UNIVERSIDAD VERACRUZANA
FACULTAD DE FÍSICA E INTELIGENCIA ARTIFICIAL
DEPARTAMENTO DE INTELIGENCIA ARTIFICIAL

"Jason Induction of Logical Decision Trees (JILDT): Una librería de aprendizaje y su aplicación en compromiso".

TESIS

Que para obtener el grado de :

Maestro en Inteligencia Artificial

Presenta:

Carlos Alberto González Alarcón

Director:

Dr. Alejandro Guerra Hernández

Revisores:

Dr. Manuel Martinez Morales
Dr. Jesus Antonio González Bernal
Dr. Nicandro Cruz Ramirez

Xalapa, Veracruz. 10 de Diciembre de 2010

Agradecimientos

A Dios, por permitirme terminar un ciclo mas en mi vida, y mas aun, por permitirme proyectar nuevas metas.

A mis padres, por quienes soy quien soy, por todo el cariño y apoyo recibido desde pequeño hasta estas instancias, por la comprensión que recibí de su parte principalmente estos últimos dos años y medio. Con cariño.

A mi abuelita, por darme mas cariño del que un nieto puede esperar, por todos esos momentos en que me has apoyado y escuchado atentamente. Con cariño.

A mis hermanas, porque me alientan siempre a seguir adelante. Chiquis, tu coraje me motiva, porque a pesar de que muchas veces me dices cuan orgullosa estas, no sabes lo orgulloso que estoy yo de ti; Mariana, que sabes que por muchos años has sido con quien mas he convivido, y mas que mi hermana has sido mi confidente y mi apoyo; Fany, tu inocencia y alegría me dan alas de esperanza, pensando siempre en un mejor futuro para ti y tus hermanas. Las quiero.

Al Dr. Alejandro Guerra Hernandez, por darme la confianza para desarrollar este trabajo. Por el apoyo y la paciencia brindados. Gracias por fomentar en mi el interés a la investigación, pero sobretodo, gracias por la amistad y las oportunidades.

A esas personitas que me han ofrecido su amistad hasta ahora, quisiera listarlos, pero no me alcanzaría el espacio. Amigos de la primaria, secundaria, preparatoria, facultad, francés, inglés y a aquellos que he conocido de algún otro modo.

A la generación MIA2K8, con quienes compartí noches de desvelo, especialmente a Omar, Gustavo, Sergio, Fernando y Luis Alberto. A las amistades formadas en las generaciones MIA2K9 y MIA2K10, y a quienes conocí en el DIA, tanto ex alumnos, como becarios o futuros alumnos de la MIA.

Al Jurado revisor de este trabajo, por sus comentarios y aportaciones realizadas. Al Dr. Jesús Antonio González Bernal, gracias por la amistad y el apoyo ofrecidos desde MICAI'10; al Dr. Nicandro Cruz Ramirez, quien desde la Facultad me ha ayudado a resolver cuanta duda se me ha presentado; y al Dr. Manuel Martinez Morales gracias por la amistad ofrecida desde el inicio de la Maestría.

Al cuerpo académico de la MIA, por sus excelentes clases, que me permitieron formarme como MIA, pero mas aun, por resolver con paciencia las dudas que nunca faltaban. Especialmente al Dr. Hector Gabriel Acosta Mesa y al Dr. Guillermo de Jesús Hoyos Rivera.

Al personal administrativo del DIA, porque definitivamente su calidad como personas hizo mas agradable el recorrido.

Al Consejo Nacional de Ciencia y Tecnología. Primeramente, al fondo de becas por otorgarme la beca de posgrado número de becario/beca 273098/223066, para estudiar la Maestría en Inteligencia Artificial en la Universidad Veracruzana, y al proyecto 78910 de Ciencia Básica, del año 2007, por la publicación de mis resultados.

Dedicatorias

A mis padres y hermanas, mi fortaleza.

A Tommy, mi angel.

A mis sobrinos, mi motivación.

A mi familia, mi protección.

A mis amigos, mi apoyo.

A mis profesores, mi orientación.

A ti, mi inspiración.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1
Introducción

1.1. Marco Teórico

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).

El estudio de los sistemas multiagente en la actualidad considera muchos campos de estudio importantes, como aprendizaje, simulación social, auto organización, planeación y robótica colectiva, entre otras; y distintas aplicaciones en disciplinas como economía, cibernética y filosofía.

En este trabajo de investigación se hace énfasis en el estudio del aprendizaje Intencional en sistemas multiagente; y se desenvuelve dentro tres áreas de investigación en el estudio de la Inteligencia Artificial: Agencia BDI: *AgentSpeak(L)* y Jason, Aprendizaje Intencional y Aprendizaje Lógico Inductivo.

1.1.1. Agentes BDI: AgentSpeak(L) y Jason

Dentro del contexto de la Inteligencia Artificial, particularmente dentro del estudio de sistemas multiagente, el modelo de agentes racionales BDI *(Belief-Desire-Intention)* ha resultado ampliamente relevante. 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 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)*.

1.1.2. Aprendizaje Intencional

Aunque 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 tanto en sistemas monoagente como 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 de 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 más el conocimiento de fondo *(background)*, se le conoce como una interpretación del agente. 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 (Quinlan, 1986) o C4.5 (Quinlan, 1986), donde los nodos son atributos.

1.1.4. Caso de estudio: Estrategias de compromiso

La estrategias de compromiso en agentes racionales nos servirán como caso de estudio para comprobar la adaptabilidad de un agente a su entorno. Esto es, bajo que circunstancias un agente debe abandonar o no una intención previamente adoptada. En este caso, el objetivo de aprendizaje no es únicamente la razón para adoptar una intención, sino también la razón para abandonarla.

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. Planteamiento del problema

Actualmente, se cuenta con una clase de agente, capaz de aprender Intencionalmente las razones para ejecutar un plan, es decir, puede aprender un contexto de plan nuevo, una vez que haya fallado en la ejecución de un plan. Para ello, se hace uso del sistema ACE/TILDE (Blockeel & De Raedt, 1998) como un comando externo.

Ortiz-Hernández (2007) describe formalmente en su tesis de maestría el mecanismo necesario para ejecutar el aprendizaje Intencional con base en las interpretaciones que el agente obtenga de su entorno. Una vez obtenidas estas interpretaciones, se ejecuta el algoritmo de inducción como un comando externo, y se leen los archivos generados por esta ejecución. La figura 1.1 ilustra como es ejecutado el aprendizaje hasta ahora: Se obtiene el estado BDI y se generan los ejemplos a través de acciones internas de Jason (ver sección 5.1), después de ésto, se generan 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.

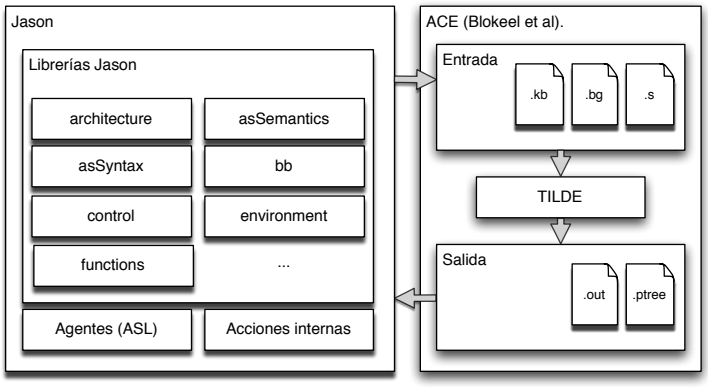


Figura 1.1 Modo operacional del aprendizaje Intencional antes de JILDT.

Tomando en cuenta la situación actual del problema, nos enfrentamos a un problema principal, el cual consiste en la implementación de un algoritmo de inducción de árboles lógicos de decisión, que nos permita sustentar el aprendizaje Intencional en sistemas multiagente, basándose en las interpretaciones de los agentes.

Si bien es cierto que se puede sustentar el aprendizaje a través del sistema ACE/TILDE, no es una solución multiplataforma viable, ya que el sistema ACE/TILDE se ejecuta únicamente en sistemas operativos UNIX. Es por ello que se desarrollará 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 podrá ser manipulado de acuerdo al problema que se esté intentando resolver, cosa que no se puede hacer en su implementación en ACE/TILDE, dado que no se considera Software Libre.

No obstante, el algoritmo de inducción de árboles lógicos de decisión, no es lo único que se implementará en este trabajo, también se busca desarrollar una librería que genere el conjunto de archivos de aprendizaje necesarios para la inducción, y de esta manera se pueda obtener el conocimiento adquirido para modificar los planes de dos clases personalizadas de agente, de las cuales, una clase cuenta con una estrategia de compromiso racional *(single-minded)* que nos servirá como caso de estudio de racionalidad.

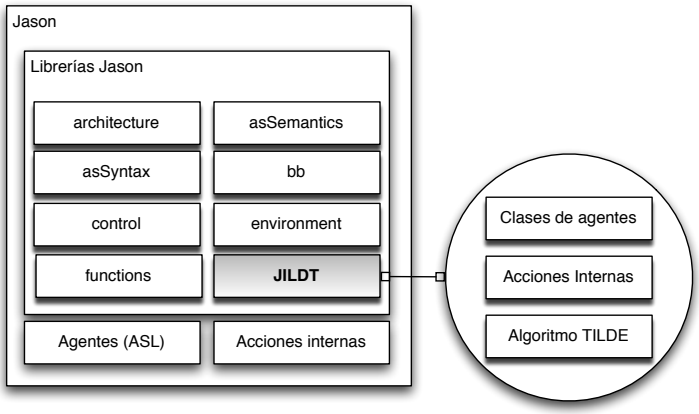


Figura 1.2 Propuesta de integración de JILDT a Jason *AgentSpeak(L)*.

1.3. Hipótesis

La hipótesis sugerida es que se puede sustentar el aprendizaje Intencional en sistemas multiagente, implementando el método de inducción de árboles lógicos de decisión como parte del conjunto de acciones internas de Jason *AgentSpeak(L)*; y que tal aprendizaje puede aproximar una reconsideración basada en políticas (Bratman, 1987).

1.4. Justificación

Contar con un mecanismo de aprendizaje integrado al conjunto de acciones internas del intérprete *Jason AgentSpeak(L)* permitirá a los agentes adaptarse a su entorno. Por ello, implementar una clase de agente capaz de aprender Intencionalmente sobre las razones para ejecutar un plan, nos permitirá apreciar cómo un agente va adaptándose a su entorno. 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.

1.5. Objetivos

El objetivo principal de este proyecto es desarrollar una librería programada en Java, la cual pueda ser empleada dentro del entorno de programación orientada a agentes Jason *AgentSpeak(L)*, y permita sustentar el aprendizaje Intencional en agentes definidos bajo ciertas especificaciones (ver sección 5.2). Para conseguir este objetivo, el proyecto se divide en dos fases:

- Fase I.
  - Desarrollar un conjunto de acciones internas de Jason que permitan configurar el conjunto de archivos que conforman la entrada del algoritmo de aprendizaje a partir del estado BDI de los agentes; y recuperar el conocimiento adquirido después de ejecutar el algoritmo de aprendizaje, a través de un comando externo que ejecuta al sistema ACE/TILDE (Blockeel et al., 1999).
  - Implementar una clase de agente que ejecute intencionalmente un plan de aprendizaje cuando sea necesario y redefina sus planes originales *(intentionalLearner)*.
  - 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)*.
- Fase II.
  - Dado que una de las características de esta librería de aprendizaje es que debe ser multiplataforma, se implementará en Java un algoritmo de Inducción de Árboles Lógicos de Decisión, como una acción interna de Jason.

1.6. Exposición y método

El documento está dividido en tres partes. La primera parte (capítulos 2-4) presenta los **conceptos básicos** para entender el estado del arte en el que se encuentra este trabajo de investigación. El capítulo 2 introduce los conceptos de agencia BDI, Intencionalidad y estrategias de compromiso, donde estas últimas son caso de estudio en esta investigación, como se mencionó anteriormente; en el capítulo 3 se describe formalmente la sintaxis y semántica de *AgentSpeak(L)* y su intérprete Jason. El capítulo 4 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 5-7) se expone la **implementación** de la librería de aprendizaje JILDT. En el capítulo 5, se explica a detalle el *modus operandi* de JILDT, mientras que en el capítulo 6 se presentan algunos experimentos realizados para probar esta librería. Una discusión sobre este trabajo es presentada en el capítulo 7, que incluye conclusiones, una comparativa con trabajos relacionados y una breve descripción de los trabajos futuros.

Finalmente, en la tercera parte se añade a manera de **apéndice**, el código de los agentes definidos para realizar los experimentos de la sección 6.1 y los artículos publicados en MICAI'10 y EUMAS'10.

Parte I

Conceptos Básicos

Capítulo 2
Agencia

Antes de describir a detalle los resultados 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. Empezaré introduciendo el término **agente**, el cual, 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 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 de acuerdo a Rao & Georgeff (1991), los cuales vienen a ser un caso de estudio en el transcurso de esta investigación.

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

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 en nuestro entorno, y además están interconectados, 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:

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 sub-sistemas 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.

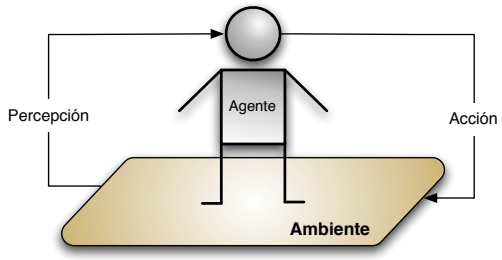


Figura 2.1 Abstracción de un agente a partir de su interacción con el medio ambiente.

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 desempeño** que define el criterio de éxito, y permite cuantificar el nivel de correctez.
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 esto 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 agentes intencionales BDI *(Belief-Desire-Intention)* ha resultado de gran relevancia dentro del contexto de la IA, y en particular 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 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. 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 *(Beliefs)*, deseos *(Desires)* e intenciones *(Intentions)*, 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 evento aplicable y relevante, el cual se convertirá en un deseo. Teniendo éste 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, los cuales nos 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.*

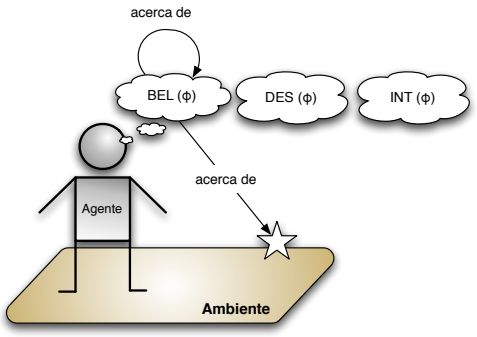


Figura 2.2 Las actitudes proposicionales son representaciones de segundo orden.

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, más 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 humanos. 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, fué 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 con 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 Disneylandia, 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; 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

Como se mencionó anteriormente, 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.

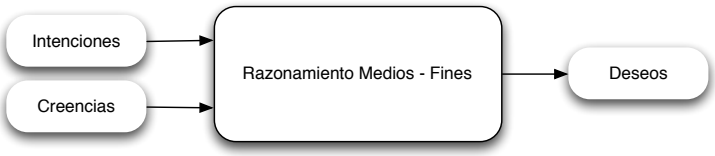


Figura 2.3 Razonamiento medios-fines.

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:

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

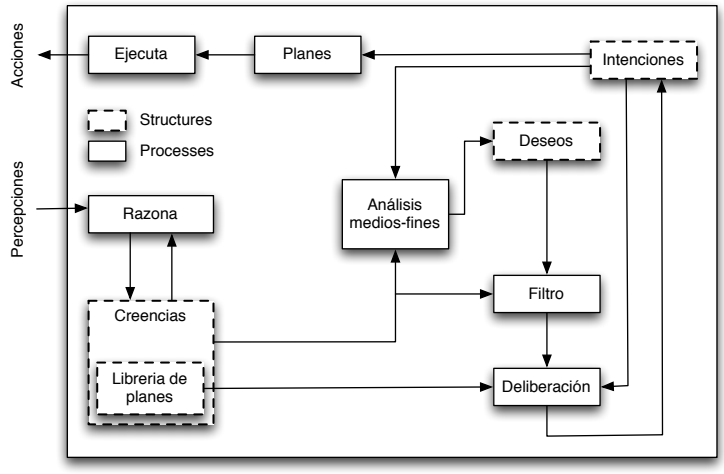


Figura 2.4 Arquitectura para agentes racionales basada en IRMA (Bratman, 1987).

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.
- **Elocuciones.** Es el acto que se lleva a cabo al decir algo: promesas, advertencias, informes, solicitudes. Una enunciación tiene una fuerza elocutoria F si el agente que la enuncia intenta llevar a cabo la elocución F con ese acto. Los verbos que nombran a las elocuciones 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 elocuciones 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 elocución si y sólo si intenta que el receptor reconozca su Intencionalidad en ese acto, al reconocer la fuerza elocutoria 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 elocutoria** de los actos de habla y la **actitud** de los estado 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. Y 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.

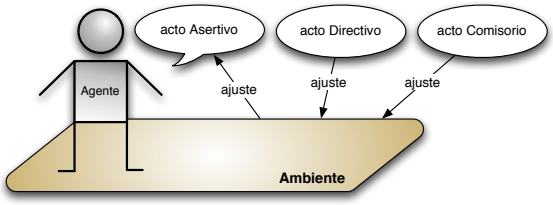


Figura 2.5 Direccionalidad en el ajuste entre los actos de habla y el medio ambiente del agente.

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.

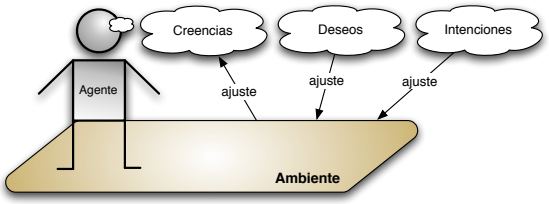


Figura 2.6 Direccionalidad en el ajuste entre los estados Intencionales BDI y el medio ambiente del agente.

- 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.

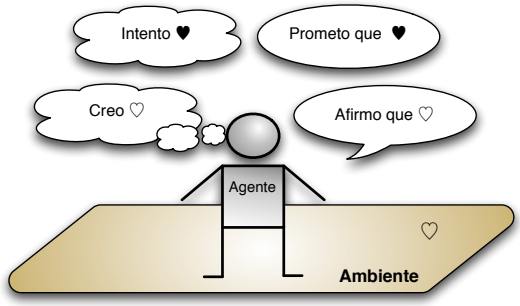


Figura 2.7 Los actos de habla ilocutorios expresan estados Intencionales que son a su vez la condición de sinceridad de lo expresado.

- 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 deseos se satisfacen si son logrados; mis intenciones se satisfacen si son llevadas 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 la 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 haca 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.

AgentSpeak(L) es un lenguaje de programación basado en una lógica restringida de primer orden con eventos y acciones; y representa un marco de trabajo elegante para programar agentes BDI. *Jason*, por su parte, es un intérprete fiel del lenguaje *AgentSpeak(L)*, programado en Java. Ambos, han sido ampliamente aceptados por la comunidad de investigación en SMA y se describen en el capítulo 3.

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 cuando y como se deben eliminar intenciones.

Dadas estas definiciones, un nuevo paradigma de programación ha emergido, la *Programación Orientada a Agentes*, introducida por Shoham (1990). El siguiente capítulo, describe un lenguaje de Programación Orientado a Agentes: *AgentSpeak(L)*, el cual presenta un marco de trabajo elegante para programar agentes BDI, basado en lógica de primer orden; y su intérprete *Jason*. Ambos, han sido ampliamente aceptados por la comunidad de investigación en SMA.

Capítulo 3
AgentSpeak(L) y Jason

Razonar acerca de los agentes Intencionales BDI es importante. Los métodos formales nos permiten especificar, diseñar y verificar a nuestros agentes racionales artificiales. Sin embargo, Rao (1996) observó que una gran parte de las implementaciones exitosas de la arquitectura BDI asumían una serie de simplificaciones, modelaban las actitudes proposicionales como estructuras de datos, y mejoraban su desempeño con base en los planes programados por el usuario, dando como resultado una aparente falta de de fundamentos teóricos. El problema de fondo es que las lógicas multimodales BDI, empleadas en la especificación de estos sistemas, eran ajenas a los problemas prácticos de su programación. Los primeros intentos para resolver este problema, se concentraron en proponer una arquitectura abstracta BDI como una idealización de la implementación de estos sistemas, que fungiera como un vehículo para continuar investigando las propiedades teóricas de la agencia Intencional. Estos trabajos culminaron en dMARS (d'Inverno et al., 1998). El problema con este enfoque es que, debido a su nivel de abstracción, no fue posible establecer una correspondencia de uno a uno entre la teoría del modelo, la teoría de prueba y el intérprete abstracto propuesto.

Con *AgentSpeak(L),* una versión simplificada de dMARS, Rao (1996) propone un camino diferente. *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[1]. 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 y al igual que éste último, ha resultado fundamental para el estudio de los lenguajes de programación orientados a agentes.

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

En la primera sección de este capítulo se describe la sintaxis y la semántica del lenguaje $AgentSpeak(L)$, así como su teoría de prueba. Después de esto, la segunda sección presenta desde una perspectiva formal la descripción de *Jason* (Bordini et al., 2007), una implementación en Java de este lenguaje de Programación Orientado a Agentes.

3.1. Lenguaje *AgentSpeak(L)*

Este lenguaje abstracto de programación orientada a agentes fue 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 implementadores 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.

3.1.1. Sintaxis

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 $b(t1,\ldots,t_n)$ $b(\mathbf{t})$ 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.*

El cuadro 3.1 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).

```
on(b,a).
on(a,table).
on(c,table).
clear(table).
clear(X) :- not(on(_,X)).

@p1
+!put(X,Y) : clear(X) <- move(X,Y).
@p2
+!put(X,Y) : not (clear(X)) <- .print("Imposible mover el bloque ",X," al bloque ",Y).
```

Cuadro 3.1 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** es 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 conveniencia 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 3.1 (líneas 7-10) describe los planes a ejecutarse para poner un bloque X sobre el bloque Y cuando, el bloque X esté libre (línea 8) o cuando el bloque X no esté libre (línea 10).

3.1.2. Semántica operacional

Definición 8 (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 9 (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 10 (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 podemos definir formalmente los conceptos de plan relevante y aplicable. Para ello necesitamos recordar el concepto de unificador más general (*MGU*):

Definición 11 (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 12 (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 13 (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 14 (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.*

Por ejemplo, si asumimos que el evento disparador seleccionado de la cola de eventos E es $\mathcal{S}_{\mathcal{E}} = !put(a,c)$, entonces los planes etiquetados como $p1$ y $p2$ (Ver el cuadro 3.1, líneas 7 y 9) son relevantes para este evento con unificador relevante $\{X/b, Y/c\}$.

Definición 15 (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.*

Continuando con el mismo ejemplo ($!put(a,c)$), y asumiendo que las creencias básicas del agente son las que se muestran en el cuadro 3.1, tenemos que el unificador aplicable es $\{X/a\}$ y que el único plan aplicable es $p2$, debido a que el bloque a no está libre.

Dependiendo del tipo de evento (interno o externo), la intención será diferente. En el caso de los eventos externos, los medios se obtienen seleccionado un plan aplicable para el evento y entonces se aplica el unificador aplicable al cuerpo del plan. Este medio es utilizado para crear una nueva intención que se agrega a I.

