing Mental Qualities to Machines*. Technical report, Computer Science Department, Stanford University, Stanford, CA., USA.

Mitchell, T. (1997). *Machine Learning*. Computer Science Series. Singapore: McGraw-Hill International Editions.

Moreira, Á. F. & Bordini, R. (2003). An operational semantics for a bdi agent-oriented programming language. In A. Omicini, L. Sterling, & P. Torroni (Eds.), *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003, Melbourne, Australia, July 15, 2003, Revised Selected and Invited Papers.*, volume 2990 of *Lecture Notes in Computer Science* (pp. 135–154). Berlin-Heidelberg, Germany: Springer Verlag.

Moreira, Á. F., Vieira, R., & Bordini, R. H. (2003). Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *DALT* (pp. 135–154).

Moro Simpson, T. (1964). *Semántica y Filosofía: Problemas y Discusiones*. Buenos Aires, Argentina: Eudeba.

Muggleton, S. & de Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19, 629–679.

Muller-Freienfels, W. (1999). Agency. In *Encyclopedia Britannica*. Encyclopedia Britannica, Inc. Internet version.

Nenhuys-Chen, S.-H. & de Wolf, R. (1997). *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Berlin Heidelberg: Springer-Verlag.

Nowaczyk, S. & Malec, J. (2007). Inductive logic programming algorithm for estimating quality of partial plans. In A. Gelbukh & Á. Kuri Morales (Eds.), *MICAI 2007: Advances in Artificial Intelligence*, volume 4827 of *Lecture Notes in Computer Science* (pp. 359–369). Springer Berlin / Heidelberg.

Ortiz-Hernández, G. (2007). Aprendizaje incremental en sistemas multi-agente bdi. Master's thesis, Maestría en Inteligencia Artificial. Universidad Veracruzana.

Perrault, C. R. & Allen, J. F. (1980). A plan-based analysis of indirect speech acts. *American Journal of Computational Linguistics*, 6(3-4), 167–182.

Plotkin, G. D. (1981). *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, University of Aarhus.

Quinlan, J. (1993). *C4.5: Programs for Machine Learning*. San Mateo, CA., USA: Morgan Kaufmann.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1, 81–106.

Rao, A. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe (Ed.), *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World* Eindhoven, The Netherlands.

Rao, A. & Georgeff, M. (1991). *Modelling Rational Agents within a BDI-Architecture*. Technical Report 14, Carlton, Victoria.

Russell, S. & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. USA: Prentice Hall, segunda edition.

Russell, S. & Subramanian, D. (1995). Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 2, 575–609.

Searle, J. R. (1962). Meaning and speech acts. *The Philosophical Review*, 71(4), 423–432.

Searle, J. R. (1979). What is an intentional state? *Mind, New Series*, 88(349), 74–92.

Searle, J. R. (1983). *Intentionality: An Essay in the Philosophy of Mind*. Cambridge University Press.

Sen, S. & Weiß, G. (1999). *Multiagent Systems, a modern approach to Distributed Artificial Intelligence*, chapter Learning in Multiagent Systems. MIT Press: Cambridge, MA., USA.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27, 379–423, 623–656.

Shoham, Y. (1990). *Agent-Oriented Programming*. Technical Report STAN–CS–1335–90, Computer Science Department, Stanford University, Stanford, CA., USA.

Singh, D., Sardina, S., & Padgham, L. (2010). Extending bdi plan selection to incorporate learning from experience. *Robotics and Autonomous Systems*, 58(9), 1067 – 1075. Hybrid Control for Autonomous Systems.

Singh, D., Sardiña, S., Padgham, L., & James, G. (2011). Integrating learning into a bdi agent for environments with changing dynamics. In *IJCAI* (pp. 2525–2530).

Singh, M. (1995). *Multiagent Systems: A theoretical framework for intentions, know-how, and communication*. Number 799 in Lecture Notes in Computer Sciences. Berlin Heidelberg: Springer-Verlag.

Subagdja, B., Sonenberg, L., & Rahwan, I. (2009). Intentional learning agent architecture. *Autonomous Agents and Multi-Agent Systems*, 18, 417–470.

Tan, P.-N., Steinbach, M., & Kumar, V. (2006). *Introduction to Data Mining*. Addison Wesley.

Utgoff, P. E. (1988). Id5: An incremental id3. In *ML* (pp. 107–120).

Vieira, R., Moreira, Á., Wooldridge, M., & Bordini, R. H. (2007). On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research*, 29, 221–267.

Witten, I. H. & Frank, E. (2000). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. San Francisco: Morgan Kaufmann.

Wooldridge, M. (2000). *Reasoning about Rational Agents*. Cambridge, MA., USA: MIT Press.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. West Sussex, England: John Wiley & Sons, LTD.

Wooldridge, M. & Jennings, N. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2), 115–152.