Definición 16 (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 17 (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 $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \cdots \wedge c_m \leftarrow !g(t); k_2; \ldots; k_n \ddagger (+!g(s) : b_1 \wedge \cdots \wedge b_j)\sigma \leftarrow (h_1; \ldots; h_k)\sigma; (k_2; \ldots k_n)\sigma] \in I$.*

No abundaremos más en los ejemplos, las siguientes funciones determinan como afecta a una intención i la ejecución de metas, acciones y submetas logradas.

Definición 18 (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 19 (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 20 (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 21 (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 3). Las funciones $top, push, first$ y *rest* tienen semántica evidente.

3.1.3. 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 22 (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 23 (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[2]. 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.

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

Algoritmo 3 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

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:

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

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 24 (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).

3.2. Jason

Jason (Bordini et al., 2005; Bordini & Hübner, 2006; Bordini et al., 2007) es un intérprete que implementa una semántica operacional extendida de $AgentSpeak(L)$ en Java. 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).

```
1  /* Creencias iniciales */
2  factorial(0,1).
3
4  /* Metas iniciales */
5  !print_factorial(5).
6
7  /* Planes */
8  @p1
9  +!print_factorial(N) <-
10   !factorial(N,F);
11   .print("Factorial de ",N," es ",F).
12 @p2
13 +!factorial(0,F) <-
14   ?factorial(0,F).
15 @p3
16 +!factorial(N,F) : N > 0 <-
17   !factorial(N-1,F1);
18   F = F1 * N.
```

Cuadro 3.2 Un agente que imprime el factorial de 5. Adaptado de Bordini et al. (2007).

- **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.
- Incorpora un ambiente de desarrollo basado en jEdit, que facilita 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.

3.2.1. Sintaxis de Jason

Consideremos un programa válido en *Jason* para ejemplificar las expresiones bien formadas (fbf) de este lenguaje de programación. El cuadro 3.2 muestra un programa válido de un agente cuya meta inicial es imprimir el factorial de cinco (línea 5). Los planes de este agente están etiquetados como $@p1$, $@p2$ y $@p3$ (líneas 8, 12 y 15).

Ahora bien, el cuadro 3.3 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. Por ejemplo, `factorial(0,1)` indica que el agente cree que el factorial de cero es uno.

Cada **plan** $p \in ps$ tiene la forma $te : ct \leftarrow h$, donde:

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

Cuadro 3.3 Sintaxis de *Jason*. Adaptada de Bordini et al. (2007).

- *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. Por ejemplo, el evento disparador del plan $@p2$ (Cuadro 3.2, línea 12) dice que ese plan es relevante cuando el agente agrega como meta, lograr el factorial de cero. Obsérvese que, 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. Por ejemplo, el contexto del plan $@p3$ (Cuadro 3.2, línea 15) nos dice que ese plan es aplicable para computar el factorial de números mayores que cero. 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. Por ejemplo, el cuerpo del plan @$p1$ (Cuadro 3.2, líneas 8-11) es una secuencia formada por una meta y una acción primitiva.

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.

3.2.2. Semántica Operacional de 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 $T_i = \emptyset$. 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 $Ctxt(p) = ct$.

3.2.2.1. Consecuencia lógica y definiciones auxiliares

Antes de definir las reglas semánticas de Jason, son necesarias algunas definiciones auxiliares.

Definición 25 (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 26 (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 27 (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 28 (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.

3.2.2.2. Reglas de transición

Las reglas de transición que definen la semántica operacional de *Jason* inducen el ciclo de razonamiento mostrado en la figura 3.1. 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.

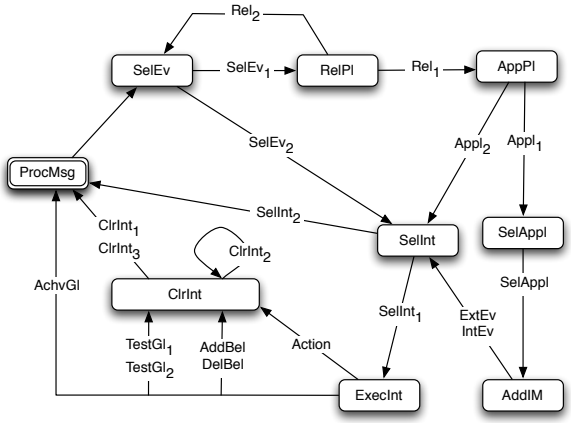


Figura 3.1 El ciclo de razonamiento de Jason. Adaptado de Bordini et al. (2007), p. 206, en Guerra-Hernández et al. (2009).

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

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.

$$(\textbf{Rel}_\textbf{1}) \quad \frac{T_\varepsilon = \langle te, i\rangle, RelPlans(ag_{ps}, te) \neq \{\}}{\langle ag, C, M, T, RelPl\rangle \rightarrow \langle ag, C, M, T', AppPl\rangle}$$

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

$$(\textbf{Rel}_\textbf{2}) \quad \frac{RelPlans(ps, te) = \{\}}{\langle ag, C, M, T, RelPl\rangle \rightarrow \langle ag, C, M, T, SelEv\rangle}$$

El caso de los planes aplicables es parecido:

$$(\textbf{Appl}_\textbf{1}) \quad \frac{ApplPlans(ag_{bs}, T_R) \neq \{\}}{\langle ag, C, M, T, ApplPl\rangle \rightarrow \langle ag, C, M, T', SelAppl\rangle}$$

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*, la cual también podrá ser seleccionada a causa de 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[3].

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

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

[3] 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.

$$\textbf{Test}_2 \quad \frac{Test(creencias,\phi) \neq \emptyset}{C, creencias \rightarrow C', creencias} \qquad C_\iota = i[head \leftarrow ?at; h]$$
$$donde: C'_\iota = \emptyset, C'_I = C_I\{C_\iota\} \cup \{i[(head \leftarrow h)\theta]\}$$
$$theta \in Test(creencias,\phi)$$

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:

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

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

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 $g\theta = TrEv(head)$

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

3.3. Instalación y ambiente de desarrollo

Jason puede descargarse desde su página principal en sourceforge[4], donde además se encuentra una descripción del lenguaje, documentación, ejemplos, demos y proyectos relacionados. Al descargar el archivo de instalación, se obtiene un directorio como el mostrado en la figura 3.3. Lo importante es que el archivo ejecutable, en este caso `Jason.app` esté en algún sitio accesible. Si se ejecuta `Jason.app` se tiene acceso a la ventana principal de un ambiente de desarrollo basado en jEdit (Ver la figura 3.3).

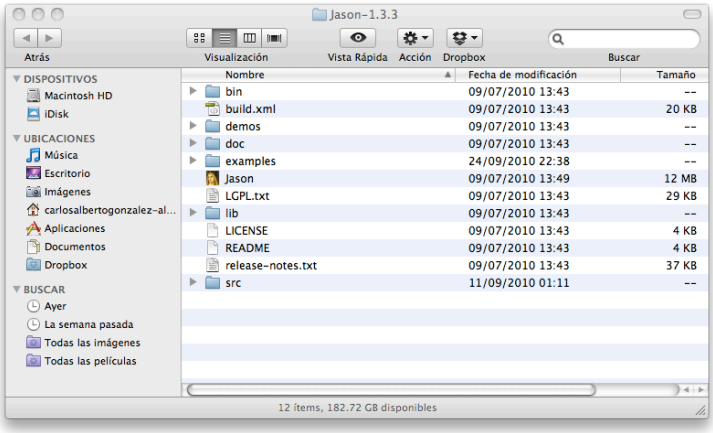


Figura 3.2 El directorio principal de Jason.

3.4. Resumen

In 1996, Anand Rao diseño un lenguaje abstracto de programación orientado a agentes basado en la arquitectura BDI, el cual nombró *AgentSpeak(L)*. En su concepción original, *AgentSpeak(L)* 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 son lo suficientemente expresivas como para especificar, definir y verificar este tipo de agentes, su alta complejidad hacía que los implementadores prefirieran usar estructuras de datos para representar a los operadores Intencionales, y se distanciasen de su definición como operadores modales.

[4] `http://jason.sourceforge.net/Jason/Jason.html`

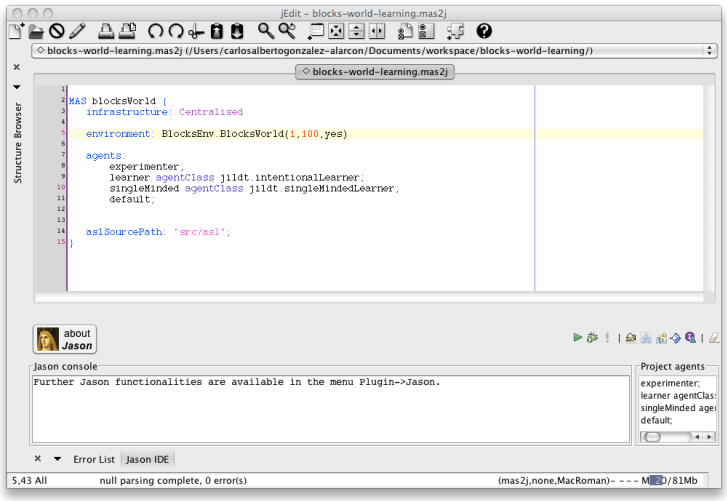


Figura 3.3 La ventana principal de Jason.

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.

Por otro lado, en el presente capítulo se presentó formalmente la implementación en Java de AgentSpeak(L): *Jason*. El cual cuenta con una semántica operacional completa y extendida, y que entre sus caracteristicas cuenta con comunicación entre agentes basados en actos de habla a través del protocolo KQML y la posibilidad de ejecutar sistemas multiagente distribuidos sobre una red, empleando protocolos como SACI y Jade. Tanto *AgentSpeak(L)* como *Jason*, son ampliamente utilizados en el área de investigación de SMA.

Capítulo 4
Aprendizaje Lógico Inductivo

El **aprendizaje** es un fenómeno multifacético que incluye adquisición de conocimiento nuevo, 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 los agentes *AgentSpeak(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 la versión en primer orden de los árboles de decisión: los **árboles lógicos de decisión**, los cuales hacen uso de enunciados en primer orden y un lenguaje basado en las interpretaciones del agente; el sistema ACE/TILDE es presentado, como antecedente del algoritmo JILDT/TILDE, descrito en el capítulo 5.

4.1. 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 4.1).

- 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.

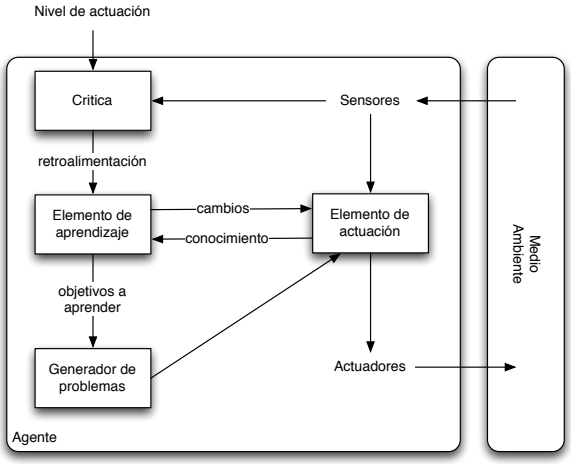


Figura 4.1 Arquitectura abstracta de un agente que aprende. Adaptada de (Russell & Norvig, 2003).

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 puedan mejorar su funcionamiento para realizar una tarea de forma automática, tomando como base su experiencia. 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 puede entenderse 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 de la siguiente manera:

$$F : E \rightarrow A$$

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, únicamente 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 29 (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, los cuales se describen más adelante en la sección 4.2.

El concepto de aprendizaje es central dentro 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 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 este 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 una clasificación para el aprendizaje, basada en el nivel de conocimiento social que tiene cada agente dentro de un SMA, es decir, en que forma 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**: En este nivel, 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**: En este nivel, 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 de los agentes, con su estado mental BDI y la comunicación a través de los actos de habla.
- **Nivel 3**: En este nivel, 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

4.1.1. 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.

La tesis de maestría de Ortiz-Hernández (2007) describe una semántica operacional[1] para definir agentes capaces de aprender Intencionalmente. A continuación se introduce una notación auxiliar que se empleará en esta semántica operacional. Dado un estado de configuración de *AgentSpeak(L)* (definición 23, página 38), C, escribimos C_E para hacer referencia al componente E de C. De forma similar para todos los demás componentes de C. También usaremos $i, i', ...$ 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. En la sección 3.2.1, página 41, se menciona que 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
+!put(X,Y) : true <-  move(X,Y).
```

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

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

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

donde I es la intención actual del agente, Bs es el conjunto de creencias del agente cuando éste seleccionó el plan p como parte de una intención y $C \in \{success, failure\}$ es la clase del ejemplo de entrenamiento.

A continuación, se describen las reglas semánticas adaptadas de (Ortiz-Hernández, 2007) para incorporar **aprendizaje intencional** en Jason. Primero, se definen las reglas para recolectar los ejemplos de entrenamiento de aquellos planes supervisados. **RecInt** agrega una creencia de la forma $intending(I, Bs)$, donde I indica la intención actual del agente y Bs el conjunto de creencias del agente al adoptar la intención I.

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

s.t. $TE(p) = l, ag_{bs} \models intending(l, Bs)$.

Una vez que se conoce el resultado de la ejecución del plan, y éste es satisfactorio, se agrega una creencia de la forma $exampleI, B, succ$.

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

s.t. $TE(p) = l \wedge ag'_{bs} \not\models intending(l, Bs), ag'_{bs} \models example(l, Bs, succ)$.

Las situaciones de fallo incluyen:

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

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

s.t. $TE(p) = l, ag'_{bs} \not\models ex(l, B, exec), ag'_{bs} \models ex(l, B, fail), C'_E = C_E \cup \{\langle +fail(l), i[p] \rangle\}$.

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

s.t. $TE(p) = l, ag_{bs} \not\models ex(l, B, exec), ag'_{bs} \models ex(l, B, fail), C'_E = C_E \cup \{\langle +fail(l), i[p] \rangle\}$.

2. Una meta de verificación falla:

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

s.t. $TE(p) = l, ag_{bs} \not\models ex(l, B, exec), ag'_{bs} \models ex(l, B, fail), C'_E = C_E \cup \{\langle +fail(l), i[p]\rangle\}$.

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. $TE(p) = l, ag_{bs} \not\models ex(l, B, exec), ag'_{bs} \models ex(l, B, fail), C'_E = C_E \cup \{\langle +fail(l), i[p]\rangle\}$.

Cabe hacer mención que la librería JILDT únicamente da 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. $NewCtxt = Tilde(p, E(p)), ag'_{bs} \not\models E(p), ag'_{ps} = ag_{ps} \setminus p, ag'_{ps} = ag_{ps} \cup TE(p) : NewCtxt \leftarrow Body(p)$

Como se mencionó anteriormente, 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 el trabajo realizado por Ortiz-Hernández (2007).

4.2. 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, los cuales 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 4.2):

- 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).

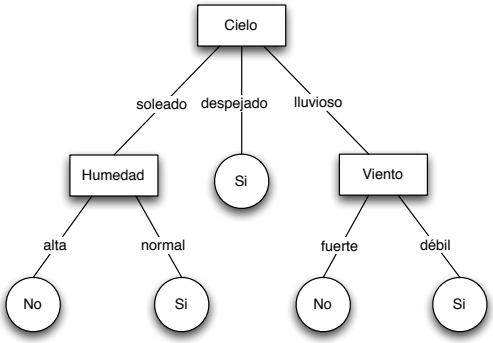


Figura 4.2 Árbol de decisión adaptado de Quinlan (1986).

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 4.2. Para clasificar el ejemplo se seguiría una rama del árbol, lo que representa la siguiente sucesión de pruebas (ver la figura 4.3).*

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

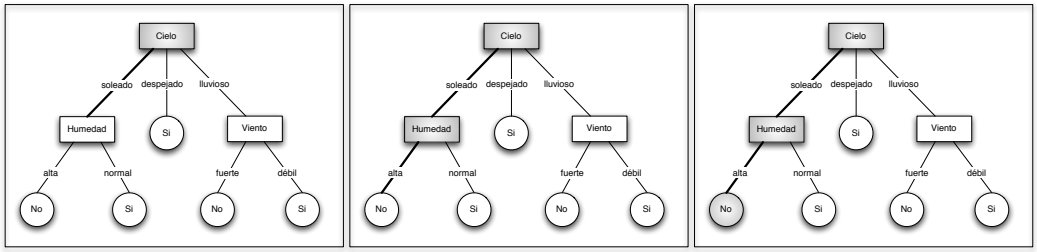


Figura 4.3 Secuencia de pasos para clasificar $\langle$cielo = soleado, temp. = calor, hum.= alta, viento = débil$\rangle$.

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 4 muestra una versión de ID3 según lo expuesto en Russell & Norvig (2003).

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 una rama -sub-árbol- 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

Algoritmo 4 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 = Aande ∈ E
10:            T.child ← ID3(E_v,F)
11:        end for
12:    end if
13:    return T                                ▷ The built tree
14: end procedure
```

de manera recursiva en cada uno de los sub-arboles. 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 30 (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 entropia e_i del intervalo i-ésimo esta dado por la ecuación:*

$$H(i) = -\sum_{i=1}^{k} p_{ij} \log_2 p_{ij} \tag{4.1}$$

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:

$$\Delta_{info}(E,A) = H(E) - \sum_{j=1}^{k} \frac{N(v_j)}{N} H(v_j) \tag{4.2}$$

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**:

$$GM = -\sum_{E_i \in E} \frac{E_i}{E} \log \frac{E_i}{E} \tag{4.3}$$

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

$$GR = \frac{\Delta_{info}}{GM} \tag{4.4}$$

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*. Básicamente el núcleo del algoritmo es el mismo que en el ID3. Sin embargo, se han incorporado algunas mejoras.

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

4.3. Programación lógica inductiva

En la sección 4.1, se introdujo la Hipótesis del Aprendizaje Inductivo (Definición 29), la cual 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. De modo informal, se dice que una teoría aprendida es correcta si es completa (tanto S como CG validan todos los ejemplos de E^+), y si es consistente (ni S 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 31 (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 S, tal que $S \cup CG$ 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 $S \cup CG$ 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 numero 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.

4.3.1. 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 el CG lo que hace la diferencia entre las dos configuraciones de aprendizaje mencionadas. En cuanto al Aprendizaje Inductivo por Implicación (AII), que es el paradigma más utilizado en la PLI, 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 31, el Aprendizaje Inductivo por Implicación se expresa formalmente como sigue (Blockeel, 1998):

Definición 32 (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 33 (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 34 (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 la Tabla 4.1 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 4.1 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 4.4).

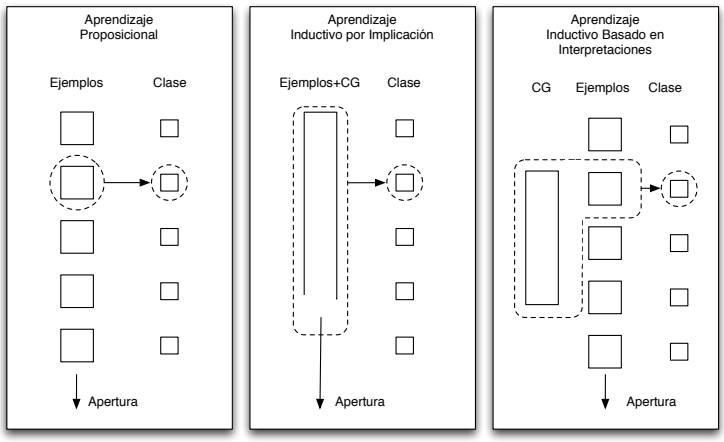


Figura 4.4 Comparación gráfica entre los aprendizajes proposicional, por implicación y basado en interpretaciones, con respecto al supuesto de localidad y la apertura de la descripción de la población de los datos. Adaptado de (Blockeel, 1998).

4.3.2. Árboles Lógicos de Decisión

En la sección 4.2, 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**.

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 35 (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:*

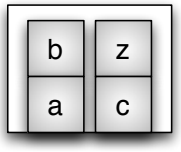
- *Cada nodo contiene una conjunción de literales.*
- *Distintos nodos pueden compartir variables únicamente por la rama izquierda.*

De esta manera, un Árbol Lógico 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 4.5 se muestra un ejemplo de árbol lógico de decisión, el cual equivale a la condición para poner un bloque *A* sobre un bloque *B*. Si se tiene la intención de poner un bloque *A* sobre un bloque *B*, y ambos bloques están libres, la ejecución es satisfactoria *succ*, en caso contrario la ejecución es fallida $fail$.

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



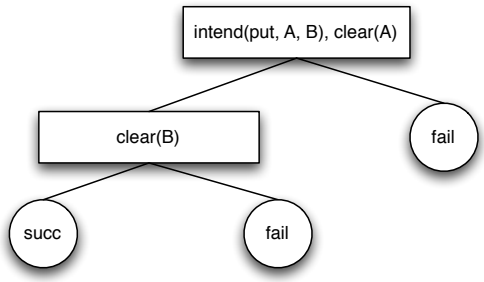


Figura 4.5 Árbol Lógico de Decisión.

y el árbol presentado en la figura 4.5; la sucesión de pruebas indicadas por las literales del árbol sobre el ejemplo sería como se muestra enseguida (ver la figura 4.6):

1. *intend(put,b,c)* $\wedge$ *clear(b)? si, éxito* $\Rightarrow$ *ir por la rama izquierda;*
2. *intend(put,b,c)* $\wedge$ *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* $intend(put,b,c) \wedge clear(b) \wedge clear(c)$*, donde* clear(c) *resultó falsa, de acuerdo a la evidencia.*

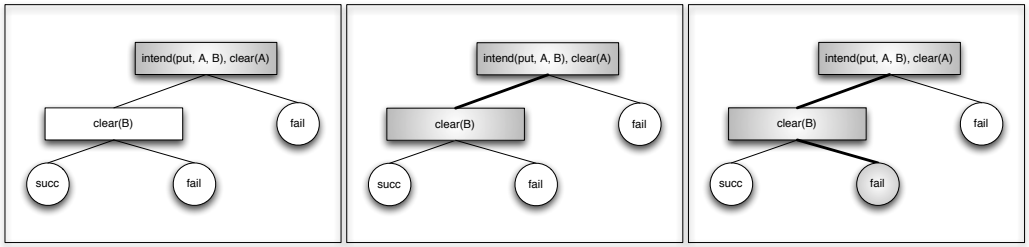


Figura 4.6 Secuencia de pasos para clasificar intend(put,b,c), con la configuración de bloques: ⟨on(b,a), on(a,table), on(c,table), on(z,c)⟩ en un árbol lógico de decisión.

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. Los árboles lógicos de decisión se pueden convertir en programas Prolog. Por ejemplo, el árbol mostrado en la figura 4.5 equivale al programa en Prolog descrito en el cuadro 4.2:

```
class(succ) :- intend(put,A,B), clear(A), clear(B), !.
class(fail) :- intend(put,A,B), clear(A), !.
class(fail).
```

Cuadro 4.2 Programa en Prolog equivalente al árbol lógico de decisión mostrado en la figura 4.5.

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 la siguiente sección.

4.3.3. Sistema ACE/ TILDE

TILDE *(Top-Dow Induction of Logical Decision Trees)*, es un algoritmo desarrollado en la Universidad de Leuven, Bélgica por Blockeel et al. (1999). TILDE hace una búsqueda en profundidad y utiliza interpretaciones para expresar el conjunto de entrenamiento y árboles lógicos para representar la hipótesis.

El sistema TILDE construye un árbol lógico a partir de un conjunto de datos, tomando aquellas literales que tengan una mayor ganancia para situarlas en cada nodo, dividiendo el árbol en dos ramas: una para aquellos ejemplos donde las literales del nodo actual se cumplen (izquierda) y el otro para las que no se cumplen (derecha).

En la sección 5.4 se describe en detalle el algoritmo de inducción, el cual forma parte de la librería de aprendizaje desarrollada en este trabajo de investigación. En esta sección se describen tanto los archivos de entrada que recibe el algoritmo TILDE, como la predisposición del lenguaje necesaria para computar los nodos del árbol.

4.3.3.1. Archivos de entrada

Para computar un árbol lógico de decisión se requiere de tres **entradas**, las cuales están definidas en los siguientes archivos:

1. Knowledge Base File (*.kb)
2. Background File (*.bg)
3. Settings File (*.s)

El archivo de base de conocimiento *(Knowledge Base File)* contiene un conjunto de **ejemplos de entrenamiento**, donde cada ejemplo de entrenamiento es conocido como **modelo**. Para fines de esta investigación, cada modelo se compone del conjunto de creencias que el agente tiene al momento de adoptar una intención; una literal indicando que la intención actual del agente; y una etiqueta que indica si la ejecución de la intención fué exitosa o fallida. En la tabla 4.3 se muestran los modelos correspondientes a los ejemplos en la figura 4.7. La clase de los ejemplos se introduce en la línea 2, y la intención asociada a la línea 3. El resto del modelo corresponde a las creencias del agente cuando éste adoptó la intención.

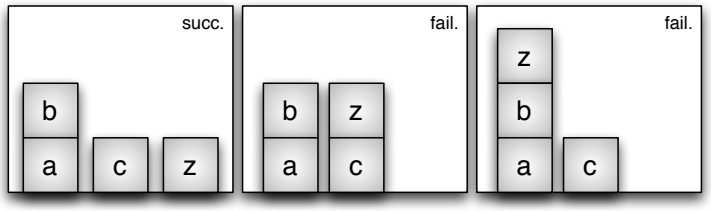


Figura 4.7 Tres ejemplos de entrenamiento del mundo de los bloques cuando se intenta poner el bloque *b* sobre el bloque *c*.

```
1  begin(model(1))
2    succ.
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 4.3 Ejemplos de entrenamiento de la figura 4.7 como modelos de TILDE. Etiquetas de clase en la línea 2.

El archivo de conocimiento general *(Background File)* almacena las reglas que cree el agente y que conforman el **Conocimiento General** sobre el dominio de experiencia del agente.

Por último, el archivo de configuración *(Settings File)* contiene la **predisposición del lenguaje**, es decir, la definición de literales que deben ser consideradas como candidatos para ser incluidos en el árbol lógico de decisión, además de otras configuraciones, como opciones de salida, métrica a utilizar, o nivel de traza de mensajes. El cuadro 4.4 muestra un ejemplo de un archivo de configuración, la predisposición del lenguaje se define en las líneas 6-13, y emplean directivas *rmode* y *lookahead*, las cuales se explican en la siguiente sección.

```
load(models).
talking(0).
output_options([c45,prolog,lp]).
classes([succ,fail]).

rmode(clear(+V1)).
rmode(on(+V1,+V2)).
rmode(on(+V2,+V1)).
rmode(intend(put,-V1,-V2)).
lookahead(intend(put,V1,V2),clear(v1)).
lookahead(intend(put,V1,V2),clear(v2)).
lookahead(intend(put,V1,V2),on(v1,V2)).
lookahead(intend(put,V1,V2),on(v2,V1)).
```

Cuadro 4.4 Ejemplo de un archivo de configuración *(settingfile.s)*.

4.3.3.2. Predisposición del lenguaje

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 la Tabla 4.5 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 4.5 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. La aplicación de esta última construcción es muy importante dentro de esta investigación, ya que vincula lógicamente las variables en el plan original, con las variables de las literales candidatas, lo que permite la generalización.

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 4.6).

```
Compact notation of tree:

intend(put,-A,-B),clear(B) ?
+--yes: [succ] 4.0 [[succ:3.0,fail:1.0]]
+--no:  [fail] 2.0 [[succ:0.0,fail:2.0]]

Equivalent prolog program:

class([succ]) :- intend(put,A,B),clear(B), !.
% 3.0/4.0=0.75
class([fail]).
% 2.0/2.0=1.0
```

Cuadro 4.6 Parte del archivo .out, resultado de la ejecución del sistema ACE/TILDE.

4.4. Resumen

El aprendizaje, visto desde un contexto de agencia se entiende como la adquisición de conocimientos y habilidades por un agente, en el que esta adquisición es conducida 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.

Blockeel et al. (1999) presentan un sistema que induce árboles lógicos de decisión: **ACE/TILDE**, el cual es reimplementado en este trabajo como una acción interna de *Jason*. En el siguiente capítulo se describe el *modus operandi* de JILDT, una librería de aprendizaje que induce árboles lógicos de decisión.

Parte II

Implementación

Capítulo 5
Jason Induction of Logical Decision Trees

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, introducido en el capítulo 4.

En este capítulo, se describe de manera más detallada el *modus operandi* de la librería de aprendizaje JILDT. La primera sección, describe las principales acciones internas que fueron implementadas para extender las capacidades de los agentes y así poder sustentar el aprendizaje en agentes definidos bajo dos clases de agente, personalizadas para llevar a cabo el aprendizaje intencional, mismas que se describen en la segunda sección de este capítulo. Dentro de esta sección, se describe también el mecanismo que se lleva a cabo para dar soporte a este tipo de aprendizaje, a través de extensiones a los planes originales. La tercera sección presenta la ontología definida en la librería JILDT, la cual define las palabras reservadas de la librería y que son de vital importancia para apoyar el proceso de aprendizaje. Por último, en la cuarta sección de este capítulo, se describe el algoritmo de inducción de árboles lógicos de decisión implementado en esta librería, extendiendo el texto previamente presentado en la sección 4.3.3.

5.1. Acciones Internas

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*.

La figura 5.1 presenta el diagrama de clases[1] de las acciones internas implementadas en JILDT. 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.

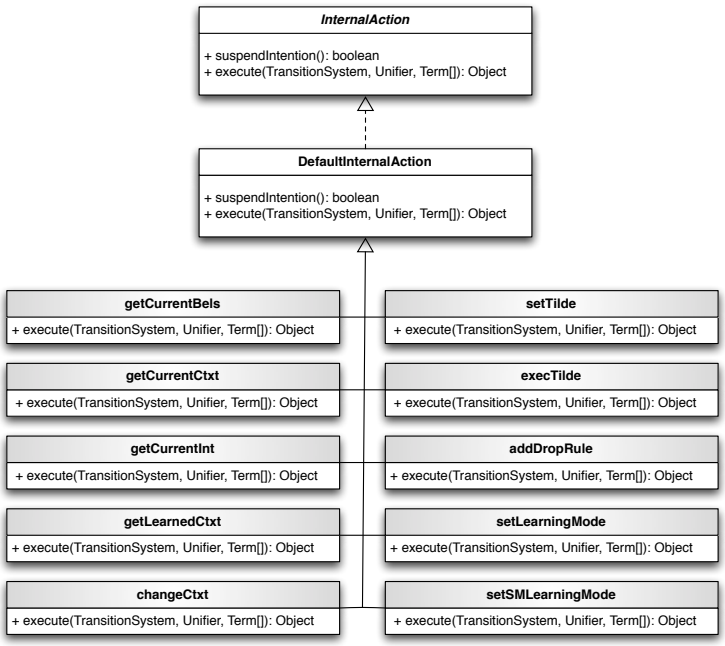


Figura 5.1 Diagrama de clase de las acciones internas en JILDT.

[1] Se emplea una notación UML estándar, donde los atributos y métodos públicos se denotan con el símbolo $+$, mientras que aquellos que son privados se denotan por $-$. 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.

La librería JILDT está definida como un paquete de Java, por lo tanto, cada una de las acciones son accesadas usando el prefijo *"jildt"* más el nombre de 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. A continuación se enlistan las principales acciones internas implementadas en JILDT.

- *jildt.getCurrentCtxt(C)*. Obtiene el contexto del plan que se esté ejecutando, el cual es unificado con la variable *C*.

- *jildt.getCurrentInt(I)*. Permite obtener la intención actual que está ejecutando el agente, unificándola con la variable *I*.

- *jildt.getCurrentBels(Bs)*. Esta acción interna nos permite recuperar la lista con las creencias actuales del agente, las cuales son unificadas con la variable *Bs*. Cabe mencionar, que solo recupera las creencias del agente relacionadas a su problemática, ya que hay creencias que son agregadas al agente como parte del proceso de aprendizaje, las cuales se explican en la sección 5.3

$$Bs = [b_0, b_1, ...b_n] \, donde \begin{cases} b_i \in B \\ b_i \notin \{example, intending, current\_path, ...\} \end{cases}$$

- *jildt.getLearnedCtxt(P,LC,F)*. Obtiene el contexto aprendido después de ejecutar el algoritmo de inducción. *P* contiene el nombre del plan, al que se le redefinirá el contexto; *LC* unifica con una lista de términos que representan el contexto aprendido; por ultimo, $F \in \{change, notchange\}$ es una variable bandera que indica si el contexto debe ser redefinido o no.

- *jildt.changeCtxt(P,LC)* . Esta acción interna, modifica el contexto del plan *P*, por el contexto aprendido *LC*. No unifica con algún resultado.

- *jildt.setTilde(P)*.Los archivos de entrada - *knowledge base file (.kb), background file (.bg) y setting file (.s)* - necesarios para inducir el aprendizaje a través de TILDE (Sección 4.3.3.1) sobre el plan *P*, son generados con esta acción interna, los cuales son generados en la ruta definida por el predicado *current_path/1* del agente, explicado en la sección 5.3. No unifica con algún resultado.

- *jildt.execTilde(T,G)*. El algoritmo de inducción de árboles lógicos de decisión, es ejecutado a través de esta acción interna. *T* es un valor booleano que indica si se desplegará ó no en consola, una traza de las acciones que va realizando el algoritmo de inducción; mientras que *G* es un valor booleano que indica si se desplegará una interfaz gráfica de usuario (GUI) con el árbol obtenido.

- *jildt.addDropRule(LC,P)*. Agrega una regla de abandono del tipo[2] `drop(I):- .intend(I) & not [LC]`, donde *LC* es el contexto aprendido a través del algoritmo de inducción, y *P* es el plan que el agente estaba intentando ejecutar al momento de aprender *LC*. No unifica con algún resultado.

- *jildt.setLearningMode(Ps)*. Modifica los planes de los agentes de tipo *intentionalLearner* (Sección 5.2.1) para habilitar el aprendizaje intencional. *Ps* es la lista de planes que se extenderán. Al ser una acción de configuración, no unifica con algún resultado.

- *jildt.setSMLearningMode(Ps)*. Modifica los planes de los agentes de tipo *singleMindedLearner* (Sección 5.2.1) para habilitar el aprendizaje intencional y reglas de abandono. *Ps* es la lista de planes que se extenderán. Al ser una acción de configuración, no unifica con algún resultado.

5.2. Clases de agente

Como se describe anteriormente en el capítulo 3, 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 (*Bs*), un conjunto de planes (*Ps*), un conjunto de eventos, un conjunto de intenciones, funciones usadas en el ciclo de razonamiento (por ejemplo, la función de actualización de creencias (*BUF)* o 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, se ha personalizado extendiendo las funciones básicas implementadas en ésta, a través de dos nuevas clases de agente: *intentionalLearner* y *singleMindedLearner*. El diagrama de clase en la figura 5.2, presenta las clases implementadas en JILDT, las cuales heredan sus métodos de la clase *Agent*, y sobrescriben el método *initAg* en ambas clases, y el método *selectIntention*, en la clase *singleMindedLearner*.

[2] Anteriormente la regla abandono era del tipo `drop(I):- intending(I,_) & not [LC]`

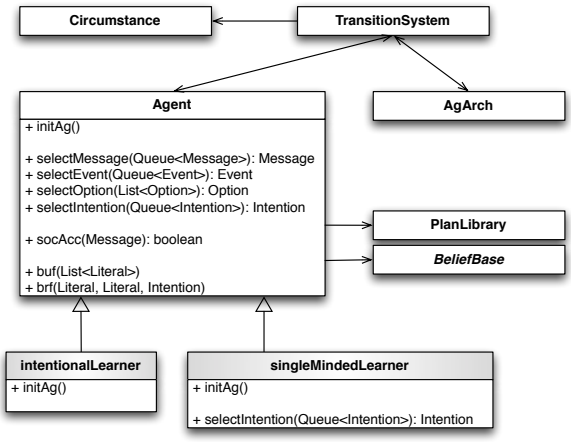


Figura 5.2 Diagrama de clase de los agentes *intentionalLearner* y *singleMindedLearner*

Con la clase **IntentionalLearner**, se pueden implementar 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. El comportamiento de esta clase de agente, 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.
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.
3. Si existe un plan aplicable, pero el contexto del plan no es consecuencia lógica del conjunto de creencias del agente, entonces se ejecuta un plan de rechazo, o plan de fallo no relevante.

El mecanismo anterior se ejemplifica en la figura 5.3. Supóngase que dentro del ciclo de razonamiento del agente, éste está intentado poner el bloque *b* sobre el bloque *c*. En el momento en que la ejecución del plan falla, se dispara el *trigger event* **-!put(b,c)** , el cual activa el plan **+!learning(put)**. Obsérvese que en esta clase de agente, no está siendo modificado el ciclo de razonamiento del agente, sino que el proceso de aprendizaje se lleva a cabo por un mecanismo de extensión de planes, el cual se describe en la sección 5.2.1.

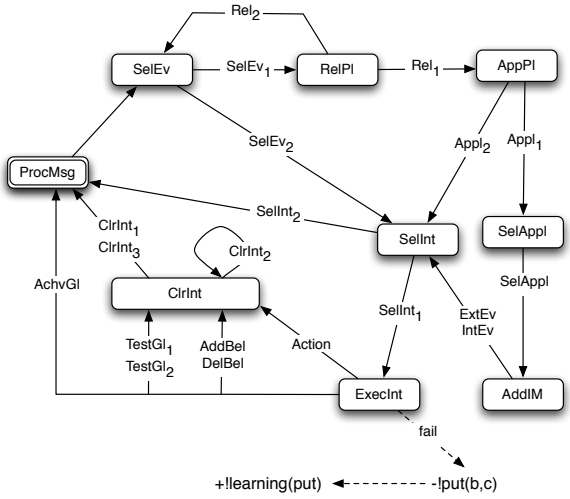


Figura 5.3 Modo operacional de un agente tipo *intentionalLearner*.

De modo muy similar, los agentes definidos como agentes de la clase **singleMindedLearner** son capaces, no solo de redefinir el contexto de sus planes, sino que también pueden adquirir conocimiento general que expresa cuando 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 de decisión, que llevan a un resultado de fallo (*failure*). 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.
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.
3. Si existe un plan aplicable, pero el contexto del plan no es consecuencia lógica del conjunto de creencias del agente, entonces se ejecuta un plan de rechazo, o plan de fallo no relevante.
4. Si al momento de tratar de ejecutar una intención *I*, el agente percibe que *drop(I)* es consecuencia lógica de su conjunto de creencias, entonces dispara el evento *+dropIntention(I)*, que ejecuta un plan etiquetado como `@dropIntention`, el cual forza al agente a abandonar la intención *I*.

El mecanismo anterior se ejemplifica en la figura 5.4 de manera muy similar al mecanismo de un agente tipo *IntentionalLearner*, con la diferencia de que el ciclo de razonamiento es modificado para percibir cuando es racional abandonar una intención ($SelInt_3$).

$$(\textbf{SelInt}_\textbf{3}) \quad \frac{Ag_{bs} \models drop(I)}{\langle ag, C, M, T, SelInt \rangle \rightarrow \langle ag, C', M, T, ClrInt \rangle}$$

donde: $\mathcal{C}'_{\mathcal{I}} = \mathcal{C}'_{\mathcal{I}} / \{\mathcal{I}\}$

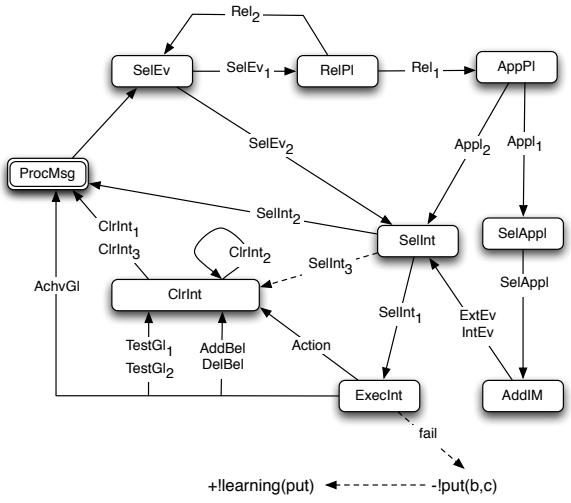


Figura 5.4 Modo operacional de un agente tipo *singleMindedLearner*.

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 *intentionalLearner* 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 35 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 *intentionalLearner* es más racional que un agente *default* de *Jason* (ver la figura 5.5), 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 más racional que un agente de la clase *intentionalLearner*, ya que el primero es capaz de abandonar una intención cuando percibe que no podrá finalizarla con éxito. Esta conclusión puede comprobarse mediante los resultados mostrados en los experimentos en la sección 6.1.

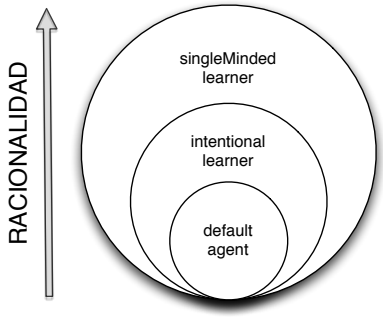


Figura 5.5 Niveles de racionalidad de agentes *default*, *intentionalLearner* y *singleMindedLearner*.

5.2.1. Extensión de planes

El hecho de que Jason, sea un intérprete de *AgentSpeak(L)* programado en Java, facilita la extensión y personalización de sus clases. De forma muy similar a las *acciones internas* y las *clases de agente*, vistos en la secciones 5.1 y 5.2 respectivamente, los planes originales de los agentes pueden ser extendidos, conforme a los protocolos en Guerra-Hernández et al. (2008b), con la finalidad de que se pueda manipular la acción que debe realizar un agente en caso de que se encuentre con un fallo en la ejecución, (la ejecución del plan de aprendizaje), o cuando se encuentre con un plan no ejecutable, el cual no amerite ejecutar un plan de aprendizaje.

Ambas clases de agente, *intentionalLearner* y *singleMindedLearner*, definen un plan para configurar el modo

de aprendizaje adecuado para cada clase de agente (ver el cuadro 5.1), el cual ejecuta la acción interna *jildt.setLearningMode*, en el caso de la clase de agente *intentionalLearner*, o *setSMLearningMode*, en la clase de agente *singleMindedLearner*. Como parámetro, se envia la lista de planes que se extenderán, o la palabra *all*, indicando que se extenderán todos los planes.

```
/* intentionalLearner*/
@initialLearningGoal
+!initialLearningGoal
  <- jildt.setLearningMode(all).
```

```
/* singleMindedLearner*/
@initialLearningGoal
+!initialLearningGoal
  <- jildt.setSMLearningMode(all).
```

Cuadro 5.1 Meta inicial de aprendizaje para las clases de agente *intentionalLearner* y *singleMindedLearner*.

Para ejemplificar en que consiste la extensión de los planes, se introduce un plan *put* simplificado de la implementación de Bordini et al. (2007), el cual se muestra en el cuadro 5.2.

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

Cuadro 5.2 Plan *put*, basado y simplificado de su implementación en Bordini et al. (2007).

El plan original es extendido (ver el cuadro 5.3) en tres partes: la primera parte, o preprocesamiento (líneas 3-5), consiste en obtener información relevante sobre el mundo actual del agente, antes de ejecutar el plan original. Primero se obtiene la intención actual del agente, mediante la acción interna `jildt.getCurrentIntention(I)`, después de obtener la intención actual, se obtienen las creencias del agente al momento de iniciar la ejecución del plan, a través de la acción interna `jildt.getCurrentBeliefs(Bs)`. Una vez que se obtiene el estado BDI del agente, se agrega la creencia `intending(I,Bs)`, que define que el agente está intentando ejecutar el plan *I*, sabiendo que el mundo percibido está dado por *Bs*. La segunda parte, introduce el cuerpo original del plan (línea 6). Por último, en la tercera parte (líneas 7-8), dado que se ejecuta únicamente si el plan original concluye con éxito, se elimina la creencia `intending(I,Bs)`, y se agrega una creencia de ejemplo `example(I,Bs,succ)`, indicando que una vez que se intentó ejecutar el plan *I*, conociendo las creencias *Bs*, la ejecución del plan fue satisfactoria (*succ*).

Si la ejecución de la intención falla, por ejemplo, cuando la acción *move* no puede ser ejecutada correctamente, un plan alternativo es agregado, como el que se muestra en el cuadro 5.4, el cual responde al evento de fallo

```
@put
+!put(X,Y) : true <-
  jildt.getCurrentInt(I);
  jildt.getCurrentBels(Bs);
  +intending(I,Bs);
  move(X,Y);
  -intending(I,Bs);
  +example(I,Bs,succ).
```

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

-!put(X,Y). Este plan es ejecutado si el agente está intentando ejecutar el plan *I*, donde *I*, corresponde al nombre de la meta del plan extendido. La primera acción consiste en eliminar la creencia `intending(I,Bs)`, y se agrega una creencia de ejemplo `example(I,Bs,fail)`, indicando que una vez que se intento ejecutar el plan *I*, conociendo las creencias *Bs*, la ejecución del plan fue fallida (*fail*). Una vez agregada esta última creencia, se intenta ejecutar el plan de aprendizaje `!learning([nombre del plan])` (descrito en la sección 5.2.2), y se agrega la creencia `example_processed(put)`, que arroja un evento *+example_processed(put)*, en el cual se puede manipular por parte del usuario, que acciones debe ejecutar el agente toda vez que existe un fallo en la ejecución de un plan.

```
@put_failCase
-!put(X,Y) : intending(put(X,Y), Bs) <-
  -intending(I,Bs);
  +example(I,Bs,fail);
  !learning(put);
  +example_processed(put).
```

Cuadro 5.4 Plan de fallo agregado por JILDT para manejar fallos que requieren aprendizaje.

También se puede dar el caso en que un plan no sea ejecutable, debido a que el contexto del plan ha sido redefinido, y por tanto, no exista un plan aplicable. Un plan que responde al evento *-!put(X,Y)* es agregado, como el mostrado en el cuadro 5.5, el cual maneja fallos en los cuales no debería ser intentando un plan de aprendizaje. En este caso decimos que el plan *put* es relevante, pero no es aplicable. Es racional evitar comprometerse a ejecutar un plan, si no existen planes aplicables para un evento determinado. Una creencia `non_applicable(put)`, que arroja un evento *+non_applicable(put)* es agregada al conjunto de creencias del agente, en el cual se puede manipular por parte del usuario, que acciones debe ejecutar el agente toda vez que rechaza la ejecución de un plan.