Apéndice A
Semantica Operacional

A.1. *AgentSpeak(L)*

Definición 15 (Agente) *Un agente es una tupla $\langle E, B, P, I, A, \mathcal{S}_{\mathcal{E}}, \mathcal{S}_{\mathcal{O}}, \mathcal{S}_{\mathcal{I}} \rangle$, donde E es un conjunto de eventos, B es un conjunto de creencias base, P es un conjunto de planes, I es un conjunto de intenciones y A es un conjunto de acciones. La función de selección $\mathcal{S}_{\mathcal{E}}$ selecciona un evento entre los miembros de E. La función de selección $\mathcal{S}_{\mathcal{O}}$ selecciona un plan aplicable u opción de entre los miembros de P. La función de selección $\mathcal{S}_{\mathcal{I}}$ selecciona una intención de entre los miembros de I.*

Definición 16 (Intenciones) *El conjunto I se compone de las intenciones del agente. Una intención es una pila de planes cerrados parcialmente (planes que pueden incluir algunas variables libres y otras con valores asignados). Una intención se denota por $[p_1 \ddagger \ldots \ddagger p_z]$, donde p_1 representa el fondo de la pila y p_z el tope de la misma. Por conveniencia, la intención $[+!true : true \leftarrow true]$ será denotada por $T = true$.*

Definición 17 (Eventos) *El conjunto E se compone de eventos. Cada evento es una tupla $\langle e, i \rangle$ donde e es un evento disparador e i es una intención. Si la intención $i = true$, al evento se le identifica como un evento externo; en cualquier otro caso es un evento interno.*

Ahora se pueden definir formalmente los conceptos de plan relevante y aplicable. Para ello es necesario recordar el concepto de unificador más general (*MGU*):

Definición 18 (Unificador) *Sean α y β términos. Una substitución θ tal que α y β sean idénticos ($\alpha\Theta = \beta\Theta$) es llamada unificador de α y β.*

Definición 19 (Generalidad entre substituciones) *Una substitución θ se dice más general que una substitución σ, si y sólo si existe una substitución γ tal que $\sigma = \theta\gamma$.*

Definición 20 (MGU) *Un unificador θ se dice el unificador más general (MGU) de dos términos, si y sólo si θ es más general que cualquier otro unificador entre esos términos.*

Definición 21 (Plan relevante) *Sea $\mathcal{S}_{\mathcal{E}}(E) = \varepsilon = \langle d, i\rangle$ y sea el plan $p = e : b_1 \wedge \cdots \wedge b_n \leftarrow h_1; \ldots; h_m$. El plan p es relevante con respeto al evento ε si y sólo si existe un unificador más general (MGU) σ tal que $d\sigma = e\sigma$. A σ se le llama el unificador relevante para ε.*

Definición 22 (Plan aplicable) *Un plan p denotado por $e : b_1 \wedge \cdots \wedge b_n \leftarrow h_1; \ldots; h_m$ es un plan aplicable con respecto a un evento ε si y sólo si existe un unificador relevante σ para ε y existe una substitución θ tal que $\forall(b_1 \wedge \cdots \wedge b_n)\sigma\theta$ es consecuencia lógica de B (creencias del agente). La composición $\sigma\theta$ se conoce como el unificador aplicable para ε; y θ se conoce como la substitución de respuesta correcta.*

Definición 23 (Intención evento externo) *Sea $\mathcal{S}_{\mathcal{O}}(O_{\varepsilon}) = p = e : b_1 \wedge \cdots \wedge b_n \leftarrow h_1; \ldots; h_m$ donde O_{ε} es el conjunto de todos los planes aplicables u opciones para el evento $\langle d, i\rangle$. El plan p es intentado con respecto al evento ε, donde la intención $i = T$, si y sólo si existe un unificador aplicable σ tal que $[+!true : true \leftarrow true \ddagger (e : b_1 \wedge \cdots \wedge b_n \leftarrow h_1; \ldots; h_m)\sigma] \in I$.*

Definición 24 (Intención evento interno) *Sea $\mathcal{S}_{\mathcal{O}}(O_{\varepsilon}) = p = +!g(s) : b_1 \wedge \cdots \wedge b_j \leftarrow h_1; \ldots; h_k$ donde O_{ε} es el conjunto de todos los planes aplicables u opciones para el evento $\varepsilon = \langle d, [p_1 \ddagger \ldots \ddagger f : c_1 \wedge \cdots \wedge c_m \leftarrow !g(t); k_2; \ldots; k_n]\rangle$. El plan p es intentado con respecto al evento ε, si y sólo si existe un unificador aplicable σ tal que $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \cdots \wedge c_m \leftarrow !g(t); k_2; \ldots; k_n \ddagger (+!g(s) : b_1 \wedge \cdots \wedge b_j)\sigma \leftarrow (h_1; \ldots; h_k)\sigma; (k_2; \ldots k_n)\sigma] \in I$.*

Las siguientes funciones determinan como afecta a una intención *i* la ejecución de metas, acciones y submetas logradas.

Definición 25 (Ejecución achieve) *Sea $\mathcal{S}_{\mathcal{I}}(I) = i = [p_1 \ddagger \ldots \ddagger f : c_1 \wedge \cdots \wedge c_m \leftarrow !g(t); h_2; \ldots; h_n]$. Se dice que la intencin i ha sido ejecutada, si y sólo si $\langle +!g(t), i\rangle \in E$.*

Definición 26 (Ejecución test) *Sea $\mathcal{S}_{\mathcal{I}}(I) = i = [p_1 \ddagger \ldots \ddagger f : c_1 \wedge \cdots \wedge c_m \leftarrow ?g(t); h_2; \ldots; h_n]$. Se dice que la intención i ha sido ejecutada, si y sólo si existe una substitución θ tal que $\forall g(t)\theta$ es una consecuencia lógica de B e i es remplazada por $[p_1 \ddagger \ldots \ddagger (f : c_1; \ldots; c_m)\sigma \leftarrow (h_2; \ldots h_n)\sigma]$.*

Definición 27 (Ejecución acción) *Sea $\mathcal{S}_{\mathcal{I}}(I) = i = [p_1 \ddagger \ldots \ddagger f : c_1 \wedge \cdots \wedge c_m \leftarrow a(t); h_2; \ldots; h_n]$. Se dice que la intención i ha sido ejecutada, si y sólo si $a(t) \in A$ e i es remplazada por $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \cdots \wedge c_m \leftarrow h_2; \ldots; h_n]$*

Definición 28 (Ejecución submeta) *Sea $\mathcal{S}_{\mathcal{I}}(I) = i = [p_1 \ddagger \ldots \ddagger p_{z-1} \ddagger !g(t) : c_1 \wedge \cdots \wedge c_m \leftarrow true]$, donde $p_{z-1} = e : b_1 \wedge \cdots \wedge b_x \leftarrow !g(s); h_2; \ldots; h_y$. Se dice que la intención i ha sido ejecutada, si y sólo si existe una substitución θ tal que $g(t)\theta = g(s)\theta$ e i es remplazada por $[p_1 \ddagger \ldots \ddagger p_{z-1} \ddagger (e : b_1 \wedge \cdots \wedge b_x)\theta \leftarrow h_2; \ldots; h_y)\theta]$.*

Ahora es posible definir un intérprete para $AgentSpeak(L)$ (Algoritmo 6). Las funciones $top, push, first$ y *rest* tienen semántica evidente.

Algoritmo 6 El algoritmo del intérprete $AgentSpeak(L)$

procedure AGENTSPEAK(L)()
  while $E \neq \emptyset$ **do**
    $\varepsilon \leftarrow \langle d, i \rangle \leftarrow \mathcal{S}_{\mathcal{E}}(E)$;
    $E \leftarrow E \backslash \varepsilon$;
    $O_\varepsilon \leftarrow \{p\theta | \theta$ es un unificador aplicable para ε y $p\}$;
    if $externo(\varepsilon)$ **then**
      $I \leftarrow I \cup [\mathcal{S}_{\mathcal{O}}(O_\varepsilon)]$;
    else
      $push(\mathcal{S}_{\mathcal{O}}(O_\varepsilon)\sigma, i)$ donde σ es un unificador aplicable para ε;
    end if
    if $first(body(top(\mathcal{S}_{\mathcal{I}}(I)))) = true$ **then**
      $x \leftarrow pop(\mathcal{S}_{\mathcal{I}}(I))$;
      $push(head(top(\mathcal{S}_{\mathcal{I}}(I)))\theta \leftarrow rest(body(top(\mathcal{S}_{\mathcal{I}}(I))))\theta, \mathcal{S}_{\mathcal{I}}(I))$
      donde θ es un mgu t.q. $x\theta = head(top(\mathcal{S}_{\mathcal{I}}(I)))\theta$;
    else if $first(body(top(\mathcal{S}_{\mathcal{I}}(I)))) = !g(t)$ **then**
      $E = E \cup \langle +!g(t), \mathcal{S}_{\mathcal{I}}(I) \rangle$
    else if $first(body(top(\mathcal{S}_{\mathcal{I}}(I)))) = ?g(t)$ **then**
      $pop(\mathcal{S}_{\mathcal{I}}(I))$;
      $push(head(top(\mathcal{S}_{\mathcal{I}}(I))))\theta \leftarrow rest(body(top(\mathcal{S}_{\mathcal{I}}(I))))\theta, \mathcal{S}_{\mathcal{I}}(I))$
      donde θ es la substitucin de respuesta correcta.
    else if $first(body(top(\mathcal{S}_{\mathcal{I}}(I)))) = a(t)$ **then**
      $pop(\mathcal{S}_{\mathcal{I}}(I))$;
      $push(head(top(\mathcal{S}_{\mathcal{I}}(I)))) \leftarrow rest(body(top(\mathcal{S}_{\mathcal{I}}(I)))), \mathcal{S}_{\mathcal{I}}(I))$;
      $A = A \cup \{a(t)\}$;
    end if
  end while
end procedure

A.1.1. Teoría de prueba

Para formular la teoría de prueba de $AgentSpeak(L)$, recurrimos a un sistema de transición, como los propuestos por Plotkin (1981).