```
1 @put_failCase_NoRelevant
2 -!put(X,Y) : not .intend(put(X,Y)) <-
3   .print("Plan ",put," non applicable.");
4   +non_applicable(put).
```

Cuadro 5.5 Plan de fallo agregado por JILDT para manejar fallos que no requieren aprendizaje.

5.2.2. Plan de aprendizaje

Como se mencionó en la sección anterior, una vez que la ejecución de un plan falla, el agente adopta la intención de aprender (a excepción de cuando el plan falla debido al rechazo de una intención), ejecutando el plan de aprendizaje mostrado en el cuadro 5.6. Primero, imprime un mensaje notificando al usuario que el agente está intentado aprender un mejor contexto; después de esto, se configuran los archivos de entrada (línea 4), a partir de los ejemplos recolectados en las creencias *example/3* y el conocimiento general que el agente posee, o *background*. Una vez generados los archivos de entrada, se ejecuta el algoritmo de inducción (línea 5) y se obtiene el contexto aprendido para el plan asociado (línea 6). Por último, un plan de revisión es ejecutado (línea 6), para redefinir el contexto del plan, en caso de ser necesario.

```
1 @learning
2 +!learning(P): true <-
3   .print(\"Trying to learn a better context...\");
4   jildt.setTilde(P);
5   jildt.execTilde(false,false);
6   jildt.getLearnedCtxt(P,LC,F);
7   !learningTest(P,LC,F).
```

Cuadro 5.6 Plan de aprendizaje.

Se definen dos planes para revisar si se debe redefinir el contexto de un plan o no (cuadro 5.7). Si la ejecución del plan de aprendizaje es satisfactoria computando un árbol lógico de decisión, se ejecuta ejecuta el plan etiquetado como `@learningTestSuccess` (líneas 1-7), donde se cambia el contexto del plan *P* (línea 5), y en el caso de la clase de agente *singleMindedLearner*, se agrega una regla de abandono (línea 7); en caso de que no se haya aprendido un mejor contexto, se ejecuta el plan etiquetado como `@learningTestFail` (líneas 8-10), donde se indica al usuario que no es posible aprender un mejor contexto.

El plan de abandono mostrado en el cuadro 5.8 y etiquetado como `@dropIntention` responde al evento *+dropIntention(I)*, el cual es lanzado una vez que el agente detecta que la creencia `drop(I)` es consecuencia lógica de su conjunto de creencias. Este plan es implementado únicamente en la clase de agente *singleMindedLearner*, y forza al agente a abandonar la intención *I*, mediante la acción interna `.drop_intention(I)`

```
1  @learningTestSuccess
2  +!learningTest(P,LC,change): true <-
3    .print(\"Learned context for \",P,\" is \", LC);
4    .print(\"Changing context in plan \" , P, \" ...\");
5    jildt.changeCtxt(P,LC);
6    .print(\"Context was changed succesfully...\");
7    jildt.addDropRule(P,LC).
8  @learningTestFail
9  +!learningTest(P,LC,notchange) : true <-
10   .print(\"It's not possible to learn better context\").
```

Cuadro 5.7 Planes de revisión del contexto aprendido para el agente *singleMindedLearner*.

(línea 6) que es parte de las acciones internas incluidas en la distribución de *Jason*. Previo a esta acción, el agente elimina la creencia de que está intentando ejecutar la intención *I* (línea 5). Al final, se agrega la creencia `dropped_int(put)` al conjunto de creencias del agente, la cual arroja un evento *+dropped_int(put)* y permite manipular por parte del usuario, que acciones debe ejecutar el agente toda vez que abandona la intención de ejecutar un plan.

```
1
2  @dropIntention
3  +dropIntention(I) : true <-
4    .print(\"Wow!! I'm sorry, I have to abandon my intention\");
5    -intending(I,_);
6    .drop_intention(I);
7    +dropped_int(I).
```

Cuadro 5.8 Plan de abandono de intenciones en el agente *singleMindedLearner*.

5.3. Ontología

En la librería JILDT, se define una ontología que especifica los predicados usados en el contexto del proceso de aprendizaje, los cuales no deben ser utilizados para un uso distinto al que están predefinidos, por lo que estos predicados deben ser tratados como un conjunto de palabras reservadas. A continuación se enlistan estos predicados y su función en el proceso de aprendizaje.

- *drop(I)*. Se utiliza como la cabeza de la regla de abandono, donde *I* unifica con la intención que se abandonará. Ejemplo:

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

- *root_path(R)*. Este predicado define en *R* la carpeta raíz, donde se guardarán los experimentos a ser ejecutados por TILDE. En caso de que no se defina este predicado en el conjunto de creencias del agente, la ruta por defecto es la carpeta en la que se está corriendo el SMA. Ejemplo:

  root_path("/Users/carlos/Desktop/").

- *current_path(P)*. De modo similar a *root_path*, este predicado define en *P* la ruta actual donde se generarán los archivos de configuración para ejecutar el algoritmo de inducción. La ruta se compone de la ruta raíz, más el nombre del agente, más el numero de experimento que está ejecutando. Ejemplo:

  current_path("/Users/carlos/Desktop/singleMinded/exp1/").

- *dropped_int(I)*. Indica que la intención *I* ha sido abandonada. Arroja el evento *+dropped_int(I)*, que permite definir las acciones que el agente realizará al abandonar la intención *I*. Ejemplo:

  +dropped_int(put(X,Y)).

- *non_applicable(P)*. Indica que no existe un plan *P* aplicable. Lanza el evento *+non_applicable(P)*, que permite definir las acciones que el agente realizará al rechazar la intención. Ejemplo:

  +non_applicable(put).

- *example_processed(P)*. Una vez que un se produce un fallo en la ejecución del plan y después de haberse ejecutado el plan de aprendizaje, se agrega una creencia *example_processed(P)*, la cual indica que ha finalizado el proceso de aprendizaje para el plan *P*; dado que lanza el evento *+example_processed(P)*, se pueden manipular las acciones que debe realizar el agente una vez que ha realizado el proceso de aprendizaje. Ejemplo:

  +example_processed(put).

- *intending(I, Bs)*. Indica que el agente aún está intentando ejecutar *I*, conociendo el conjunto de creencias *Bs*. Ejemplo:

  intending(put(b,c), [on(b,a), on(a,table), on(c,table), on(z,table), clear(table)])

- *example(I,Bs,Class)*. Los ejemplos que servirán para inducir los árboles lógicos de decisión, se almacenan utilizando este predicado, donde *I* es la intención que está tratando de ejecutarse, *Bs* es el conjunto de creencias que el agente conocía al momento de adoptar la intención, y $Class \in \{succ, fail\}$, dependiendo si el ejemplo corresponde a una ejecución satisfatoria o fallida. Ejemplo:

  example(put(b,c), [on(b,a), on(a,table), on(c,table), on(z,table), clear(table)],*succ*)

5.4. Algoritmo de inducción JILDT/TILDE

Como se mencionó en la sección 4.3.3, la inducción de árboles lógicos de decisión, o TILDE por sus siglas en inglés (*Top-Down Induction of Logical Decision Trees*), ha sido utilizada para sustentar el aprendizaje en el contexto de agentes Intencionales BDI (Guerra-Hernández et al., 2004b), principalmente porque las entradas requeridas para este método son fácilmente obtenidas del estado mental de tales agentes, dado que son enunciados escritos en lógica de primer orden; 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 el contexto de un plan, y el árbol en sí 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 (ver la figura. 5.6). Esta sección presenta la implementación del algoritmo de inducción de árboles de decisión, descrito en la sección 4.3.3, como una acción interna de *Jason*

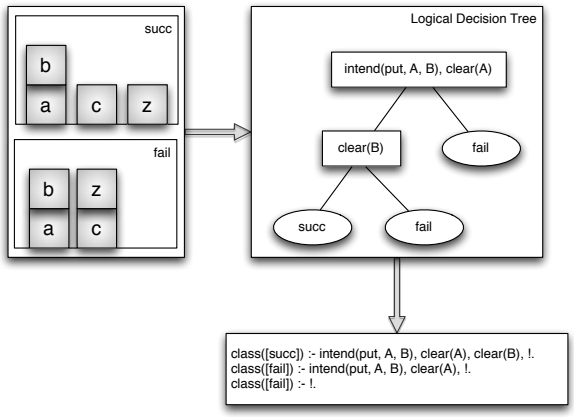


Figura 5.6 Ejemplo de un árbol lógico de decisión, para el mundo de los bloques.

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 5 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 $\rho(Q)$ a través de Q y la predisposición del lenguaje (*language bias*) (directivas *rmode* y *lookahead*

definidos en el archivo de configuración). El candidato que maximice su valor de *gain ratio* (Eq. 5.4) es seleccionado como nodo del árbol (línea 2).

2. El procedimiento se detiene cuando un criterio de parada es alcanzado (línea 3), 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.
4. De otro modo, se computa un nodo interno, en el que el contenido del nodo interno $Conj$ se computa extrayendo la consulta Q del candidato Q_b (línea 6). 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, con la etiqueta $Conj$ (líneas 9-11). Al final, el procedimiento regresa el árbol construido T.

Algoritmo 5 Inducción de Árboles Lógicos de Decisión.

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

El mejor candidato Q_b es seleccionado (Algoritmo 6) de la siguiente manera: Cada candidato en $r_i \in \rho(\leftarrow Q)$ induce una partición en los ejemplos en E; se computa una medida de calidad usando la métrica *Gain Ratio* (Eq. 5.4) (Quinlan, 1993). Para esto, una matriz *counter* (ver la figura 5.7) almacena el número de ejemplos satisfactorios o fallidos para cada clase c (líneas 7-12). El candidato seleccionado es aquel que maximiza la métrica *Gain Ratio* (línea 17).

Dado que *Gain Ratio* es una métrica de información basada en entropía, se explica a continuación la abstracción del término entropía. La entropía es una medida de incertidumbre asociada a un evento; la entropía de un conjunto de ejemplos E está dada por la ecuación 5.1 (Shannon, 1948).

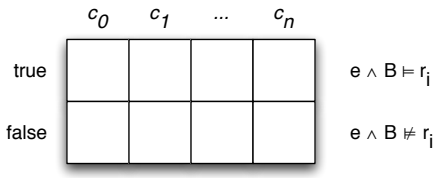


Figura 5.7 Matriz *counter* que almacena el número de ejemplos satisfactorios o fallidos para cada clase c.

Algoritmo 6 Selección del mejor candidato

```
 1: procedure BEST(E,R)                          ▷ E is a set of examples, R = ρ(← Q) refinements of Q
 2:     i := 1;
 3:     for all r_i ∈ R do
 4:         for all c ∈ C do                     ▷ C = {succ, fail}
 5:             counter[true][c] := 0; counter [false][c] := 0;
 6:         end for
 7:         for all e ∈ E do
 8:             if e ∧ B ⊨ r_i then
 9:                 counter[true][class(e)]++;
10:             else
11:                 counter[false][class(e)]++;
12:             end if
13:             s_i := gainRatio(E, counter);
14:         end for
15:         i++;
16:     end for
17:     return Q_b := r_i s.t. max(s_i)
18: end procedure
```

$$s(E) = -\sum_{i=1}^{k} p(c_i,E)\log_2 p(c_i,E) \tag{5.1}$$

donde k es el número de clases, c_i son cada una de las clases, y $p(c_i,E)$ es la proporción de ejemplos en E que pertenecen a la clase c_i.

Como estamos interesados en el refinamiento de candidatos r_i 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 5.2.

$$infoGain(E, counter) = s(E) - \sum_{V \in \{true, false\}} \frac{counter(v)}{|E|} s(E_{r_i}) \tag{5.2}$$

donde *counter* es la matriz computada en el algoritmo 6 y $E_{r_i} \subseteq E$ es la partición inducida por r_i en los ejemplos E. La ganancia máxima que un candidato r_i puede obtener está dada por la ecuación 5.3.

$$maxGain(E) = -\sum_{E_{r_i} \subseteq E} \frac{|E_{r_i}|}{|E|} \log_2 \frac{|E_{r_i}|}{|E|} \tag{5.3}$$

La métrica *Gain Ratio* penaliza a los atributos que generan muchas particiones, por ejemplo atributos de tipo *ID*, y se computa por la ecuación 5.4.

$$gainRatio(E, counter) = \frac{infoGain(E, counter)}{maxGain(E)} \tag{5.4}$$

Ejemplo 10 *Supóngase que se quiere calcular el valor de Gain Ratio de* $Q_i = [intend(put, A, B), clear(B)]$ *con los ejemplos presentados en la tabla 5.9.*

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

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

```
begin(model(3)).
succ.
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)).
succ.
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)).
succ.
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 5.9 Ejemplo de modelos para un agente *singleMinded*.

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 5.1. Se busca la incertidumbre basándose en cuantos ejemplos satisfacen Q_i (6) y cuantos no (2).

$$\begin{aligned} s(E) &= -(\tfrac{6}{8}\log_2\tfrac{6}{8}) - (\tfrac{2}{8}\log_2\tfrac{2}{8}) \\ &= -(0{,}75\log_2 0{,}75) - (0{,}25\log_2 0{,}25) \\ &= -(0{,}75 \times -0{,}415037) - (0{,}25 \times -2) \\ &= -(-0{,}311277) - (-0{,}5) \\ &= 0{,}311277 + 0{,}5 \\ &= 0{,}811277 \end{aligned}$$

Después, se calcula la ganancia de información usando la ecuación 5.2, tomando como referencia el total de ejemplos de la clase succ *(5) y la clase* fail *(3).*

$$\begin{aligned} infoGain(E, counter) &= 0{,}811277 - (\tfrac{5}{8}(-(\tfrac{5}{5}\log_2\tfrac{5}{5}) - (\tfrac{0}{5}\log_2\tfrac{0}{5})) + \tfrac{3}{8}(-(\tfrac{1}{3}\log_2\tfrac{1}{3}) - (\tfrac{2}{3}\log_2\tfrac{2}{3}))) \\ &= 0{,}811277 - (\tfrac{5}{8}(0-0) + \tfrac{3}{8}(-(0{,}333333 \times -1{,}584962) - (0{,}666666 \times -0{,}584962))) \\ &= 0{,}811277 - (\tfrac{3}{8}(-(-0{,}528320) - (-0{,}389974))) \\ &= 0{,}811277 - (\tfrac{3}{8}(0{,}528320 + 0{,}389974)) \\ &= 0{,}811277 - (0{,}375 \times 0{,}918294) \\ &= 0{,}811277 - 0{,}344360 \\ &= 0{,}466917 \end{aligned}$$

La ganancia máxima se calcula usando la ecuación 5.3:

$$\begin{aligned} maxGain(E) &= -(\tfrac{6}{8}\log_2\tfrac{6}{8}) - (\tfrac{2}{8}\log_2\tfrac{2}{8}) \\ &= -(0{,}75\log_2 0{,}75) - (0{,}25\log_2 0{,}25) \\ &= -(0{,}75 \times -0{,}415037) - (0{,}25 \times -2) \\ &= -(-0{,}311277) - (-0{,}5) \\ &= 0{,}311277 + 0{,}5 \\ &= 0{,}811277 \end{aligned}$$

Por último, el coeficiente Gain Ratio está dado por la ecuación 5.4.

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

5.4.1. Generación de candidatos: Función ρ

Los candidatos Q_i que se emplean para construir el árbol, son computados combinatoriamente a través de la predisposición del lenguaje (directivas *rmode* y *lookaahead*), el cual especifica que tipo de hipótesis son válidas, en este caso, que tipo de literales o conjunciones de literales pueden ser introducidas como nodos del árbol. Se genera un conjunto de hechos del tipo *rmode(conj)*, bajo las condiciones definidas en la sección 4.3.3. Por ejemplo, supóngase la predisposición del lenguaje mostrado en el cuadro 5.10, y la consulta inicial $Q = [true]$, se generarán los siguientes candidatos:

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

```
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 5.10 Predisposición del lenguaje para el mundo de los bloques.

Estos candidatos son generados debido a que los modos de los enunciados *rmode* para los átomos *clear* y *on* (Cuadro 5.10, líneas 1-3), requieren que existan variables declaradas, las cuales son introducidas por el enunciado *rmode(intend(put,-,-))* y los enunciados *lookahead*. Supóngase ahora que el candidato que maximizó la proporción 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 átomo es *on* o *clear*. En caso de que el candidato *clear(B)*, maximice la proporción de ganancia, se puede generar un árbol como el que se muestra es la figura 5.6.

5.4.2. Abstracción de clases

Además de las acciones internas y clases de agente implementadas, JILDT contiene otras clases que permiten abstraer algunos conceptos necesarios para la ejecución de TILDE. La clase *model* (ver la figura 5.8) abstrae el concepto de un modelo (ver sección 4.3.3), el cual es un ejemplo de entrenamiento, almacenado en el archivo que contiene la base de conocimiento (*knowledgebase file*). El método estático *join*, une la base de conocimiento del modelo, con un objeto de la clase *BeliefBase* que contiene las reglas de conocimiento general *(backgroud)* -.

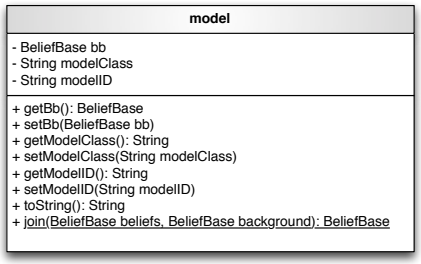


Figura 5.8 Diagrama de clase de la clase *model*.

Dado que la clase *Term* que por defecto viene en el paquete *jason.asSyntax* de la distribución de *Jason*, no permite manipular términos que inicien con un signo, por ejemplo *clear(+V1)*, se ha implementado una clase que abstrae un *rmode* de TILDE, el cual puede comenzar con {+,-}. La clase *tildeTerm* implementa las interfaces *Cloneable* y *Comparable*. El diagrama de clase de la clase *tildeTerm* se muestra en la figura 5.9.

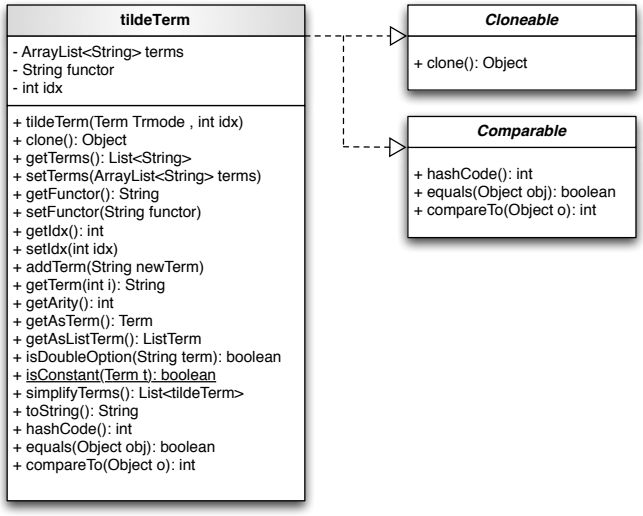


Figura 5.9 Diagrama de clase de la clase *tildeTerm*.

Para abstraer el concepto de nodo interno y hoja, se definió la interfaz *tildeTree*, de la cual se hace la implementación a través de la clase *iNode*. Por otra parte, la clase *leaf* implementa la interfaz *tildeTree*, y hereda los métodos de la clase *iNode*. El diagrama de clase se muestra en la figura 5.10.

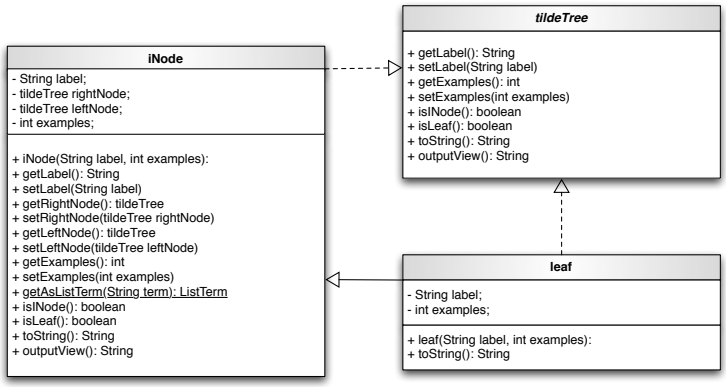


Figura 5.10 Diagrama de clase de las clases *iNode, leaf* y la interfaz *tileTree*.

Por último, se implementó una clase de interfaz de usuario, que funge como herramienta de visualización, ya que despliega una ventana que contiene información de donde se ejecuto el algoritmo de inducción, el árbol computado, y su interpretación como reglas. La figura 5.11 muestra un ejemplo de como aparece una ventana de esta clase al ser ejecutada.

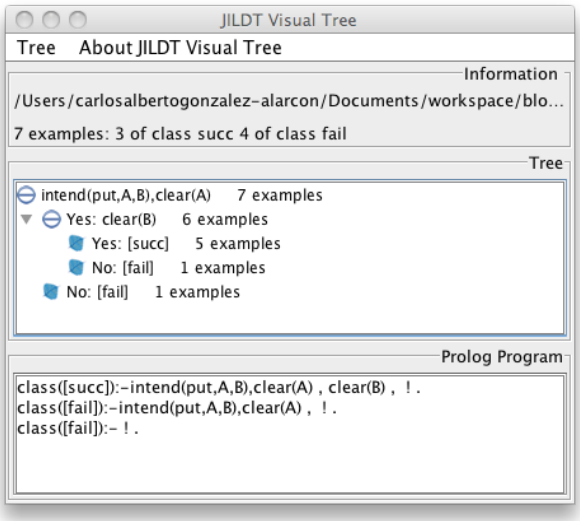


Figura 5.11 GUI desplegando un árbol lógico de decisión para el mundo de los bloques.

5.5. Resumen

La librería JILDT implementa el aprendizaje Intencional en agentes definidos bajo ciertas clases de agente. Esto es posible gracias a un mecanismo en el que los planes originales son extendidos, para poder generar ejemplos de ejecuciones, con base en el estado mental de los agentes, y poder lidiar con ejecuciones fallidas, ejecutando el proceso de aprendizaje. Las capacidades de los agentes son extendidas a través de nuevas acciones internas 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, implementado como una acción interna integrada a Jason. Dos clases de agente personalizadas fueron implementadas, extendiendo las funciones básicas de los agentes por defecto de *Jason*, para sustentar el aprendizaje Intencional. 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. En la primera sección se muestran dos experimentos ejecutados en el ya conocido **mundo de los bloques** que nos servirán para demostrar que la racionalidad de los agentes va aumentando gracias al mecanismo de aprendizaje Intencional sustentado por JILDT. En el primer experimento se utiliza JILDT para generar el conjunto de aprendizaje y ejecutar el algoritmo de inducción (TILDE) como un **comando externo**; en el segundo experimento se ejecuta TILDE como una acción interna **integrada** a Jason. Estos dos experimentos corresponden a la primera y segunda fase de este trabajo de investigación, respectivamente, de acuerdo a los objetivos planteados en la sección 1.5.

En la segunda sección se presenta un experimento realizado con el el problema de las **Tarjetas de Bongard** (Bongard, 1970), el cual fue seleccionado para comparar los árboles generados por **JILDT/TILDE** con sus equivalentes generados por **ACE/TILDE**. Tanto los experimentos de la primera sección como los de la segunda nos arrojaron los resultados esperados, sin embargo, como trabajo futuro se contempla trabajar en la mejora del desempeño de JILDT/TILDE.

Los experimentos de la sección 6.1.1 se ejecutaron en una máquina virtual con Ubuntu 9.10 y una memoria base de 1 GB. El equipo local es un equipo MacBook con Sistema Operativo MacOS X Snow Leopard 10.6.5, procesador Intel Core 2 Duo a 2.26 GHz y memoria RAM DDR3 de 4 GB a 1067 MHz. Los experimentos en la sección 6.1.2 se ejecutaron sobre el Sistema Operativo MacOS X Snow Leopard 10.6.5 en una máquina Mac Pro con procesador Intel Xeon Quad-Core a 2 x 2.66 GHz y memoria RAM DDR3 de 8 GB a 1066 MHz. El experimento de la sección 6.2 se corrió en el mismo equipo local donde se ejecutaron los experimentos de la sección 6.1.1.

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 encima de otro bloques. El objetivo es construir una o más 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).

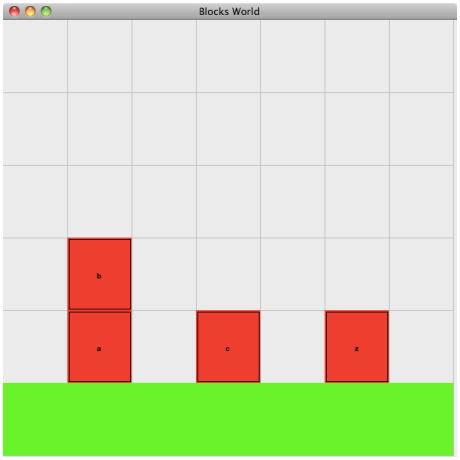


Figura 6.1 Mundo de los bloques simulado en Jason. Adaptado de Bordini et al. (2007)

6.1.1. TILDE como comando externo

Se ha diseñado este experimento para comparar el comportamiento de un agente por defecto Jason *(Default)*, un agente de la clase *intentionalLearner (learner)* que aprende las razones para adoptar un plan, y un agente de la clase *singleMindedLearner (singleMinded)* que además aprende sus políticas para abandonar intenciones.

[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, es decir, todos creen ciegamente que pueden poner bloques en cualquier lugar, y que es su única competencia. Además perciben del ambiente: $on(b,a)$, $on(a,table)$, $on(c,table)$ y $on(z,table)$.