Definición 29 (Sistema transición BDI) *Un sistema de transición BDI es un par* $\langle \Gamma, \vdash \rangle$ *que consiste en:*

- *Un conjunto* Γ *de configuraciones; y*
- *Una relación binaria de transición* $\vdash \subseteq \Gamma \times \Gamma$.

Definición 30 (Configuración BDI) *Una tupla $\langle E_i, B_i, I_i, A_i, i\rangle$, donde $E_i \subseteq E$, $B_i \subseteq B$, $I_i \subseteq I$, $A_i \subseteq A$, e i es la etiqueta de la transición; es una configuración BDI.*

El conjunto de planes P no forma parte de las configuraciones, pues se asume que permanece constante[1]. Tampoco se lleva un registro explícito de las metas, pues se asume que éstas aparecen como intenciones cuando son adoptadas por los agentes. Ahora es posible escribir reglas de transición que lleven al agente de una configuración BDI a otra. La primera regla define la transición al intentar un plan al nivel más alto (un fin, en términos de razonamiento medios-fines). La regla especifica como el agente modifica sus intenciones en respuesta a un evento externo:

$$\textbf{(IntendEnd)}\frac{\langle\{\ldots,\langle+!g(t),T\rangle,\ldots\},B_iI_i,A_i,i\rangle}{\langle\{\ldots\},B_i,I_i\cup\{[p\sigma\theta]\},A_i,i+1\rangle}$$

donde: $p = +!g(s) : b_1 \wedge \cdots \wedge b_m \leftarrow h_1;\ldots;h_n \in P$, $\mathcal{S}_\mathcal{E}(E) = \langle+!g(t),T\rangle$, $g(t)\sigma = g(s)\sigma$ y $\forall(b_1 \wedge \cdots \wedge b_m)\theta$ es consecuencia lógica de B_i.

La regla para intentar un medio es similar a la regla para intentar un fin, sólo que el plan aplicable es colocado sobre la pila cuyo tope es la intención dada como segundo argumento del evento elegido:

$$\textbf{(IntendMeans)}\frac{\langle\{\ldots,\langle+!g(t),j\rangle,\ldots\},B_i\{\ldots,[p_1 \ddagger \ldots \ddagger p_z],\ldots\},A_i,i\rangle}{\langle\{\ldots\},B_i,\{\ldots,[p_1 \ddagger \ldots \ddagger p_z \ddagger p\sigma\theta],\ldots\},A_i,i+1\rangle}$$

donde $p_z = f : c_1 \wedge \cdots \wedge c_y \leftarrow !g(t);h_2;\ldots;h_n$, $p = +!g(s) : b_1 \wedge \cdots \wedge b_m \leftarrow k_1;\ldots;k : x$, $\mathcal{S}_\mathcal{E}(E) = \langle+!g(t),j\rangle$ es $[p_1 \ddagger \ldots \ddagger p_n]$, $g(t)\sigma = g(s)\sigma$ y $\forall(c_1 \wedge \cdots \wedge c_y)\theta$ es una consecuencia lógica de B_i.

A. Rao define una regla más para la adopción de metas y propone que el lector puede elaborar reglas parecidas para el resto de las transiciones en el sistema. De esta forma, es posible definir derivaciones y refutaciones, usando las reglas de prueba.

Definición 31 (Derivación) *Una derivación BDI es una secuencia finita o infinita de configuraciones* $\gamma_0,\ldots,\gamma_i,\ldots$.

La noción de refutación en $AgentSpeak(L)$ se da con respecto a una intención particular. La refutación de una intención inicia cuando ésta es adoptada y termina cuando su pila queda vacía. Por lo tanto, usando las reglas anteriores es posible verificar la seguridad y viabilidad del sistema. Además hay una correspondencia de uno a uno entre las reglas de prueba y la semántica operacional del sistema. Dentro de las extensiones posibles se encuentran operadores más interesantes para el cuerpo de los planes (aquellos de la lógica dinámica) y post condiciones diferenciadas para los casos de éxito y de fracaso, como se especifica en dMARS (d'Inverno et al., 1998).

[1] Este no es el caso si el agente puede modificar sus planes originales, por ejemplo, mediante aprendizaje.

A.2. Jason

La semántica operacional de *Jason* está definida en términos de un sistema de transición Γ entre configuraciones. Una **configuración** $\langle ag, C, M, T, s\rangle$ está compuesta por:

- El **programa del agente** ag es una tupla $\langle bs, ps\rangle$ formada por las creencias bs y los planes ps del agente.
- Una **circunstancia** del agente C es una tupla $\langle I, E, A\rangle$ donde I es el conjunto de **intenciones** $\{i_1, i_2, \ldots, i_n\}$ tal que cada $i \in I$ es una pila de planes parcialmente instanciados $p \in ps$. El operador $\ddagger$ es usado para separar los elementos de las pilas. $[\alpha \ddagger \beta]$ es una pila de dos elementos, α en su tope; E es un conjunto de **eventos** $\{\langle te_1, i_1\rangle, \langle te_2, i_2\rangle, \ldots, \langle te_n, i_n\rangle\}$, tal que cada te es un $triggerEvent$ y cada i es una intención no vacía (evento interno) o vacía $\top$ (evento externo); y A es el conjunto de **acciones** a ser ejecutadas por el agente en el ambiente.
- M es una tupla $\langle In, Out, SI\rangle$ donde In es el **buzón** del agente, Out es una lista de mensajes a ser enviados, y SI es un registro de intenciones suspendidas (aquellas que esperan un mensaje de respuesta para reanudarse).
- T es una tupla $\langle R, Ap, \iota, \varepsilon, \rho\rangle$ donde R es el conjunto de **planes relevantes** dado cierto evento; Ap es el conjunto de **planes aplicables** (el subconjunto de planes $p \in R$ tal que $bs \models Ctxt(p)$, donde la función $Ctxt$ regresa el contexto de un plan o $true$ si tal contexto está vacío); ι, ε y ρ son, respectivamente, la intención, el evento, y el plan actualmente considerados en el razonamiento del agente.
- $s \in \{SelEv, RelPl, AppPl, SelAppl, SelInt, AddIM, ExecInt, ClrInt, ProcMsg\}$ indica el **estado actual** en el ciclo de razonamiento del agente.

Las transiciones se definen en términos de reglas semánticas con la forma:

$$(\textbf{rule id}) \quad \frac{cond}{C \rightarrow C'}$$

donde $C = \langle ag, C, M, T, s\rangle$ es una configuración que puede transformarse en una nueva configuración C', si $cond$ se cumple. Para efectos de describir las reglas semánticas adoptamos la siguiente notación:

- Para hacer referencia al componente E (eventos) de una circunstancia C, escribimos C_E. De manera similar accedemos a los demás componentes de una configuración.
- Para indicar que no hay ninguna intención siendo considerada en la ejecución del agente, se emplea $T_\iota = \emptyset$. De forma similar para T_ε, T_ρ y demás registros de una configuración.
- Se usa $i[p]$ para denotar que p es el plan en el tope de la intención i.
- Si asumimos que p es un plan de la forma $te : ct \leftarrow h$, entonces: $TrEv(p) = te$ y $Ctxt(p) = ct$.

A.2.1. Consecuencia lógica y definiciones auxiliares

Definición 32 (Planes relevantes) *El conjunto de planes relevantes con respecto a un evento disparador te está dado por:*

$$PlanesRel(ps,te) = \{p\theta \mid p \in ps \wedge \theta = mgu(te, TrEv(p))\}$$

Definición 33 (Planes aplicables) *Dadas las creencias de un agente (bs) y un conjunto Rel de planes relevantes, el conjunto de planes aplicables está dado por:*

$$PlanesApp(bs,Rel) = \{p\theta \mid p \in Rel \wedge \theta \quad t.q. \quad bs \models Ctxt(p)\theta\}$$

Definición 34 (Test) *Dadas las creencias de un agente (bs) y una fbf at, el conjunto de substituciones que validan la fbf contra las creencias está dado por:*

$$Test(bs,at) = \{\theta \mid bs \models at\theta\}$$

Es necesario definir la noción específica de **consecuencia lógica** que *Jason* emplea. Para ello asumiremos la existencia de un procedimiento que computa el unificador más general (*mgu*) entre dos literales y con él definiremos la relación de consecuencia lógica ($\models$). Esta relación se utiliza para computar planes relevantes, aplicables y metas de verificación (?). Aunque todo este aparato es similar al usado en la programación lógica, necesitamos considerar que los átomos en Jason pueden estar anotados, por ejemplo, la creencia `factorial(0,1)[self]`, indica que un agente cree por sí mismo que el factorial de 0 es 1.

Definición 35 (Consecuencia Lógica) *Decimos que una fórmula atómica at_1 con anotaciones $s_{1_1}, \ldots s_{1_n}$ es consecuencia lógica de un conjunto de fórmulas atómicas de base bs, denotado por $bs \models at_1[s_{1_1}, \ldots s_{1_n}]$ si y sólo si existe una fórmula atómica $at_2[s_{2_1}, \ldots s_{2_m}] \in bs$ tal que (i) $at_1\theta = at_2$ para algún unificador más general θ y (ii) $\{s_{1_1}, \ldots s_{1_n}\} \subseteq \{s_{2_1}, \ldots s_{2_m}\}$.*

Finalmente, las funciones de selección de $AgentSpeak(L)$ aquí son denotadas por S_E, S_{Ap} y S_I.