```
// Beliefs
clear(X) :- not(on(_,X)).
clear(table).
// Perceptions of the environment
on(b,a).
on(a,table).
on(c,table).
on(z,table).
// Plans
@put
+!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** pide a los otros agentes que coloquen el bloque b sobre el bloque c, pero con cierta probabilidad, $p(N)$, éste introduce ruido en el experimento, colocando el bloque z sobre el bloque b. 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 los otros agentes poner el bloque b sobre el bloque c. Esto significa que los otros agentes pueden percibir el ruido antes o al mismo tiempo la intención de poner b en c. En el cuadro 6.2 se muestra el código del SMA en el que corre el experimento[2]. El código de los agentes se presenta en el apéndice A.

```
MAS blocksWorld {
   infrastructure: Centralised
   environment: BlocksEnv.BlocksWorld(1,100,yes)

   agents:
       experimenter;
       learner agentClass jildt.intentionalLearner;
       singleMinded agentClass jildt.singleMindedLearner;
       default;
   aslSourcePath: "src/asl";
}
```

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

[2] 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.

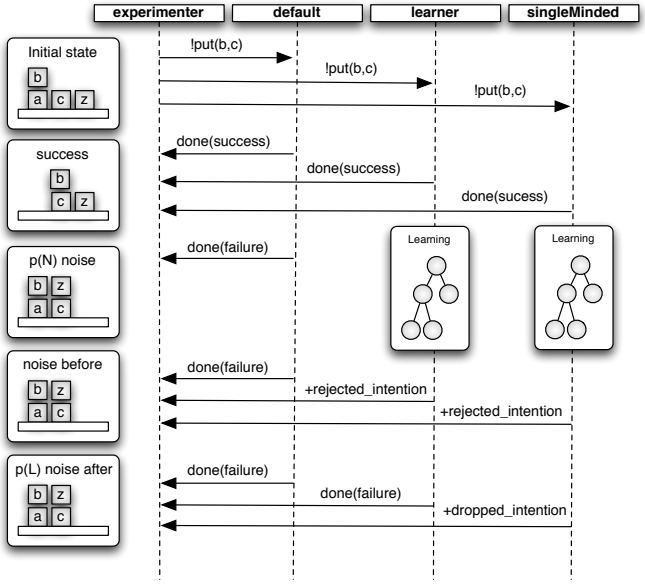


Figura 6.2 Procedimiento experimental en el mundo de los bloques.

En el cuadro 6.3 se muestra una parte de los resultados numéricos, los cuales son el promedio de 10 corridas, donde cada corrida ejecuta 100 veces el procedimiento descrito anteriormente en la figura 6.2, para una probabilidad de latencia de 50 %. La probabilidad de ruido varía y toma los valores (90 %, 70 %, 50 %, 30 %, y 10 %).

Los valores más bajos configuran entornos menos dinámicos, libre de sorpresas, lo que da como resultado un entorno efectivamente observable. 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 del plan se considera un comportamiento **irracional**.
- Negarse a formar una intención, porque el plan no es aplicable; abandonar una intención debido a que existe una razón para creer que ésta fallará; y lograr el objetivo de poner *b* en *c* se consideran comportamientos **racionales**.

La figura 6.3 resume el resultado de los experimentos ejecutados, donde las probabilidades de 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.

Agente	p(N)	Irracional			Racional			
		Después	Antes	Total	Rechazo	Abandono	Logro	Total
default	90	43.8	48.2	92.0	00.0	00.0	08.0	08.0
learner	90	48.7	37.3	86.0	04.5	00.0	09.5	14.0
singleMinded	90	44.5	38.8	83.3	03.2	03.8	09.7	16.7
default	70	34.5	36.0	70.5	00.0	00.0	29.5	29.5
learner	70	33.2	13.3	46.5	20.6	00.0	32.9	53.5
singleMinded	70	18.4	16.4	34.8	16.3	17.5	31.4	65.2
default	50	22.5	26.3	48.8	00.0	00.0	51.2	51.2
learner	50	26.1	05.4	31.5	20.7	00.0	47.8	68.5
singleMinded	50	11.6	09.9	21.5	16.1	14.9	47.5	78.5
default	30	14.2	15.0	29.2	00.0	00.0	70.8	70.8
learner	30	15.1	02.4	17.5	11.8	00.0	70.7	82.5
singleMinded	30	03.3	03.7	07.0	10.9	12.0	70.1	93.0
default	10	04.2	05.5	09.7	00.0	00.0	90.3	90.3
learner	10	05.3	01.0	06.3	04.9	00.0	88.8	93.7
singleMinded	10	00.9	00.9	01.8	03.8	03.4	91.0	98.2

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

Figura 6.3 Resultados experimentales de desempeño. Izquierda: Agente *Default*. Centro: Agente *Learner*. Derecha: Agente *SingleMinded*.

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, el bloque Y debe estar libre.

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

De esta manera, el agente *learner* puede rechazar la intención de poner *b* en *c* si percibe que *c* no está libre. 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, el agente *singleMinded* aprende la siguiente regla para abandonar la intención cuando el bloque *Y* no está libre[3].

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

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 *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.

Para altas probabilidades de ruido y la latencia, la posibilidad de recoger ejemplos de entrenamiento contradictorios aumenta, y decae el rendimiento de los agentes *learner* y *singleMinded*. Por ejemplos contradictorios queremos decir que para la misma configuración de bloques, los ejemplos pueden ser etiquetados como satisfactorios *(succ)*, pero también como fallidos *(fail)*. Esto sucede porque los ejemplos de entrenamiento se basan en las creencias del agente cuando el plan fue adoptado como una intención, de modo que la aparición posterior de ruido no está incluida.

6.1.2. TILDE como acción interna de Jason

Como se mencionó anteriormente, este experimento corresponde a la segunda fase de este trabajo de investigación, en la que se implementa el algoritmo de inducción TILDE como una acción interna de Jason. El procedimiento a seguir es el mismo que el que se muestra en la figura 6.2. Dado que el motivo de este experimento es probar el uso de JILDT/TILDE, la interacción con el agente *default* es innecesaria, debido a que no emplea el mecanismo de aprendizaje, y su comportamiento es el mismo.

[3] Dado que estos fueron los primeros resultados, la regla de abandono se formaba con el predicado *intending/2*, más adelante se cambió, usando la primitiva de Jason *.intend/1* .

La figura 6.4 muestra el desempeño del agente *learner* ejecutando el algoritmo de inducción a través del sistema ACE/TILDE (izquierda), y como una acción interna de Jason (derecha). Como se puede apreciar, el comportamiento es muy similar, variando la racionalidad debido a la naturaleza aleatoria del experimento, por lo que las conclusiones son las mismas que en la sección anterior.

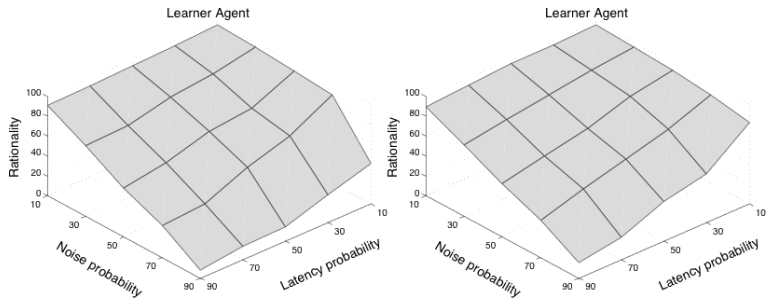


Figura 6.4 Resultados experimentales: ACE/TILDE vs JILDT/TILDE. Izquierda: Agente *Learner* ejecutado ACE/TILDE. Derecha: Agente *Learner* ejecutando JILDT/TILDE.

De igual modo, el agente *singleMinded* se comporta muy similar ejecutando la inducción a través de ACE/-TILDE o como una acción interna de Jason. La figura 6.5 muestra estos resultados. Aunque las gráficas no son precisamente muy parecidas, la conclusión es la misma, es decir, se muestra un comportamiento más racional que los otros dos, sin embargo, con altas probabilidades de ruido y la latencia, sigue recogiendo ejemplos de entrenamiento contradictorios, reduciendo su rendimiento.

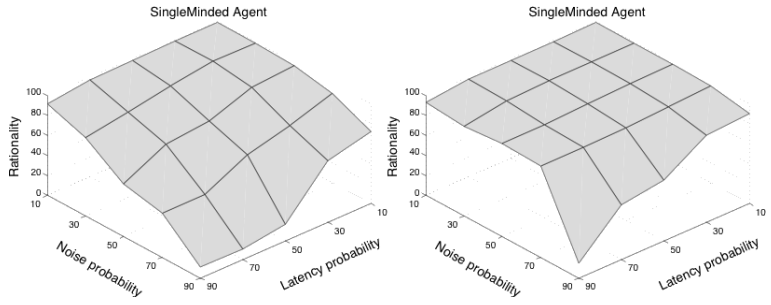


Figura 6.5 Resultados experimentales: ACE/TILDE vs JILDT/TILDE. Izquierda: Agente *SingleMinded* ejecutado ACE/TILDE. Derecha: Agente *SingleMinded* ejecutando JILDT/TILDE.

Como se mencionó en la sección anterior, con cierta probabilidad de ruido $P(N)$ el agente experimentador coloca el bloque z sobre el bloque c, y permite a los agentes aprender que para poner un bloque sobre otro, éste último debe estar libre. Sin embargo, en la práctica un agente debería saber que ambos bloques deben estar libres, por lo que se agrega una nueva nueva probabilidad de ruido al experimento para aprender esta condición. Con cierta probabilidad $P(T)$ el agente experimentador puede mover el bloque z sobre el bloque c o sobre el bloque b (ver la figura 6.6).

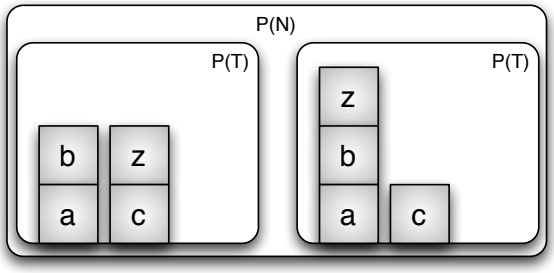


Figura 6.6 Probabilidad de Tipo de Ruido $P(T)$.

En este caso, los agentes *learner* y *singleMinded* son capaces de redefinir el contexto de su plan *put(X,Y)* de la siguiente manera:

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

Además, el agente *singleMinded* aprende las siguientes reglas de abandono:

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

El desempeño de los agentes *learner* y *singleMinded* usando esta última configuración se resume en la figura 6.7.

En la figura 6.8 se muestra el desempeño del agente *singleMinded* corriendo en las tres configuraciones del experimento[4]. Aunque no son muy parecidas las gráficas, el comportamiento es similar, teniendo siempre conflictos de racionalidad con valores altos probabilidad de ruido y latencia.

[4] TILDE como comando externo aprendiendo $clear(Y)$, TILDE como acción interna aprendiendo $clear(Y)$ y TILDE como acción interna aprendiendo $(clear(X) \& clear(Y))$

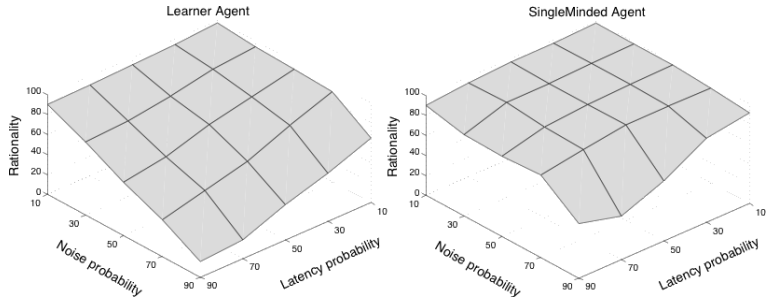


Figura 6.7 Resultados experimentales. Aprendiendo (clear(X) & clear(Y)). Izquierda: Agente *Learner*. Derecha: Agente *singleMinded*.

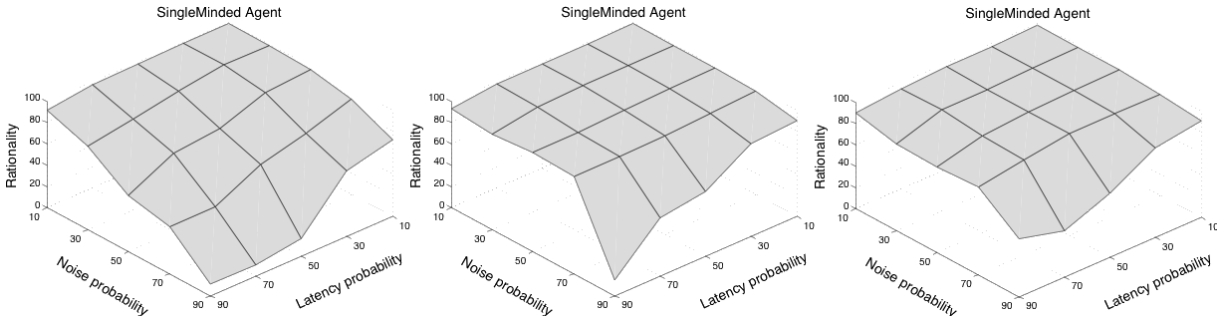


Figura 6.8 Resultados experimentales. Comportamiento del agente *singleMinded*.

Las tablas 6.4 y 6.5 muestran parte de la salida del experimento con los agentes *experimenter* y *singleMinded*, aprendiendo un nuevo contexto, y enviando mensajes de abandono o rechazo.

```
[experimenter] ====================================================
[experimenter] Experiment Number: 10/100
[experimenter] Noise : on(z,c)
[experimenter] Latency: Before
[experimenter] -------------------------
[singleMinded] current context for plan: put is true
[singleMinded] Trying to learn a better context...
[singleMinded] /Users/carlosalbertogonzalez-alarcon/Desktop/singleMinded/singleMinded_1/singleMinded/exp7
[singleMinded] Tilde was executed successfully...
[singleMinded] Learned context for put is [clear(X),clear(Y)]
[singleMinded] Changing context in plan put ...
[singleMinded] Context was changed succesfully...
[experimenter] Agent singleMinded has failed
```

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

```
[experimenter] ====================================================
[experimenter] Experiment Number: 11/100
[experimenter] Experiment without noise
[experimenter] --------------------------
[singleMinded] current context for plan: put is (clear(X) & clear(Y))
[singleMinded] Yeah, I did the task successfully
[experimenter] Agent singleMinded has succeed.
[experimenter] ====================================================
[experimenter] Experiment Number: 12/100
[experimenter] Noise : on(z,c)
[experimenter] Latency: After
[experimenter] --------------------------
[singleMinded] current context for plan: put is (clear(X) & clear(Y))
[singleMinded] Wow!! I'm sorry, I have to abandon my intention
[experimenter] Agent singleMinded has failed because it abandoned the intention
[experimenter] ====================================================
[experimenter] Experiment Number: 13/100
[experimenter] Noise : on(z,b)
[experimenter] Latency: Before
[experimenter] --------------------------
[singleMinded] Plan put non applicable.
[experimenter] Agent singleMinded has failed because it didn't take the intention
```

Cuadro 6.5 Salida de la ejecución del experimento con el agente *singleMinded (Continuación)*.

6.2. Tarjetas de Bongard

Como se mencionó en la introducción de este capítulo, usaremos el ejemplo de las tarjetas de Bongard para comparar un árbol generado por JILDT/TILDE con su equivalente generado por ACE/TILDE. Las tarjetas de Bongard consisten en un conjunto de imágenes, donde cada una de ellas está etiquetada como $\oplus$ ó $\ominus$. La tarea consiste en clasificar nuevas imágenes en una de estas clases observando los objetos dentro de éstas. A este tipo de problemas se les conoce como problemas de Bongard, después de que Mikhail Bongard los utilizara para las pruebas de reconocimiento de patrones (Bongard, 1970). La figura 6.9 muestra un ejemplo de las tarjetas de Bongard, donde cada clase tiene cuatro ejemplos.

Considerando únicamente la forma del objeto, la configuración de la dirección en que apuntan los triángulos, y la posición relativa, las imágenes de la figura se puede representar de la siguiente manera:

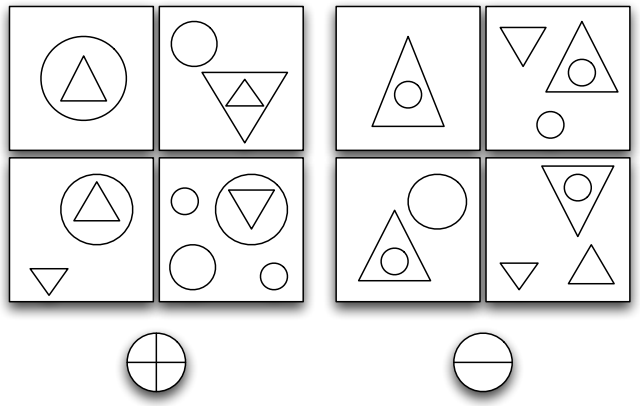


Figura 6.9 Tarjetas de Bongard. (Bongard, 1970)

Imagen 1 {circle(o1), triangle(o2), config(o2, up), in(o2, o1)}
Imagen 2 {circle(o3), triangle(o4), config(o4, up), triangle(o5), config(o5, down), in(o4, o5)}
Imagen 3 {triangle(o6), circle(o7), triangle(o8), config(o8,up), in(o8,o7)}
etc.

El archivo que contiene los modelos (bongard.kb) tiene un total de 392 ejemplos. El archivo de configuración (bongard.s) se muestra en el cuadro 6.6. No se contempla el uso de conocimiento general (archivo *.bg).

```
load(models).
talking(4).
output_options([c45,prolog,lp]).
classes([pos,neg]).

rmode(triangle(+-S)).
rmode(square(+-S)).
rmode(circle(+-S)).
rmode(in(+S1,+-S2)).
rmode(config(+S,up)).
rmode(config(+S,down)).
```

Cuadro 6.6 Archivo de configuración *bongard.s*.

El árbol resultante de ejecutar ACE/TILDE se muestra en el cuadro 6.8, mientras que el árbol resultante de ejecutar JILDT/TILDE se muestra en el cuadro 6.7. Se puede apreciar una gran semejanza entre ambos árboles, donde sólo cambian algunas variables no declaradas anteriormente, que no afectan el funcionamiento al momento

de clasificar, dado que pueden tomar cualquier cualquier valor. Por ejemplo las ramas $in(C,-D)$ e $in(C,F)$. Por último, en la figura 6.9 se muestra la interfaz gráfica de usuario desplegando el árbol generado por JILDT/TILDE.

```
triangle(-A) ?
+--yes: in(A,-B) ?
|       +--yes: triangle(B) ?
|       |       +--yes: [pos] 82.0 [[pos:82.0,neg:0.0]]
|       |       +--no:  circle(-C) ?
|       |               +--yes: in(C,-D) ?
|       |               |       +--yes: [neg] 28.0 [[pos:0.0,neg:28.0]]
|       |               |       +--no:  [pos] 34.0 [[pos:34.0,neg:0.0]]
|       |               +--no:  [neg] 36.0 [[pos:0.0,neg:36.0]]
|       +--no:  circle(-E) ?
|               +--yes: in(E,-F) ?
|               |       +--yes: [neg] 79.0 [[pos:0.0,neg:79.0]]
|               |       +--no:  [pos] 12.0 [[pos:12.0,neg:0.0]]
|               +--no:  [neg] 34.0 [[pos:0.0,neg:34.0]]
+--no:  [neg] 87.0 [[pos:0.0,neg:87.0]]
```

Cuadro 6.7 Bongard: Árbol resultante de ejecutar ACE/TILDE.

```
triangle(A)
+ yes:  in(A,B)
 |     + yes:  triangle(B)
 |      |     + yes:  [pos]   82
 |      |     + no:  circle(C)
 |      |      |     + yes:  in(C,F)
 |      |      |      |     + yes:  [neg]   28
 |      |      |      |     + no:  [pos]   34
 |      |      |     + no:  [neg]   36
 |     + no:  circle(D)
 |      |     + yes:  in(D,E)
 |      |      |     + yes:  [neg]   79
 |      |      |     + no:  [pos]   12
 |      |     + no:  [neg]   34
+ no:  [neg]   87
```

Cuadro 6.8 Bongard: Árbol resultante de ejecutar JILDT/TILDE.

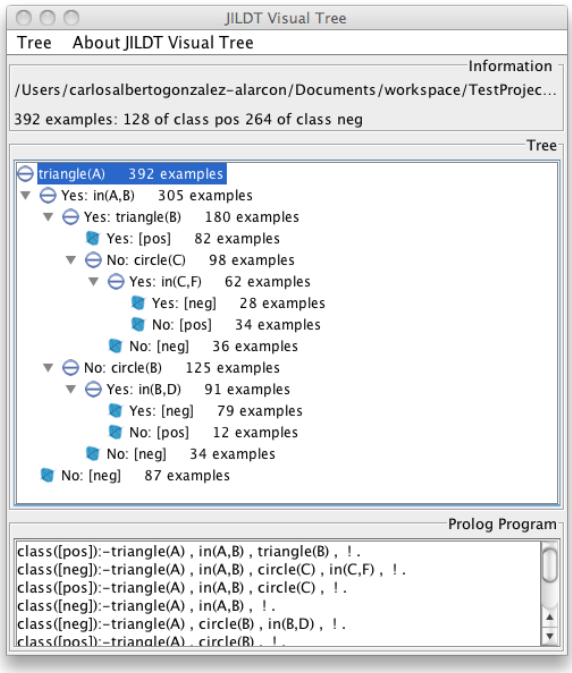


Figura 6.10 Interfaz Gráfica de Usuario para la tarjetas de Bongard.