A.2.2. Reglas de transición

Las reglas de transición que definen la semántica operacional de *Jason* inducen el ciclo de razonamiento mostrado en la figura A.1. Las etiquetas de los nodos corresponden al estado en el ciclo de razonamiento. Los arcos están etiquetados con los identificadores de las reglas de transición que definiremos a continuación. El estado inicial *ProcMsg* se encarga de actualizar el estado del agente a partir de las percepciones y comunicaciones recibidas por el agente.

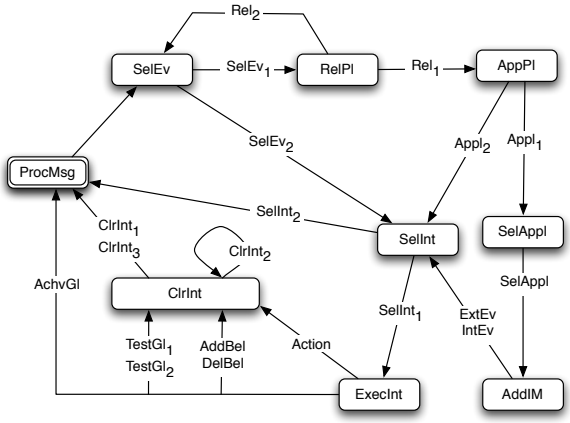


Figura A.1 El ciclo de razonamiento de Jason. Adaptado de Bordini et al. (2007), p. 206, en Guerra-Hernández et al. (2009).

$$(\mathbf{SelEv_1}) \quad \frac{\mathcal{S}_{\mathcal{E}}(C_E) = \langle te, i \rangle}{\langle ag, C, M, T, SelEv \rangle \rightarrow \langle ag, C', M, T', RelPl \rangle}$$

s.t. $C'_E = C_E \backslash \{\langle te, i \rangle\}, T'_\varepsilon = \langle te, i \rangle$

La regla Rel_1 asigna a R el conjunto de planes relevantes. Si no existe ningún plan relevante, el evento es descartado de ε por la regla Rel_2.

$$(\mathbf{Rel_1}) \quad \frac{T_\varepsilon = \langle te, i \rangle, RelPlans(ag_{ps}, te) \neq \{\}}{\langle ag, C, M, T, RelPl \rangle \rightarrow \langle ag, C, M, T', AppPl \rangle}$$

s.t. $T'_R = RelPlans(ag_{ps}, te)$

$$(\mathbf{Rel_2}) \quad \frac{RelPlans(ps, te) = \{\}}{\langle ag, C, M, T, RelPl \rangle \rightarrow \langle ag, C, M, T, SelEv \rangle}$$

El caso de los planes aplicables es parecido:

$$(\mathbf{Appl_1}) \quad \frac{ApplPlans(ag_{bs}, T_R) \neq \{\}}{\langle ag, C, M, T, ApplPl \rangle \rightarrow \langle ag, C, M, T', SelAppl \rangle}$$

s.t. $T'_{Ap} = AppPlans(ag_{bs}, T_R)$

La siguiente regla asume la existencia de una función de selección S_{Ap}, la cual selecciona un plan a partir del conjunto Ap de planes aplicables.

$$(\textbf{SelAppl}) \quad \frac{\mathcal{S}_{\mathcal{O}}(T_{Ap}) = (p,\theta)}{\langle ag,C,M,T,SelAppl\rangle \rightarrow \langle ag,C,M,T',AddIM\rangle}$$

s.t. $T'_p = (p,\theta)$

Recordemos que en Jason se distinguen dos tipos de eventos, internos y externos:

$$(\textbf{ExtEv}) \quad \frac{T_\varepsilon = \langle te,\top\rangle, T_p = (p,\theta)}{\langle ag,C,M,T,AddIM\rangle \rightarrow \langle ag,C',M,T,SelInt\rangle}$$

s.t. $C'_I = C_I \cup \{[p\theta]\}$

$$(\textbf{SelInt}_\textbf{1}) \quad \frac{C_I \neq \{\}, \mathcal{S}_{\mathcal{I}}(C_I) = i}{\langle ag,C,M,T,SelInt\rangle \rightarrow \langle ag,C,M,T',ExecInt\rangle}$$

s.t. $T'_\iota = i$

La regla para seleccionar una intención a ser ejecutada es como sigue:

$$(\textbf{SelInt}_\textbf{2}) \quad \frac{C_I = \{\ \}}{\langle ag,C,M,T,SelInt\rangle \rightarrow \langle ag,C,M,T,ProcMsg\rangle}$$

El grupo de reglas que se describen a continuación, expresan el efecto de la ejecución de los planes. El plan siendo ejecutado es siempre aquel que se encuentra en el tope de la intención que ha sido previamente seleccionada. Todas las reglas en este grupo terminan descartando i, por lo que otra intención puede ser seleccionada eventualmente. Las reglas se ejecutan dependiendo del componente del cuerpo del plan que se ha seleccionado:

$$\textbf{Action} \quad \frac{}{C, creencias \rightarrow C', creencias} \quad C_\iota = i[head \leftarrow a; h]$$

$$donde: \quad C'_\iota = \_, C'_A = C_A \cup \{a\}$$

$$C'_I = (C_I \backslash \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

La siguiente regla registra una nueva meta de tipo *achieve*, que también podrá ser seleccionada por la regla $SelEv$:

$$(\textbf{AchvGl}) \quad \frac{T_\iota = i[head \leftarrow !at; h]}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M, T, ProcMsg\rangle}$$

s.t. $C'_E = C_E \cup \{\langle +!at, T_\iota\rangle\}, C'_I = C_I \setminus \{T_\iota\}$

Observe como la intención que generó el evento interno es removida del conjunto de intenciones C_I. Esto implementa la suspensión una intención. Sólo cuando el curso de acción definido ha sido terminado, se puede continuar con la ejecución de la intención que había sido suspendida, a partir de la siguiente fórmula del cuerpo de un plan dado[2].

Las metas de tipo test, se procesan mediante las siguientes dos reglas:

$$\textbf{Test}_\textbf{1} \quad \frac{Test(creencias, \phi) = \emptyset}{C, creencias \rightarrow C', creencias} \; C_\iota = i[head \leftarrow ?at; h]$$
$$donde: \quad C'_\iota = \emptyset$$
$$C'_I = C_I\{C_\iota\} \cup \{i[head \leftarrow h]\}$$

$$\textbf{Test}_\textbf{2} \quad \frac{Test(creencias, \phi) \neq \emptyset}{C, creencias \rightarrow C', creencias} \qquad C_\iota = i[head \leftarrow ?at; h]$$
$$donde: C'_\iota = \emptyset, C'_I = C_I\{C_\iota\} \cup \{i[(head \leftarrow h)\theta]\}$$
$$theta \in Test(creencias, \phi)$$

Al igual que en dMARS (d'Inverno et al., 1998), los agentes en Jason pueden agregar o eliminar creencias durante la ejecución de sus planes. Las siguientes reglas se encargan de ello:

[2] A partir de la versión 0.9.4 se implementó un nuevo operador para indicar sub-meta (!!). Éste es un nuevo tipo de fórmula, que no suspende las intenciones, si no que crea una nueva pila.

$$\textbf{AddBel}\ \frac{}{C, creencias \rightarrow C', creencias'}\ C_\iota = i[head \leftarrow +at; h]$$
$$donde: \quad C'_\iota = \emptyset, creencias \models at$$
$$C_E \cup \{(+at, C_\iota)\}$$
$$C'_I = C_I\{C_\iota\} \cup \{i[head \leftarrow h']\}$$
$$\textbf{DelBel}\ \frac{}{C, creencias \rightarrow C', creencias'}\ C_\iota = i[head \leftarrow -at; h]$$
$$donde: \quad C'_\iota = \emptyset, creencias \not\models at$$
$$C_E \cup \{(-at, C_\iota)\}$$
$$C'_I = C_I\{C_\iota\} \cup \{i[head \leftarrow h']\}$$

Para concluir con la semántica operacional de Jason se definen dos reglas más, las llamadas *clearing house rules*. $ClearInt_1$ simplemente remueve una intención del conjunto de intenciones de un agente cuando no hay más que hacer al respecto, es decir, ya no quedan más fórmulas (acciones o metas) que ejecutar en el cuerpo del plan.

$$(\textbf{ClrInt}_\textbf{1}) \quad \frac{T_\iota = [head \leftarrow \top]}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag, C', M, T, ProcMsg\rangle}$$

t.q. $C'_I = C_I \setminus \{T_\iota\}$

$$(\textbf{ClrInt}_\textbf{2}) \quad \frac{T_\iota = i[head \leftarrow \top]}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag, C', M, T, ClrInt\rangle}$$

t.q. $C'_I = (C_I \setminus \{T_\iota\}) \cup \{k[(head' \leftarrow h)\theta]\}$ si $i = k[head' \leftarrow g; h]$ y $g\theta = TrEv(head)$

$$(\textbf{ClrInt}_\textbf{3}) \quad \frac{T_\iota \neq [head \leftarrow \top] \wedge T_\iota \neq i[head \leftarrow \top]}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag, C, M, T, ProcMsg\rangle}$$

Apéndice B
Código de los agentes del mundo de los bloques

B.1. Agente *learner*

```
// Agent learner in project jildt_bw1.mas2j */

/* jildt settings  */
jildt_settings(excludeBels, [noExperiments]).
jildt_settings(inductionLevel, agentSpeak).

/* Initial beliefs and rules */
clear(X) :- not(on(_,X)).
clear(table).

/* Plans */
{begin learnablePlans}
@put
+!put(X,Y) : true <-
  move(X,Y);
  .print("Yeah, I did the task succesfully");
  .send(experimenter,tell,experiment(done)).

@put_failCase
-!put(X,Y) : true <-
  .send(experimenter,tell,experiment(done)).
{end}

@put_failCaseNonApplicable
-!put(X,Y)[error(Error)] : .member(Error,[no_applicable,no_relevant,no_option,wrong_arguments,unknown]) <-
  .print("Plan +!put produced an irrelevant failure.");
  .send(experimenter,tell,[non_applicable(put),experiment(done)]).
```