6.3. 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 un incremento de racionalidad desde el agente *default* hacia el agente *singleMinded*, como se describió en la página 84, figura 5.5, mostrando un nivel de adaptabilidad al ambiente.

El segundo y tercer experimento nos permiten demostrar como el algoritmo de inducción JILDT/TILDE funciona de manera muy similar a la implementación ACE/TILDE, con algunas diferencias. En términos de eficiencia, la ejecución del algoritmo como una acción interna de Jason resulta ser más costosa computacionalmente, debido a la complejidad del lenguaje de implementación *(Java)*, y el proceso de generación de candidatos. Sin embargo, con pocos ejemplos, como en el caso del mundo de los bloques, los tiempos de ejecución son muy similares.

Una diferencia más con ACE/TILDE, es que JILDT/TILDE no implementa un algoritmo de post-poda, dado que no fue necesario en estos experimentos, sin embargo, eventualmente se pretende implementar este mecanismo.

Capítulo 7
Conclusiones

La librería JILDT proporciona a *AgentSpeak(L)* las extensiones necesarias para definir agentes que aprenden Intencionalmente, esto gracias a las bondades que nos ofrece el modelo BDI de agentes. El estado mental de los agentes proporciona los ejemplos de entrenamiento para inducir el aprendizaje, y el contexto de sus planes puede modificarse mediante la inducción de arboles lógicos de decisión, debido a la naturaleza conjuntiva de las ramas de estos árboles lógicos de decisión.

Usando JILDT, se implementó una clase de agente con compromiso racional *(singleMindedLearner)*, la cual modifica su sistema de transición para habilitar el abandono de intenciones. Básicamente, cada vez que el sistema está en en el paso *execInt* y se dispara una regla de abandono, se elimina la intención en vez de ser ejecutada. 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.

Los resultados experimentales son muy prometedores. En comparación con otros experimentos sobre compromiso de (Kinny, 1991), se observa que agentes definidos como instancias de las clases *intentionalLearner* 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. Parece ser que los agentes adoptan una actitud confiada hacia los planes de éxito, y una actitud cautelosa hacia los planes que fallan, pero son necesarios más experimentos para confirmar esta hipótesis.

Se obtuvo una mejor comprensión del método inductivo, el cual puede ser mejorado. Por ejemplo, es posible utilizar la consulta inicial de TILDE para evitar el uso de *lookaheads*, reduciendo el número de candidatos, durante el cómputo del árbol de decisión lógico.

Los resultados experimentales sugieren que la inducción se podría mejorar si los ejemplos de entrenamiento no sólo representan las creencias del agente cuando éste adoptó la intención, sino también cuando se ha finalizado o se ha abandonado. Lo anterior con el fin de minimizar los efectos de latencia en el ruido.

7.1. Trabajos relacionados

Subagdja et al. (2009) propone una arquitectura para aprendizaje Intencional. Su uso del término aprendizaje Intencional es ligeramente diferente, lo que significa que el aprendizaje era el objetivo de los agentes BDI en lugar de un resultado imprevisto. 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. 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) presenta 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, 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.

7.2. Trabajos futuros

Entre los trabajos futuros de esta investigación, se busca mejorar el desempeño del algoritmo. Una forma de hacerlo, es introduciendo la intención actual del agente como consulta inicial al momento de empezar a construir el árbol lógico de decisión. De este modo, las variables de las literales son introducidas, y se elimina el cómputo de los candidatos con directiva *lookahead*, que tenían como principal objetivo introducir estas variables. Por ejemplo, originalmente la consulta inicial al construir el árbol es $[true]$, y genera los siguientes candidatos:

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

Introduciendo la intención actual del agente como consulta inicial generaría los siguientes candidatos:

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

De este modo, se tiene una cantidad reducida de candidatos. Otra mejora de desempeño 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.

Un problema que se pudo observar en las gráficas experimentales en la sección 6.1, es que el desempeño de los agentes *intentionalLearner* y *singleMindeLearner* decae cuando se tienen probabilidades altas de latencia. Como trabajo futuro, se pretende extender la representación de estos ejemplos, incluyendo ejemplos de entrenamiento que representen las creencias del agente cuando una intención ha finalizado o ha sido rechazada o abandonada, además de los ejemplos de entrenamiento que representan las creencias del agente cuando se adoptó la intención.

En el capítulo 4 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).

Por último, se busca que JILDT aumente el nivel de aprendizaje con base en su nivel de conocimiento social en un entorno multiagente, como se describe en la página 55. Esto, nos lleva a un aprendizaje distribuido, colectivo y social. Los protocolos de aprendizaje están previamente formalizados en (Guerra-Hernández et al., 2004a).

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.

Bongard, M. (1970). *Pattern Recognition*. Spartan Books.

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.

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. (2010). 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., 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, 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.

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.

Parte III

Apéndice

Apéndice A
Código de los agentes del mundo de los bloques

A.1. Agente *default*

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

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

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

@put_fail
-!put(X,Y) : true <-
     .send(experimenter,tell,experiment(done)).
```

A.2. Agente *learner*

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

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

```

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

@example
+example_processed(put) : true <-
     -example_processed(put);
     .send(experimenter,tell,experiment(done)).

@reject
+non_applicable(put) : true <-
     -non_applicable(put);
     .send(experimenter,tell,[non_applicable(put),experiment(done)]).
```

A.3. Agente *singleMinded*

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

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

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

@example
+example_processed(put) : true <-
     -example_processed(put);
     .send(experimenter,tell,experiment(done)).

@reject
+non_applicable(put) : true <-
     -non_applicable(put);
     .send(experimenter,tell,[non_applicable(put),experiment(done)]).

@dropped
+dropped_int(put(X,Y)) : true <-
     -dropped_int(put(X,Y));
     .send(experimenter,tell,[dropped_int(put(X,Y)),experiment(done)]).
```

A.4. Agente *experimenter*

```
// Agent experimenter in project BlocksWorld.mas2j

/* Initial beliefs and rules */
agentExp(singleMinded).
noExperiments(1).
noiseConf(0.5,0.3,0.5). // (P(N), P(L), P(T))

/* Initial goals */
!run_all_experiments.

/* Plans */
+!run_all_experiments : noExperiments(101) <- true.

+!run_all_experiments : noExperiments(NE)[source(_)] <-
     .print("==================================================");
     .print("Experiment Number: ",NE, "/100");
     ?agentExp(Ag);
     !run_one_experiment(Ag);
     .wait("+experiment(done)[source(Ag)]");
     NewNE = NE + 1;
     -+noExperiments(NewNE);
     !result(Ag);
     -experiment(done)[source(_)];
     !run_all_experiments.

+!run_one_experiment(Ag) : .random(R) & noiseConf(NP,_,_)[source(_)] & R < NP  <-
     !move_block(A,B);
     .print("Noise : on(",A,",",B,")");
     !noisy_movement(A,B,Ag).

+!run_one_experiment(Ag) : true <-
     .print("Experiment without noise");
     .print("-------------------------");
     .send(Ag,achieve,put(b,c)).

+!move_block(X,Y) : .random(R) & noiseConf(_,_,PM)[source(_)] & R < PM <-
     X = z;
     Y = c.

+!move_block(X,Y) : true <-
     X = z;
     Y = b.

+!noisy_movement(A,B,Ag) : .random(R) & noiseConf(_,NS,_)[source(_)] & R < NS <-
     .print("Latency: After");
     .print("-------------------------");
     .send(Ag,achieve,put(b,c));
     move(A,B).
```

```
49
50  +!noisy_movement(A,B,Ag) : true <-
51       .print("Latency: Before");
52       .print("--------------------------");
53       move(A,B);
54       .send(Ag,achieve,put(b,c)).
55
56  +!result(Ag) : on(b,c)  <-
57       .print("Normal has succeed.");
58       move(b,a).
59
60  +!result(Ag) : non_applicable(put) <-
61       -non_applicable(put)[source(Ag)];
62       .print("Normal has failed because it didn't take the intention");
63       move(z,table).
64
65  +!result(Ag) : dropped_int(put(X,Y)) <-
66       -dropped_int(put(X,Y))[source(Ag)];
67       .print("Normal has failed because it abandoned the intention");
68       move(z,table).
69
70  +!result(Ag) : true <-
71       .print("Normal has failed");
72       move(z,table).
```

Apéndice B
Publicaciones

Se publicaron dos articulos relacionados con este trabajo de investigación, los cuales corresponden a las fases I y II como se menciona en la sección 1.5.

Los primeros resultados (Guerra-Hernández et al., 2010), corresponden al uso de TILDE como un comando de inducción **externo**. Estos resultados fueron publicados dentro del *9th Mexican International Conference of Artificial Intelligence*, celebrado en la ciudad de Pachuca de Soto, Hidalgo del dia 9 al 12 de Noviembre de 2010, y siendo sede la Univerdidad Autónoma del Estado de Hidalgo.

Los resultados obtenidos de la segunda fase de esta investigación han sido aceptados para publicarse dentro del *9th European Workshop on Multi-Agent Systems*, a celebrarse en la ciudad de París, Francia los días 16 y 17 de Diciembre de 2010, siendo sede la Universidad Descartes de París.

Jason Induction of Logical Decision Trees: A Learning Library and Its Application to Commitment

Alejandro Guerra-Hernández[1], Carlos Alberto González-Alarcón[1], and Amal El Fallah Seghrouchni[2]

[1] Departamento de Inteligencia Artificial
Universidad Veracruzana
Facultad de Física e Inteligencia Artificial
Sebastián Camacho No. 5, Xalapa, Ver., México, 91000
aguerra@uv.mx, dn_carlos@hotmail.com
[2] Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
4, Place Jussieu, Paris, France, 75005
Amal.Elfallah@lip6.fr

Abstract. This paper presents JILDT (Jason Induction of Logical Decision Trees), a library that defines two learning agent classes for Jason, the well known java-based implementation of *AgentSpeak*(*L*). Agents defined as instances of JILDT can learn about their reasons to adopt intentions performing first-order induction of decision trees. A set of plans and actions are defined in the library for collecting training examples of executed intentions, labeling them as succeeded or failed executions, computing the target language for the induction, and using the induced trees to modify accordingly the plans of the learning agents. The library is tested studying commitment: A simple problem in a world of blocks is used to compare the behavior of a default Jason agent that does not reconsider his intentions, unless they fail; a learning agent that reconsiders when to adopt intentions by experience; and a single-minded agent that also drops intentions when this is rational. Results are very promissory for both, justifying a formal theory of single-mind commitment based on learning, as well as enhancing the adopted inductive process.

Keywords: Multi-Agent Systems, Intentional Learning, Commitment, AgentSpeak(L).

1 Introduction

It is well known that the the Belief-Desire-Intention (BDI) model of agency [9,10] lacks of learning competences. Intending to cope with this problem, this paper introduces JILDT (Jason Induction of Logical Decision Trees): A library that defines two learning agent classes for Jason [3], the well known java-based implementation of the *AgentSpeak*(*L*) BDI model [11]. Agents defined as instances

G. Sidorov et al. (Eds.): MICAI 2010, Part I, LNAI 6437, pp. 374–385, 2010.

of the JILDT *intentionalLearner* class can learn about their reasons to adopt intentions, performing first-order induction of logical decision trees [1]. A set of plans and actions are defined in the library for collecting training examples of executed intentions, labeling them as succeeded or failed executions, computing the target language for the induction, and using the induced trees to modify accordingly the plans of the learning agents. In this way, the intentional learning approach [5] can be applied to any Jason agent by declaring the membership to this class.

The second class of agents defined in JILDT deals with single-mind commitment [9], i.e., an agent is single-mind committed if once he intends something, he maintains his intention until he believes it has been accomplished or he believes it is not possible to eventually accomplish it anymore. It is known that Jason agents are not single-minded by default [3,6]. So, agents defined as instances of the JILDT *singleMinded* class achieve single-mind commitment, performing a policy-based reconsideration, where policies are rules for dropping intentions learned by the agents. This is foundational and theoretical relevant, since the approach reconciles policy-based reconsideration, as defined in the theory of practical reasoning [4], with computational notions of commitment as the single-mind case [9]. Attending in this way the normative and descriptive aspects of reconsideration, opens the door for a formal theory of reconsideration in $AgentSpeak(L)$ based on intentional learning.

Organization of the paper is as follows: Section 2 offers a brief introduction to the $AgentSpeak(L)$ agent oriented programming language, as defined in Jason. An agent program, used in the rest of the paper, is introduced to exemplify the reasoning cycle of Jason agents. Section 3 introduces the Top-Down Induction of Logical Decision Trees (Tilde) method, emphasizing the way Jason agents can use it for learning. Section 4 describes the implementation of the JILDT library. Section 5 presents the experimental results for three agents in the blocks world: a default Jason agent, an intentional learner and a single-mind committed agent. Section 6 offers discussion, including related and future work.

2 Jason and AgentSpeak(L)

Jason [3] is a well known java-based implementation of the $AgentSpeak(L)$ [11] abstract language for BDI agents. As usual an agent ag is formed by a set of plans ps and beliefs bs. Each belief $b_i \in bs$ is a ground first-order term. Each plan $p \in ps$ has the form $trigger\ event : context \leftarrow body$. A trigger event can be any update (addition or deletion) of beliefs (at) or goals (g).The context of a plan is an atom, a negation of an atom or a conjunction of them. A non empty plan body is a sequence of actions (a), goals, or belief updates. $\top$ denotes empty elements, e.g., plan bodies, contexts, intentions. Atoms (at) can be labelled with sources. Two kinds of goals are defined, achieve goals (!) and test goals (?).

The operational semantics [3] of the language, is given by a set of rules that define a transition system (see figure 1) between configurations $\langle ag, C, M, T, s\rangle$, where:

- ag is an agent program formed by a set of beliefs bs and plans ps.
- An agent circumstance C is a tuple $\langle I, E, A\rangle$, where: I is a set of intentions; E is a set of events; and A is a set of actions to be performed in the environment.
- M is a set of input/output mailboxes for communication.
- T stores the current applicable plans, relevant plans, intention, etc.
- s labels the current step in the reasoning cycle of the agent.

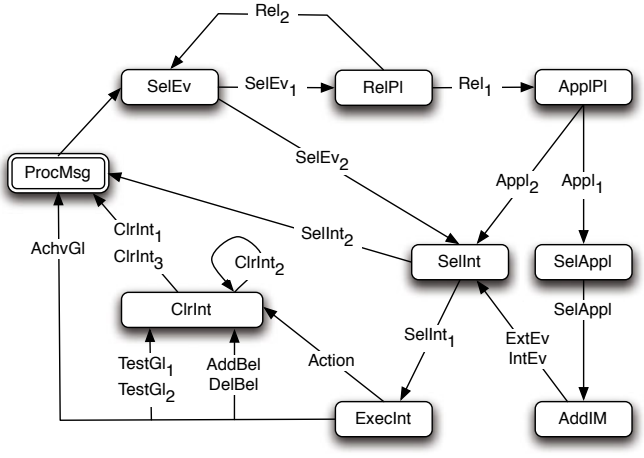


Fig. 1. The transition system for AgentSpeak(L) operational semantics

An artificially simplified agent program for the blocks world environment, included in the distribution of Jason, is listed in the table 1. Examples in the rest of this paper are based on this agent program. Initially he believes that the *table* is clear (line 3) and that something with nothing on is clear too (line 2). He has a plan labeled *put* (line 10) expressing that to achieve putting a block X on Y, in any context ($true$), he must move X to Y. Our agent is bold about putting things somewhere else. Now suppose the agent starts running in his environment, where someone else asks him to put b on c. A reasoning cycle of the agent in the transition system of Jason is as follows: at the configuration $procMsg$ the beliefs about $on/2$ are perceived (lines 5–8) reflecting the state of the environment; and an event $+!put(b, c)$ is pushed on C_E. Then this event is selected at configuration $SelEv$ and the plan *put* is selected as relevant at configuration $RelPl$. Since the context of *put* is true, it is always applicable and it will be selected to form a new intention in C_I at $AddIM$. Once selected for execution at $SelInt$, the action $move(b, c)$ will be actually executed at $ExecInt$ and since there is nothing else to be done, the intention is dropped from C_I at $ClrInt$. Coming back to $ProcMsg$ results in the agent believing $on(b, c)$ instead of $on(b, a)$.

Table 1. A simplified agent in the blocks world

```
1  // Beliefs
2  clear(X) :- not(on(_,X)).
3  clear(table).
4  // Beliefs perceived
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).
```

Now, what if something goes wrong? For instance, if another agent puts the block z on c before our agent achieves his goal? Well, his intention will fail. And it will fail every time this happens. The following section introduces the induction of logical decision trees, and the way they can be used to learn things like *put* is applicable only when Y is clear.

3 Tilde

Top-down Induction of Logical Decision Trees (Tilde) [1] has been used for learning in the context of Intentional BDI agents [5], mainly because the inputs required for this method are easily obtained from the mental state of such agents; and the obtained hypothesis are useful for updating the plans and beliefs of the agents, i.e., these trees can be used to express hypotheses about the successful or failed executions of the intentions, as illustrated in figure 2. This section introduces Tilde emphasizing this compatibility with the agents in Jason.

A Logical Decision Tree is a binary first-order decision tree where:

- Each node is a conjunction of first-order literals; and
- The nodes can share variables, but a variable introduced in a node can only occur in the left branch below that node (where it is true).

Three inputs are required to compute a logical decision tree: First, a set of training examples known as models, where each trainning example is composed by the set of beliefs the agent had when the intention was adopted; a literal coding what is intended; and a label indicating a successful or failed execution of the intention. Models are computed every time the agent believes an intention has been achieved (success) or dropped (failure). Table 2 shows two models corresponding to the examples in figure 2. The class of the examples is introduced at line 2, and the associated intention at line 3. The rest of the model corresponds to the beliefs of the agent when he adopted the intention.

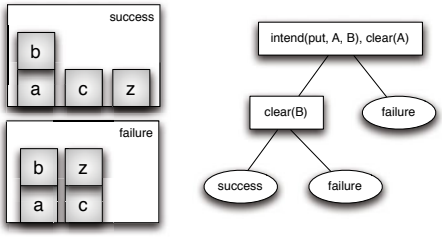


Fig. 2. A Tilde simplified setting: two training examples and the induced tree, when intending to put b on c

Table 2. The training examples from figure 2 as models for Tilde. Labels at line 2.

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

Second, the rules believed by the agent, like *clear*/1 in table 1 (lines 2–3), do not form part of the training examples, since they constitute the background knowledge of the agent, i.e., general knowledge about the domain of experience of the agent.

And third, the language bias, i.e., the definition of which literals are to be considered as candidates to be included in the logical decision tree, is defined combinatorially after the literals used in the agent program, as shown in table 3. The *rmode* directives indicate that their argument should be considered as a candidate to form part of the tree. The *lookahead* directives indicate that the conjunction in their argument should be considered as a candidate too. The last construction is very important since it links logically the variables in the intended plan with the variables in the candidate literals, enabling generalization.

For the considered example, the induced decision tree for two successful examples and one failed is showed in table 4. Roughly, it is interpreted as: when intending to put a block A on B, the intention succeeds if B is clear (line 2), and fails otherwise (line 3). With more examples, it is expected to build the tree equivalent to the one shown in figure 2.

Induction is computed recursively as in ID3. A set of candidates is computed after the language bias, and the one that maximizes information gain is selected as the root of the tree. The process finishes when a stop criteria is reached. Details about upgrading ID3 to Tilde, can be found in [2].

Table 3. The language bias defining the vocabulary to build the decision tree

```
rmode(clear(V1)). rmode(on(V1,V2)). rmode(on(V2,V1)).
rmode(intend(put,V1,V2)).
lookahead(intend(put,V1,V2),clear(V1)).
lookahead(intend(put,V1,V2),clear(V2)).
lookahead(intend(put,V1,V2),on(V1,V2)).
lookahead(intend(put,V1,V2),on(V2,V1)).
```

Table 4. The induced Logical Decision Tree

```
intend(put,A,B),clear(B) ?
+--yes: [succ] 1.0 [[succ:1.0,fail:0.0]]
+--no: [fail] 1.0 [[succ:0.0,fail:1.0]]
```

4 Implementation

JILDT implements two classes of agents: The first one is the *intentionaLearner* class, that implements agents capable of redefining the context of their plans accordingly to the induced decision trees. In this way, the reasons to adopt a plan that has failed, as an intention in future deliberations, are reconsidered. The second one is the *singleMindedLearner* class, that implements agents that are also capable of learning rules that express when it is rational to drop an intention. The body of these rules is obtained from the branches in the induced decision trees that lead to failure. For this, the library defines a set of plans to allow the agents to autonomously perform inductive experiments, as described in section 3, and to exploit their discoveries. The table 5 lists the main actions implemented in java to be used in the plans of the library. The rest of the section describes the use of these plans by a learning agent.

Table 5. Principal actions defined in the JILDT library

Action	Description
`getCurrentBels(Bs)`	Bs unifies with the list of current beliefs of the agent.
`getCurrentCtxt(C)`	C unifies with the context of the current plan.
`getCurrentInt(I)`	I unifies with the current intention.
`getLearnedCtxt(P,LC,F)`	LC unifies with the learned context for plan P. F is true if a new different context has been learned.
`changeCtxt(P,LC)`	Changes the context of plan P for LC.
`setTilde(P)`	Builds the input files for learning about plan P.
`execTilde`	Executes Tilde saving inputs and results.
`addDropRule(LC,P)`	Adds the rule to drop plan P accordingly to LC.
`setLearningMode`	Modifies plans to enable learning (intentionalLearner).
`setSMLearningMode`	Modifies plans to enable learning and dropping rules (singleMindedLearner class).

Both classes of agents define a plan `@initialLearningGoal` to set the correct learning mode (intentional or singleMinded) by extending the user defined plans to deal with the learning process. For example, such extensions applied to the plan *put*, as defined for the agent listed in table 1, are shown in the table 6. The original body of the plan is at line 6. If this plan is adopted as an intention and correctly executed, then the agent believes (line 8) a new *succ*essful training *example* about *put*, including his beliefs at the time the plan was adopted.

Table 6. JILDT extensions for plan *put* (original body at line 6)

```
1  @[put]
2  +!put(X,Y) : true <-
3    jildt.getCurrentInt(I);
4    jildt.getCurrentBels(Bs);
5    +intending(I,Bs);
6    move(X,Y);
7    -intending(I,Bs);
8    +example(I,Bs,succ);
```

Fun starts when facing problems: First, if the execution of an intention fails, for instance, because *move* could not be executed correctly, an alternative added plan, as the one showed in table 7, responds to failure event $-!put(X, Y)$. The result is a $fail$ure training example added to the beliefs of the agent (line 4) and an inductive process intended to be achieved (line 5).