B.2. Agente *singleMinded*

```
// Agent singleMinded in project jildt_bwl.mas2j

/* jildt settings  */
jildt_settings(excludeBels, [noExperiments]).
jildt_settings(inductionLevel, agentSpeak).
jildt_settings(learningPlansSrc, "../sml/smLearnPlans.asl").

/* Initial beliefs and rules */
clear(X) :- not(on(_,X)).
clear(table).

/* Plans */
{begin learnablePlans}
@put
+!put(X,Y) : true <-
  move(X,Y);
  .print("Yeah, I did the task succesfully");
  .send(experimenter,tell,experiment(done)).

@put_failCase
-!put(X,Y) : true <-
  .send(experimenter,tell,experiment(done)).
{end}

@put_failCaseNonApplicable
-!put(X,Y)[error(Error)] : .member(Error,[no_applicable,no_relevant,no_option,wrong_arguments,unknown]) <-
  .print("Plan +!put produced an irrelevant failure.");
  .send(experimenter,tell,[non_applicable(put),experiment(done)]).

@dropPlan
+dropIntention(I) : true <-
  .print("Wow!! I'm sorry, I have to abandon my intention");
  .drop_intention(I);
  .send(experimenter,tell,[dropped_int(put(X,Y)),experiment(done)]).
```

B.3. Agente *default*

```
// Agent default in project jildt_bwl.mas2j

/* Initial beliefs and rules */
clear(X) :- not(on(_,X)).
clear(table).
```

```
6
7   /* Plans */
8   @put
9   +!put(X,Y) : true <-
10    move(X,Y);
11    .print("Yeah, I did the task succesfully");
12    .send(experimenter,tell,experiment(done)).
13
14  @put_fail
15  -!put(X,Y) : true <-
16    .send(experimenter,tell,experiment(done)).
```

B.4. Agente *experimenter*

```
1   // Agent experimenter in project jildt_bw1.mas2j
2
3   /* Initial beliefs and rules
4   noiseOcur(0.5).
5   noiseOptn(0.5).
6   noiseLatn(0.5).
7   agent2exp(singleM).
8   noExperiments(1).
9
10  /* Initial goals */
11  !run_all_experiments.
12
13  /* Plans */
14  +!run_all_experiments : noExperiments(101) <- true.
15
16  +!run_all_experiments : noExperiments(NE)[source(_)] <-
17    .print("=====================================================");
18    .print("Experiment Number: ",NE, "/100");
19    ?agent2exp(Ag);
20    !run_one_experiment(Ag);
21    .wait("+experiment(done)[source(Ag)]");
22    NewNE = NE + 1;
23    -+noExperiments(NewNE);
24    !result(Ag);
25    -experiment(done)[source(Ag)];
26    !run_all_experiments.
27
28  +!run_one_experiment(Ag) : .random(R) & noiseOcur(NP) & R < NP  <-
29    !getNoiseOption(z,B);
30    .print("Noise : on(",z,",",B,")");
31    !noiseMovement(z,B,Ag).
32
```

```
33  +!run_one_experiment(Ag) : true <-
34    .print("Experiment without noise");
35    .print("--------------------------");
36    .send(Ag,achieve,put(b,c)).
37
38  +!getNoiseOption(X,Y): .random(R) & noiseOptn(OP) & R < OP <- Y = c.
39  +!getNoiseOption(X,Y): true <- Y = b.
40
41  +!noiseMovement(A,B,Ag) : .random(R) & noiseLatn(LP) & R < LP <-
42    .print("Latency: After");
43    .print("--------------------------");
44    .send(Ag,achieve,put(b,c));
45    move(A,B).
46
47  +!noiseMovement(A,B,Ag) : true <-
48    .print("Latency: Before");
49    .print("--------------------------");
50    move(A,B);
51    .send(Ag,achieve,put(b,c)).
52
53  +!result(Ag) : on(b,c)  <-
54    .print("Normal has succeed.");
55    move(b,a).
56
57  +!result(Ag) : non_applicable(put) <-
58    -non_applicable(put)[source(Ag)];
59    .print("Normal has failed because it didn't take the intention");
60    move(z,table).
61
62  +!result(Ag) : dropped_int(put(X,Y))<-
63    -dropped_int(put(X,Y))[source(Ag)];
64    .print("Normal has failed because it abandoned the intention");
65    move(z,table).
66
67  +!result(Ag) : true <-
68    .print("Normal has failed");
69    move(z,table).
```