Table 7. A plan added by JILDT to deal with *put* failures requiring induction

```
1  @[put_failCase]
2  -!put(X,Y) : intending(put(X,Y), Bs) <-
3    -intending(I,Bs);
4    +example(I,Bs,fail);
5    !learning(put);
6    +example_processed;
```

But, if the context of the plan *put* is different from *true*, because the agent already had learned a new context, or because he was defined like that, a failure event will be produced and the inductive process should not be intended. In this case we say that plan *put* was relevant but non applicable. The plan in table 8 deals with this situation. It is rational to avoid commitment if there is no applicable plans for a given event.

Observe that there is a small ontology associated to the inductive processes. Table 9 lists the atomic formulae used with this purpose. These formulae should be treated as a set of reserved words.

Table 8. A plan added by JILDT to deal with *put* being non applicable

```
1  @[put_failCase_NoRelevant]
2  -!put(X,Y) : not intending(put(X,Y),_) <-
3      .print("Plan ",put," non applicable.");
4      +non_applicable(put).
```

Table 9. A small ontology used by JILDT

Atom	Description
`drop(I)`	I is an intention to be dropped. Head of dropping rules.
`root_path(R)`	R is the current root to Tilde experiments.
`current_path(P)`	P is the current path to Tilde experiments.
`dropped_int(I)`	The intention I has been dropped.
`example(P,Bs,Class)`	A training example for plan P, beliefs Bs and $Class$.
`intending(I, Bs)`	I is being intended yet. $Class$ is still unknown.
`non_applicable(TE)`	There were no applicable plans for the trigger event TE.

There is a plan `@learning` to build the inputs required by *Tilde* and executing it. If the agent succeeds in computing a Logical Decision Tree with the examples already collected, then he uses the tree to construct a new context for the associated plan (branches leading to success) and a set of rules for dropping the plan when it is appropriate (branches leading to failure). Two plans in the library are used to verify if something new has been learned.

5 Experiments

We have designed a very simple experiment to compare the behavior of a default Jason agent, an intentional learner, and single-minded agent that learns his policies for dropping intentions. For the sake of simplicity, these three agents are defined as shown in figure 1, i.e., they are all bold about putting blocks somewhere else; and that is their unique compentece.

The experiment runs as illustrated in figure 3: The *experimenter* asks the other agents to achieve putting the block b on c, but with certain probability $p(N)$, he introduces noise in the experiment by putting the block z on c. There is also a latency probability $p(L)$ for the last event: The *experimenter* could put block z before or after it asks the others agents to put b on c. This means that the other agents can perceive noise before or while intending to put b on c.

Numerical results are shown in table 10 (average of 10 runs, each one of 100 experiments) for a probability of latency of 50%. The probability of noise varies (90%, 70%, 50%, 30%, and 10%). Lower values configure less dynamic environments free of surprises and effectively observable. The performance of the agent is interpreted as more or less rational as follows: dropping an intention because of the occurrence of an error, is considered irrational. Refusing to form an intention because the plan is not applicable; dropping the intention because

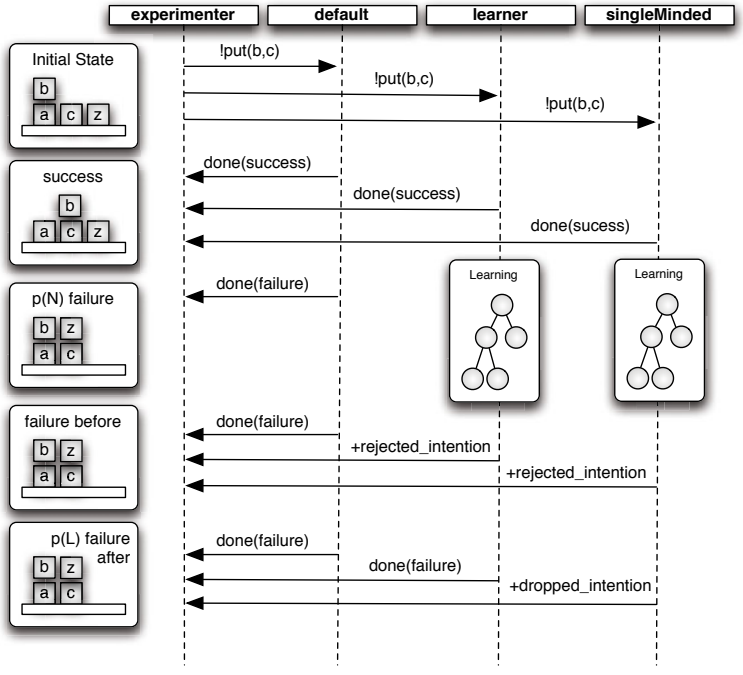


Fig. 3. The experiment process

of a reason to believe it will fail; and achieving the goal of putting b on c are considered rational behaviors.

Figure 4 summarizes the result of all the executed experiments, where the probabilities of noise and latency range on $\{90\%, 70\%, 50\%, 30\%, 10\%\}$. As expected the performance of the *default* agent is proportionally inverse to the probability of noise, independently of the probability of latency.

The *learner* agent reduces the irrationality due to noise before the adoption of the plan as intention, because eventually he learns that in order to intend to put a block X on a block Y, Y must be clear:

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

Once this has been done, the *learner* can refuse to intend putting b on c if he perceives c is not clear. So, for low latency probabilities, he performs better than the default agent, but of course his performance decays as the probability of latency increases; and, more importantly: there is nothing to do if he perceives noise after the intention has been adopted. In addition, the *singleMinded* agent learns the following rule for dropping the intention when block Y is not clear:

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

Table 10. Experimental results (average from 10 runs of 100 iterations each one) for a probablity of latency of p(L)=0.5 and different probabilities of noise p(N)

Agent	p(N)	Irrational			Rational			
		after	before	total	refuse	drop	achieve	total
default	90	43.8	48.2	92.0	00.0	00.0	08.0	08.0
learner	90	48.7	37.3	86.0	04.5	00.0	09.5	14.0
singleMinded	90	44.5	38.8	83.3	03.2	03.8	09.7	16.7
default	70	34.5	36.0	70.5	00.0	00.0	29.5	29.5
learner	70	33.2	13.3	46.5	20.6	00.0	32.9	53.5
singleMinded	70	18.4	16.4	34.8	16.3	17.5	31.4	65.2
default	50	22.5	26.3	48.8	00.0	00.0	51.2	51.2
learner	50	26.1	05.4	31.5	20.7	00.0	47.8	68.5
singleMinded	50	11.6	09.9	21.5	16.1	14.9	47.5	78.5
default	30	14.2	15.0	29.2	00.0	00.0	70.8	70.8
learner	30	15.1	02.4	17.5	11.8	00.0	70.7	82.5
singleMinded	30	03.3	03.7	07.0	10.9	12.0	70.1	93.0
default	10	04.2	05.5	09.7	00.0	00.0	90.3	90.3
learner	10	05.3	01.0	06.3	04.9	00.0	88.8	93.7
singleMinded	10	00.9	00.9	01.8	03.8	03.4	91.0	98.2

Every time a *singleMindedLearner* agent instance is going to execute an intention, first it is verified that no reasons to drop the intention exist; otherwise the intention is dropped. So, when the *singleMinded* agent already intends to put b on c and the experimenter puts the block z on c, he rationally drops his intention. In fact, the *singleMinded* agent only fails when it is ready to execute the primitive action *move* and noise appears.

For high probabilities of both noise and latency, the chances of collecting contradictory training examples increases and the performance of the *learner* and *the singleMinded* agents decay. By contradictory examples we mean that for the same blocks configuration, examples can be labeled as success, but also as failure. This happens because the examples are based on the beliefs of the agent when the plan was adopted as an intention, so that the later occurrence of noise is not included.

In normal situations, an agent is expected to have different relevant plans for a given event. Refusing should then result in the adoption of a different relevant plan as a new intention. That is the true case of policy-based reconsideration, abandon is just an special case. Abandon is interpreted as rational behavior: the agent uses his learned policy-based reconsideration to prevent a real failure.

6 Discussion and Future Work

Experimental results are very promising. When compared with other experiments about commitment [7], it is observed that the *intentionalLearner* and *singleMinded* agents are adaptive: they were bold about *put*, and then they adopt a cautious strategy after having problems with their plan. Using intentional learning provides convergence to the right level of boldness-cautiousness

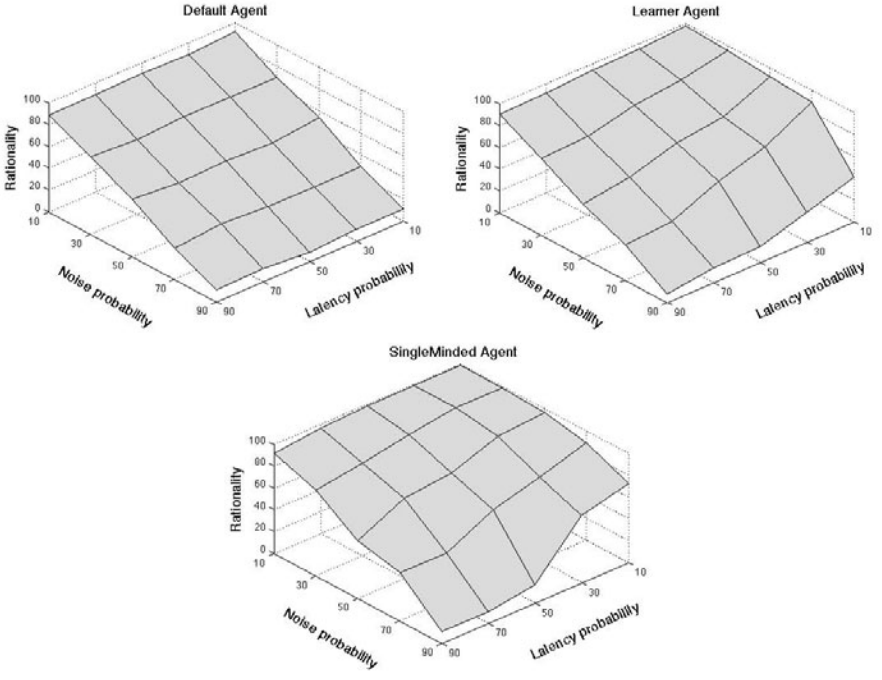


Fig. 4. The experiment results. Left: *Default* performance. Right: *Learner* performance. Center: *SingleMinded* performance.

based on their experience. But also, it seems that a bold attitude is adopted toward successful plans, and a cautious one toward failed plans; but more experiments are required to confirm this hypothesis.

The JILDT library provides the extensions to $AgentSpeak(L)$ required for defining intentional learning agents. Using the library, it was also easy to implement a single-mind committed class of agents. Extensions with respect to implementation include: implementing the inductive algorithm in java as an action of the JILDT library. Currently, the library computes the inputs for Tilde, but executes it to compute the logical decision trees. In this sense, we obtained a better understanding of the inductive method that will enable us to redefine it in JILDT. For instance, experimental results suggest that induction could be enhanced if the training examples represent not only the beliefs of the agent when the intention was adopted, but also when it was accomplished or dropped, in order to minimize the effects of the latency in noise.

The transition system for the *singleMinded* agents has been modified to enable dropping intentions. Basically, every time the system is at *execInt* and a drop learned rule fires, the intention is dropped instead of being executed. It is possible now to think of a formal operational semantics for $AgentSpeak(L)$ commitment based on policy-based reconsideration and intentional learning.

In [12] an architecture for intentional learning is proposed. Their use of the term intentional learning is slightly different, meaning that learning was the goal of the BDI agents rather than an incidental outcome. Our use of the term is strictly circumscribed to the practical rationality theory [4] where plans are predefined and the target of the learning processes is the BDI reasons to adopt them as intentions. A similar goal is present in [8], where agents can be seen as learning the selection function for applicable plans. The main difference with our work is that they propose an *ad hoc* solution for a given non BDI agent. Our approach to single-mind commitment evidences the benefits of generalizing intentional learning as an extension for Jason.

Acknowledgments. Authors are supported by Conacyt CB-2007 fundings for project 78910. The second author is also supported by scholarship 273098.

References

1. Blockeel, H., De Raedt, L.: Top-down induction of first-order logical decision trees. Artificial Intelligence 101(1-2), 285–297 (1998)
2. Blockeel, H., Raedt, L., Jacobs, N., Demoen, B.: Scaling up inductive logic programming by learning from interpretations. Data Mining and Knowledge Discovery 3(1), 59–93 (1999)
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley, England (2007)
4. Bratman, M.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge (1987)
5. Guerra-Hernández, A., Ortíz-Hernández, G.: Toward BDI sapient agents: Learning intentionally. In: Mayorga, R.V., Perlovsky, L.I. (eds.) Toward Artificial Sapience: Principles and Methods for Wise Systems, pp. 77–91. Springer, London (2008)
6. Guerra-Hernández, A., Castro-Manzano, J.M., El Fallah Seghrouchni, A.: CTL AgentSpeak(L): a Specification Language for Agent Programs. Journal of Algorithms (64), 31–40 (2009)
7. Kinny, D., Georgeff, M.P.: Commitment and effectiveness of situated agents. In: Proceeding of the Twelfth International Conference on Artificial Intelligence IJCAI 1991, Sidney, Australia, pp. 82–88 (1991)
8. Nowaczyk, S., Malec, J.: Inductive Logic Programming Algorithm for Estimating Quality of Partial Plans. In: Gelbukh, A., Kuri Morales, Á.F. (eds.) MICAI 2007. LNCS (LNAI), vol. 4827, pp. 359–369. Springer, Heidelberg (2007)
9. Rao, A.S., Georgeff, M.P.: Modelling Rational Agents within a BDI-Architecture. In: Huhns, M.N., Singh, M.P. (eds.) Readings in Agents, pp. 317–328. Morgan Kaufmann, San Francisco (1991)
10. Rao, A.S., Georgeff, M.P.: Decision procedures for BDI logics. Journal of Logic and Computation 8(3), 293–342 (1998)
11. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
12. Subagdja, B., Sonennberg, L., Rahwan, I.: Intentional learning agent architecture. Autonomous Agents and Multi-Agent Systems 18, 417–470 (2008)

Jason Induction of Logical Decision Trees (Jildt): A learning library and its application to Commitment

Alejandro Guerra-Hernández[1], Carlos Alberto González-Alarcón[1], and Amal El Fallah Seghrouchni[2]

[1] Departamento de Inteligencia Artificial
Universidad Veracruzana
Sebastián Camacho No. 5, Xalapa, Ver., México, 91000
aguerra@uv.mx, dn_carlos@hotmail.com
[2] Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
4, Place Jussieu, Paris, France, 75005
Amal.Elfallah@lip6.fr

Abstract. This paper[3] presents JILDT, a library that defines two agent classes for Jason, the java-based implementation of *AgentSpeak*(*L*). The agents defined as instances of these classes can learn about their reasons to adopt and abandon intentions, performing induction of logical decision trees. The library enables collecting training examples composed by the beliefs of the agents when a plan is adopted as an intention, labeled as success or failed executions. The induced trees are used to refine the context of the plans in question and to form rules for dropping intentions. The library is tested studying commitment in relation to intentional learning: A simple problem in a world of blocks is used to compare the behavior of a default Jason agent that does not reconsider his intentions; a learning agent that reconsiders by experience when to adopt intentions; and a single-minded agent that also drops intentions when this is rational. Results are very promissory for justifying a formal theory of single-mind commitment based on learning.

Keywords: Intentional Learning, AgentSpeak(L), Inductive Logic Programming, Logical Decision Trees, Commitment.

1 Introduction

It is well known that the Belief-Desire-Intention (BDI) model of agency [10, 11] lacks of learning competences. Coping with this, we introduce JILDT (Jason Induction of Logical Decision Trees): A library that defines two learning agent classes for Jason [3], the well known java-based implementation of the *AgentSpeak*(*L*) model [12].

[3] Published partially in MICAI 2010, LNAI Vol. 6437:375–385.

Agents defined as instances of the JILDT *intentionalLearner* class can learn about their reasons to adopt intentions, performing first-order induction of logical decision trees [1]. A set of plans and actions are defined in the library for collecting training examples of executed intentions, labeling them as succeeded or failed executions, computing the target language for the induction, and using the induced trees to modify accordingly the plans of the learning agents. In this way, the intentional learning approach [6] can be applied to any Jason agent by declaring the membership to this class.

The second class of agents defined in JILDT deals with single-mind commitment [10], i.e., once an agent intends something, he maintains his intention until he believes it has been accomplished or he believes it is not possible to eventually accomplish it anymore. It is known that Jason agents are not single-minded by default [3, 7]. So, agents defined as instances of the JILDT *singleMinded* class achieve single-mind commitment, performing a policy-based reconsideration, where policies are rules learned by the agents for dropping intentions . This is foundational and theoretical relevant, since the approach reconciles policy-based reconsideration, as defined in the theory of practical reasoning [4], with computational notions of commitment as the single-mind case [10]. Attending in this way the normative and descriptive aspects of reconsideration, opens the door for a formal theory of reconsideration in *AgentSpeak*(L) based on intentional learning.

Organization of the paper is as follows: Section 2 offers a brief introduction to the *AgentSpeak*(L) agent oriented programming language, as defined in Jason. An agent program, used in the rest of the paper, is introduced to exemplify the reasoning cycle of Jason agents. Section 3 introduces the Top-Down Induction of Logical Decision Trees (Tilde) method, emphasizing the way Jason agents can use it for learning. Section 4 describes the implementation of the JILDT library. Section 5 presents the experimental results for three agents in the blocks world: a default Jason agent, an intentional learner and a single-mind committed agent. Section 6 offers discussion, including related and future work.

2 Jason and AgentSpeak(L)

Jason [3] is a well known Java based implementation of the *AgentSpeak*(L) [12] abstract language for BDI agents. As usual an agent ag is formed by a set of plans ps and beliefs bs. Each belief $b_i \in bs$ is a ground first-order term. Each plan $p \in ps$ has the form $trigger\ event : context \leftarrow body$. A trigger event can be any update (addition or deletion) of beliefs (at) or goals (g).The context of a plan is an atom, a negation of an atom or a conjunction of them. A non empty plan body is a sequence of actions (a), goals, or belief updates. $\top$ denotes empty elements, e.g., plan bodies, contexts, intentions. Atoms (at) can be labelled with sources. Two kinds of goals are defined, achieve goals (!) and test goals (?).

The operational semantics [3] of the language, is given by a set of rules that define a transition system (see figure 1) between configurations $\langle ag, C, M, T, s\rangle$, where:

– ag is an agent program formed by a set of beliefs bs and plans ps.
– An agent circumstance C is a tuple $\langle I, E, A\rangle$, where: I is a set of intentions; E is a set of events; and A is a set of actions to be performed in the environment.
– M is a set of input/output mailboxes for communication.
– T stores the current applicable plans, relevant plans, intention, etc.
– s labels the current step in the reasoning cycle of the agent.

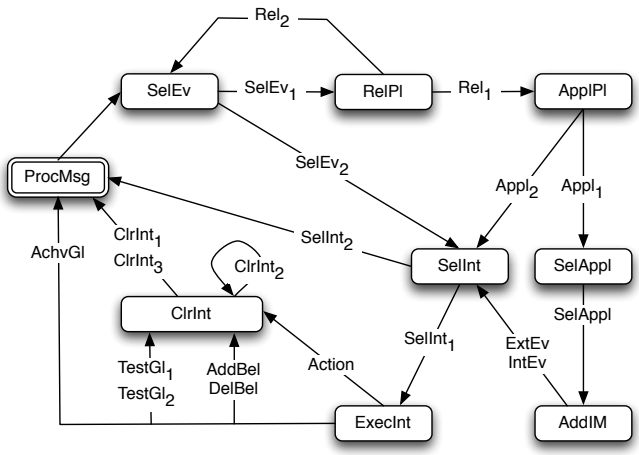


Fig. 1. The transition system for AgentSpeak(L) operational semantics.

An artificially simplified agent program for the blocks world environment, included in the distribution of Jason, is listed in the table 1. Examples in the rest of this paper are based on this agent program. Initially he believes that the *table* is clear (line 3) and that something with nothing on is clear too (line 2). He has a plan labeled *put* (line 10) expressing that to achieve putting a block X on Y, in any context ($true$), he must move X to Y. Our agent is bold about putting things somewhere else.

Now suppose the agent starts running in his environment, where someone else asks him to put b on c. A reasoning cycle of the agent in the transition system shown in Figure 1 is as follows: at the configuration labeled as $procMsg$ the beliefs about $on/2$ are perceived (lines 5–8) reflecting the state of the environment; and an event $+!put(b, c)$ is pushed on C_E. Then this event is selected at configuration $SelEv$ and the plan *put* is selected as relevant at configuration $RelPl$. Since the context of *put* is true, it is always applicable and it will be selected to form a new intention in C_I at $AddIM$. Once selected for execution at $SelInt$, the action $move(b, c)$ will be actually executed at $ExecInt$ and since there is nothing else to be done, the intention is dropped from C_I at $ClrInt$. Coming back to $ProcMsg$ results in the agent believing $on(b, c)$ instead of $on(b, a)$.

Table 1. A simplified agent in the blocks world.

```
1   // Beliefs
2   clear(X) :- not(on(_,X)).
3   clear(table).
4   // Beliefs perceived
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).
```

Now, what if something goes wrong? For instance, if another agent puts the block z on c before our agent achieves his goal? Well, his intention will fail. And it will fail every time this happens. The following section introduces the induction of logical decision trees, and the way they can be used to learn that *put* is applicable only when X and Y are clear.

3 Tilde

The Top-down Induction of Logical Decision Trees (Tilde) [1] is an inductive logic programming technique, that we have adopted for learning in the context of BDI agents [6]. The first-order representation of Tilde is adequate to form training examples as sets of beliefs, i.e., the beliefs of the agent supporting the adoption of a plan as an intention; and the obtained hypothesis are useful for updating the plans and beliefs of the agents, i.e., a logical tree expresses hypotheses about the successful or failed executions of the intentions, as illustrated in figure 2.

In what follows, Tilde is briefly introduced, emphasizing the compatibility with the agents in Jason. First, a Logical Decision Tree is a binary first-order decision tree where:

- Each node is a conjunction of first-order literals; and
- The nodes can share variables, but a variable introduced in a node can only occur in the left branch below that node (where it is true).

Three inputs are required to compute a logical decision tree: First, a set of training examples, where each trainning example, known as model, is composed by the set of beliefs the agent had when the intention was adopted; a literal coding what is intended; and a label indicating a successful or failed execution of the intention. Models are computed every time the agent believes an intention has been achieved (success) or dropped (failure). Table 2 shows two models corresponding to the examples in figure 2. The class of the examples is introduced

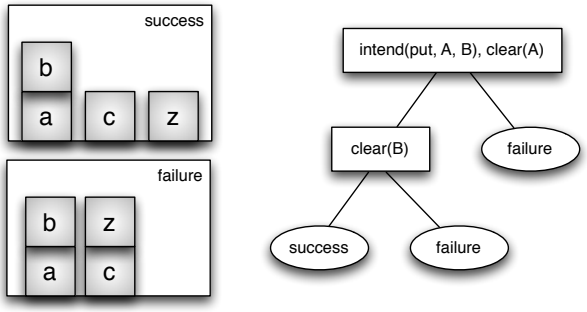


Fig. 2. A Tilde simplified setting: two training examples and the induced tree, when intending to put b on c.

Table 2. The training examples from figure 2 as models for Tilde. Labels at line 2.

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

at line 2, and the associated intention at line 3. The rest of the model corresponds to the beliefs of the agent when he adopted the intention.

Second, the rules believed by the agent, like *clear*/1 in table 1 (lines 2–3), do not form part of the training examples, since they constitute the background knowledge of the agent, i.e., general knowledge about the domain of experience of the agent.

And third, the language bias, i.e., the definition of which literals are to be considered as candidates to be included in the logical decision tree, is defined combinatorially after the literals used in the agent program, as shown in table 3. The *rmode* directives indicate that their argument should be considered as a candidate to form part of the tree. The *lookahead* directives indicate that the conjunction in their argument should be considered as a candidate too. The last construction is very important since it links logically the variables in the intended plan with the variables in the candidate literals, enabling generalization.

For the considered example, the induced decision tree for two successful examples and one failed is showed in table 4. Roughly, it is interpreted as: when intending to put a block A on B, the intention succeeds if A is clear and B is clear (line 3); and fails otherwise (lines 4 and 5). This tree is equivalent to the one shown in figure 2.

Table 3. The language bias defining the vocabulary to build the decision tree.

```
1  rmode(clear(V1)). rmode(on(V1,V2)). rmode(on(V2,V1)).
2  rmode(intend(put,V1,V2)).
3  lookahead(intend(put,V1,V2),clear(V1)).
4  lookahead(intend(put,V1,V2),clear(V2)).
5  lookahead(intend(put,V1,V2),on(V1,V2)).
6  lookahead(intend(put,V1,V2),on(V2,V1)).
```

Table 4. The induced Logical Decision Tree.

```
1  intend(put,A,B),clear(A) ?
2  + yes:  clear(B) ?
3  |     + yes:  [succ]   [1 examples(s)]
4  |     + no:  [fail]   [1 example(s)]
5  + no:  [fail]   1
```

3.1 Building a Logical Desicion Tree

Logical Decision Trees are computed recursively, as in ID3-like algorithms (table 1). Given an inicial query $Q = true$ and a set of labeled examples E, a set of refinement candidates $\rho(Q)$ is computed after the language bias (rmodes and lookaheads) and the query Q. Candidates are especializations of Q. The candidate that maximizes gain ratio (Eq. 1) is selected (line 2). The procedure finishes when a stop criteria (line 3) is reached, e.g., the gain ratio is not improved by the candidate. If this is the case, a leaf with the majority class for the example is returned. Otherwise, an internal node has to be computed. In order to do that, the content of the internal node $Conj$ is computed extracting the query Q from the candidate Q_b (line 6). The set of examples E is partitioned (lines 7–8) in those examples where the query succeeds (E_1) and those where it does not (E_2). Then the procedure is called recursively to build the left and right branches of the internal node with content $Conj$ (lines 9-11). The procedures returns the built tree T.

The best candidate Q_b is selected (Algorithm 2) as follows: Each refinement candidate $r_i \in \rho(\leftarrow Q)$ induces a partition in the examples E. A quality criterion of such split is computed using gain ratio (Eq. 1) as measure. For this, a matrix *counter* stores the number of examples that succeds and fails of each class c (lines 7–12). The candidate selected is the one that maximizes gain ration (line 17). Gain ration is an entropy based measure of information. The entropy of a set of examples E is:

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

Algorithm 1 Induction of Logical Decision Trees.

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

where k is the number of classes, c_i are the classes and $p(c_i, E)$ is the proportion of examples in E that belongs to class c_i.

We are interested in refinement candidates r_i that reduces the entropy of the examples, i.e., those that increases information. The Information Gain of a candidate, as usual, is computed as follows:

$$infoGain(E, counter) = s(E) - \sum_{V \in \{true, false\}} \frac{counter(V)}{|E|} s(E_{r_i})$$

where *counter* is the matrix computed in algorithm 2 and $E_{r_i} \subseteq E$ is the partition induced by r_i in the examples E. The maximal gain that a refinement candidate r_i can get is:

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

Finally, Gain Ratio is computed as follows:

$$gainRatio(E, counter) = \frac{infoGain(E, counter)}{maxGain(E)} \quad (1)$$

4 Implementation

JILDT implements two classes of agents: The first one is the *intentionalLearner* class, that defines agents capable of refining the context of their plans accordingly to the induced decision trees. In this way, the reasons to adopt a plan that has failed, as an intention in future deliberations, are reconsidered. The second one is the *singleMindedLearner* class, that implements agents that are also capable

Algorithm 2 Best candidate selection.

```
1: procedure BEST(E,R)      ▷ E is a set of examples, R = ρ(← Q) refinements of Q
2:     i := 1;
3:     for all r_i ∈ R do
4:         for all c ∈ C do                                   ▷ C = {success, failure}
5:             counter[true][c] := 0; counter [false][c] := 0;
6:         end for
7:         for all e ∈ E do
8:             if e ∧ B ⊨ r_i then
9:                 counter[true][class(e)]++;
10:            else
11:                counter[false][class(e)]++;
12:            end if
13:            s_i := gainRatio(E, counter);
14:        end for
15:        i++;
16:    end for
17:    return Q_b := r_i s.t. max(s_i)
18: end procedure
```

of learning rules that express when it is rational to drop an intention. The body of these rules is obtained from the branches in the induced decision trees that lead to failure. For this, the library defines a set of plans to allow the agents to autonomously perform inductive experiments, as described in section 3, and to exploit their discoveries. Table 5 lists the main internal actions implemented in java to be used in the plans of the library. The rest of the section describes the use of these plans by a learning agent.

Both classes of agents define a plan `@initialLearningGoal` to set the correct learning mode (intentional or singleMinded) by extending the user defined plans to deal with the learning process. For example, such extensions applied to the plan *put*, as defined for the agent listed in table 1, are shown in the table 6. The original body of the plan is at line 6. If this plan is adopted as an intention and correctly executed, then the agent believes (line 8) a new *succ*essful training *example* about *put*, including his beliefs at the time the plan was adopted.

Fun starts when facing problems: First, if the execution of an intention fails, for instance, because *move* could not be executed correctly, then a failure event is produced, e.g., $-!put(X, Y)$. JILDT adds the plan shown in table 7 to process these failure events, resulting in a $fail$ure training example added to the beliefs of the agent (line 4) and an inductive process intended to be achieved (line 5). The plan in table 8 deals with failures due to the lack of applicable plans. situation. It is rational to avoid commitment if there is no applicable plans for a given event.

The plan to achieve learning is shown in table 9. If the agent succeeds in computing a logical decision tree with the examples already collected, then he uses the tree to construct a new context for the associated plan (branches leading to

Table 5. Internal actions defined in the JILDT library.

Action	Description
`getCurrentBels(Bs)`	Bs unifies with the list of current beliefs of the agent.
`getCurrentCtxt(C)`	C unifies with the context of the current plan.
`getCurrentInt(I)`	I unifies with the current intention.
`getLearnedCtxt(P,LC,F)`	LC unifies with the learned context for plan P. F is true if a new different context has been learned.
`changeCtxt(P,LC)`	Changes the context of plan P for LC.
`setTilde(P)`	Builds the input files for learning about plan P.
`execTilde(T,G)`	Executes Tilde saving inputs and results. If T is *true*, a trace of Tilde algorithm is shown. If G is true, a visual tree is throwed.
`addDropRule(LC,P)`	Adds the rule to drop plan P accordingly to LC.
`setLearningMode`	Modifies plans to enable learning (intentionalLearner).
`setSMLearningMode`	Modifies plans to enable learning and dropping rules (singleMindedLearner class).

Table 6. JILDT extensions for plan *put* (original body at line 6).

```
1  @put
2  +!put(X,Y) : true <-
3    jildt.getCurrentInt(I);
4    jildt.getCurrentBels(Bs);
5    +intending(I,Bs);
6    move(X,Y);
7    -intending(I,Bs);
8    +example(I,Bs,succ);
```

success) and a set of rules for dropping the plan when it is appropriate (branches leading to failure). Two plans in the library are used to verify if something new has been learned.

There is a small ontology associated to the inductive processes. Table 10 lists the atomic formulae used with this purpose. These formulae should be treated as a set of reserved words.

The Tilde algorithm has been implemented as an action of JILDT. Logical consequence is computed using the Jason logical consequence method for logic formulae. Convergence of our implementation has been tested satisfactorily against ACE/Tilde [2] (same output for Bongard problems). The induced tree can be traced during induction and inspected in a graphic viewer (Figure 3).

5 Experiments

We have designed a very simple experiment to compare the behavior of a default Jason agent, an intentional learner, and single-minded agent that learns his

Table 7. A plan added by JILDT to deal with *put* failures requiring induction.

```
@put_failCase
-!put(X,Y) : intending(put(X,Y), Bs) <-
  -intending(I,Bs);
  +example(I,Bs,fail);
  !learning(put);
  +example_processed;
```

Table 8. A plan added by JILDT to deal with *put* being non applicable.

```
@put_failCase_NoRelevant
-!put(X,Y) : not .intend(put(X,Y)) <-
   .print("Plan ",put," non applicable.");
   +non_applicable(put).
```

policies for dropping intentions. For the sake of simplicity, these three agents are defined as shown in table 1, i.e., they are all bold about putting blocks somewhere else; and that is their unique competence.

The experiment runs as illustrated in figure 4: The *experimenter* asks the other agents to achieve putting the block b on c, but with certain probability $p(N)$, he introduces noise in the experiment by putting the block z on c, or on b. There is also a latency probability $p(L)$ for the last event: The *experimenter* could put block z before or after it asks the others agents to put b on c. This means that the other agents can perceive noise before or while intending to put b on c.

Numerical results are shown in table 11 (average of 10 runs, each one of 100 experiments) for a probability of latency of 50%. The probability of noise varies (90%, 70%, 50%, 30%, and 10%). Lower values configure less dynamic environments free of surprises and effectively observable. The performance of the agent is interpreted as more or less rational as follows: dropping an intention because of the occurrence of an error, is considered irrational. Refusing to form an intention because the plan is not applicable; dropping the intention because of a reason to believe it will fail; and achieving the goal of putting b on c are considered rational behaviors.

Figure 5 summarizes the result of all the executed experiments, where the probabilities of noise and latency range on $\{90\%, 70\%, 50\%, 30\%, 10\%\}$. As expected the performance of the *default* agent is proportionally inverse to the probability of noise, independently of the probability of latency.

The *learner* agent reduces the irrationality due to noise before the adoption of the plan as intention, because eventually he learns that in order to intend to put a block X on a block Y, X and Y must be clear:

Table 9. The learning plan

```
1  @learning
2  +!learning(P): true <-
3  .print(\"Trying to learn a better context...\");
4  jildt.setTilde(P);
5  jildt.execTilde(false,false);
6  jildt.getLearnedCtxt(P,LC,F);
7  !learningTest(P,LC,F).
```

Table 10. A small ontology used by JILDT.

Atom	Description
`drop(I)`	I is an intention to be dropped. Head of dropping rules.
`root_path(R)`	R is the current root to Tilde experiments.
`current_path(P)`	P is the current path to Tilde experiments.
`dropped_int(I)`	The intention I has been dropped.
`example(P,Bs,Class)`	A training example for plan P, beliefs Bs and $Class$.
`intending(I, Bs)`	I is being intended yet. $Class$ is still unknown.
`non_applicable(TE)`	There were no applicable plans for the trigger event TE.

Table 11. Experimental results (average from 10 runs of 100 iterations each one) for a probablity of latency of p(L)=0.5 and different probabilities of noise p(N).

Agent	p(N)	Irrational			Rational			
		after	before	total	refuse	drop	achieve	total
default	90	43.8	48.2	92.0	00.0	00.0	08.0	08.0
learner	90	48.7	37.3	86.0	04.5	00.0	09.5	14.0
singleMinded	90	44.5	38.8	83.3	03.2	03.8	09.7	16.7
default	70	34.5	36.0	70.5	00.0	00.0	29.5	29.5
learner	70	33.2	13.3	46.5	20.6	00.0	32.9	53.5
singleMinded	70	18.4	16.4	34.8	16.3	17.5	31.4	65.2
default	50	22.5	26.3	48.8	00.0	00.0	51.2	51.2
learner	50	26.1	05.4	31.5	20.7	00.0	47.8	68.5
singleMinded	50	11.6	09.9	21.5	16.1	14.9	47.5	78.5
default	30	14.2	15.0	29.2	00.0	00.0	70.8	70.8
learner	30	15.1	02.4	17.5	11.8	00.0	70.7	82.5
singleMinded	30	03.3	03.7	07.0	10.9	12.0	70.1	93.0
default	10	04.2	05.5	09.7	00.0	00.0	90.3	90.3
learner	10	05.3	01.0	06.3	04.9	00.0	88.8	93.7
singleMinded	10	00.9	00.9	01.8	03.8	03.4	91.0	98.2

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

Then, the *learner* can refuse to intend putting b on c if he perceives c is not clear. So, for low latency probabilities, he performs better than the default agent, but of course his performance decays as the probability of latency increases; and, more importantly: there is nothing to do if he perceives noise after the intention

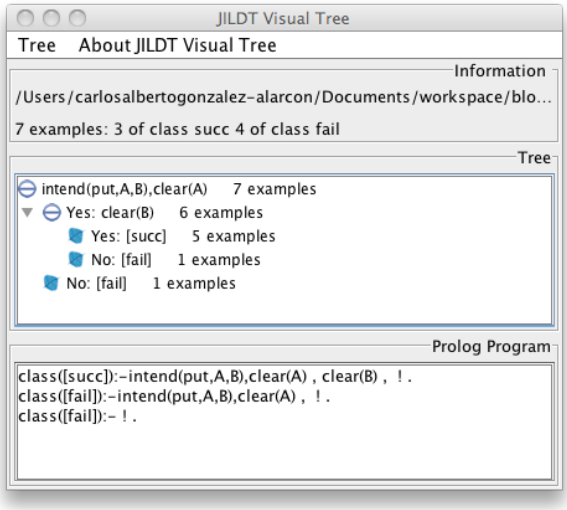


Fig. 3. The viewer displaying a Logical Decision Tree for the blocks world.

has been adopted. In addition, the *singleMinded* agent learns the following rules for dropping the intention when blocks X and Y are not clear:

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

Every time a *singleMindedLearner* agent is going to execute an intention, he verifies that no reasons to drop the intention exist; otherwise the intention is dropped. So, when the *singleMinded* agent already intends to put b on c and the experimenter puts the block z on c, he rationally drops his intention. In fact, the *singleMinded* agent only fails when it is ready to execute the primitive action *move* and noise appears.

For high probabilities of both noise and latency, the chances of collecting contradictory training examples increases and the performance of the *learner* and *the singleMinded* agents decay. By contradictory examples we mean that for the same blocks configuration, examples can be labeled as success, but also as failure. This happens because the examples are based on the beliefs of the agent when the plan was adopted as an intention, so that the later occurrence of noise is not included.

In normal situations, an agent is expected to have different relevant plans for a given event. Refusing should then result in the adoption of a different relevant plan as a new intention. That is the true case of policy-based reconsideration, abandon is just an special case. Abandon is interpreted as rational behavior: the agent uses his learned policy-based reconsideration to prevent a real failure.

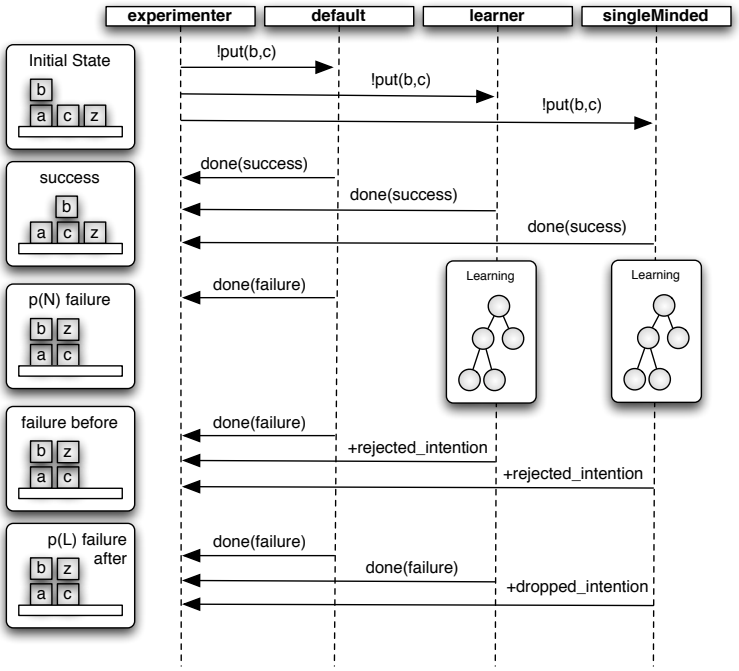


Fig. 4. The experiment process.

6 Discussion and future work

Experimental results are very promising. When compared with other experiments about commitment [8], it is observed that the *intentionalLearner* and *singleMinded* agents are adaptive: they were bold about *put*, and then they adopt a cautious strategy after having problems with their plan. Using intentional learning provides convergence to the right level of boldness-cautiousness based on their experience. And also, it seems that a bold attitude is adopted toward successful plans, and a cautious one toward failed plans; but more experiments are required to confirm this hypothesis.

The JILDT library provides the extensions to $AgentSpeak(L)$ required for defining intentional learning agents. Using the library, it was easy to implement a single-mind committed class of agents. We obtained a better understanding of the inductive method, that will enable us to improve it. For instance, it is possible to use the initial query of TILDE to avoid the use of lookaheads, reducing the number of candidates while computing the logical decision tree. Experimental results suggest that induction could be enhanced if the training examples represent not only the beliefs of the agent when the intention was adopted, but also when it was accomplished or dropped, in order to minimize the effects of the latency in noise. Also, incremental versions of the inductive method are now envisioned, as well as social learning protocols [5] exploiting distributed knowledge.

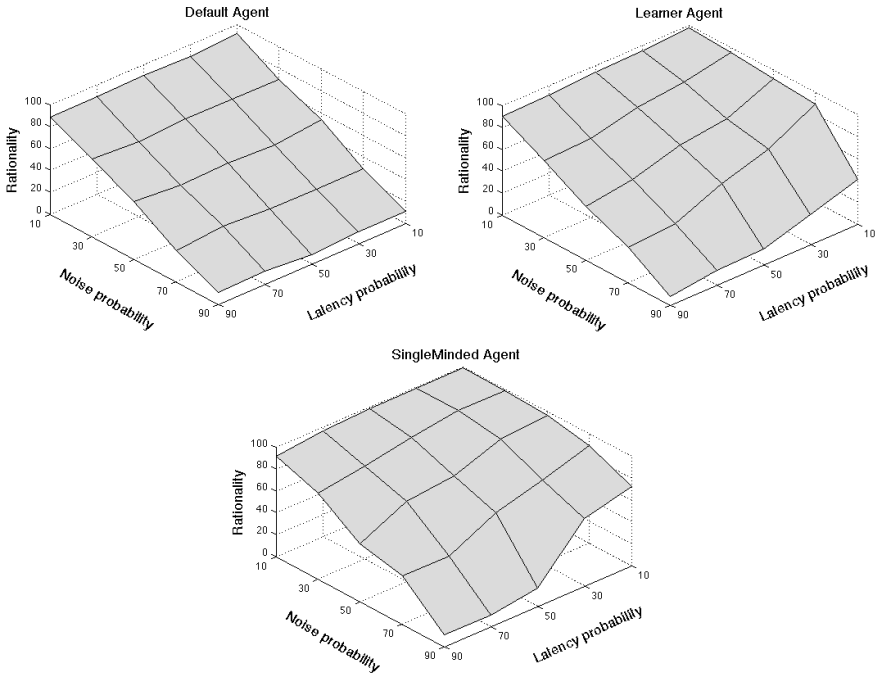


Fig. 5. The experiment results. Left: $Default$ performance. Right: $Learner$ performance. Center: $SingleMinded$ performance

The transition system (Figure 1) for the $singleMinded$ agents has been modified to enable dropping intentions. Basically, every time the system is at $execInt$ and a drop learned rule fires, the intention is dropped instead of being executed. It is possible now to think of a formal operational semantics for $AgentSpeak(L)$ commitment based on policy-based reconsideration and intentional learning.

In [13] an architecture for intentional learning is proposed. Their use of the term intentional learning is slightly different, meaning that learning was the goal of the BDI agents rather than an incidental outcome. Our use of the term is strictly circumscribed to the practical rationality theory [4] where plans are predefined and the target of the learning processes is the BDI reasons to adopt them as intentions. A similar goal is present in [9], where agents can be seen as learning the selection function for applicable plans. The main difference with our work is that they propose an *ad hoc* solution for a given non BDI agent. Our approach to single-mind commitment evidences the benefits of generalizing intentional learning as an extension for Jason.

Acknowledgments. Authors are supported by Conacyt CB-2007 fundings for project 78910. The second author is also supported by scholarship 273098.

References

1. Blockeel, H., De Raedt, L.: Top-down induction of first-order logical decision trees. Artificial Intelligence, 101(1–2):285–297 (1998)
2. Blockeel, H., Raedt, L., Jacobs, N., Demoen, B.: Scaling up inductive logic programming by learning from interpretations. Data Mining and Knowledge Discovery, 3(1):59–93 (1999)
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley, England (2007)
4. Bratman, M.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge (1987)
5. A. Guerra-Hernández, A. El-Fallah-Seghrouchni, and H. Soldano. Distributed learning in BDI Multiagent Systems. In R. Baeza-Yates, M. J.L., and E. Chávez, editors, *Fifth Mexican International Conference on Computer Science*, pages 225–232, USA, 2004. Sociedad Mexicana de Ciencias de la Computación (SMCC), IEEE Computer Society.
6. Guerra-Hernández, A., Ortíz-Hernández, G.: Toward BDI sapient agents: Learning intentionally. In: Mayorga, R.V., Perlovsky, L.I. (eds.) Toward Artificial Sapience: Principles and Methods for Wise Systems, pp. 77–91. Springer, London (2008)
7. Guerra-Hernández, A., Castro-Manzano, J. M., El Fallah Seghrouchni, A.: CTL AgentSpeak(L): a Specification Language for Agent Programs. Journal of Algorithms, (64):31–40 (2009)
8. Kinny, D., Georgeff, M. P.: Commitment and effectiveness of situated agents. In Proceeding of the Twelfth International Conference on Artificial Intelligence IJCAI-91, pp. 82–88, Sidney, Australia (1991)
9. Nowaczyk, S., Malec, J.: Inductive Logic Programming Algorithm for Estimating Quality of Partial Plans. In MICAI 2007, LNAI, vol. 4827, pp. 359–369 Springer Verlag, Heidelberg (2007)
10. Rao, A.S., Georgeff, M.P.: Modelling Rational Agents within a BDI-Architecture. In: Huhns, M.N., Singh, M.P., (eds.) Readings in Agents, pp. 317–328. Morgan Kaufmann (1991)
11. Rao, A.S., Georgeff, M.P.: Decision procedures for BDI logics. Journal of Logic and Computation 8(3), pp. 293–342 (1998)
12. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) MAAMAW. LNCS, vol. 1038, pp. 42–55. Springer Verlag, Heidelberg (1996)
13. Subagdja, B., Sonennberg, L., Rahwan, I.: Intentional learning agent architecture. Autonomous Agents and Multi-Agent Systems, 18:417–470 (2008)