Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Departamento de Ciencias de la Computación

Tesina de grado presentada por Adrián Biga

G-JASON : Una Implementación de un Sistema de Razonamiento Procedural (PRS) que incorpora incertidumbre

---

Dirigida por:
Dra Ana Casali

Dedicado a

mis abuelas que me apoyan incondicionalmente desde el cielo.

Agradecimientos

Agradezco a mi familia por estar «siempre» a mi lado. A mi esposa Graciela por su eterna paciencia y aguante, en las buenas y en las malas. A Ana por haberme apoyado y ayudado durante toda mi carrera. Sin su «incondicional aguante» no hubiera sido fácil alcanzar este objetivo.

Muchas Gracias a todos!

Resumen

Hasta el momento, las mejores implementaciones de agentes de la familia del modelo BDI (B: Belief, D: Desire, I: Intention) son los llamados Sistemas de Razonamiento Procedural, PRS (Procedural Reasoning Systems). En esta tesina se plantea una extensión de los sistemas PRS para permitir generar agentes más flexibles, que puedan representar la incertidumbre del entorno y distintos grados en los planes del agente. Esta extensión está inspirada en el modelo de agente BDI graduado (g-BDI) planteado por Casali (2009) donde se representan grados en las distintas estructuras de datos que representan los estados mentales del agente (B,D e I). La extensión propuesta se definió a nivel sintáctico y semántico y se implementó en JASON [1] que es una implementación de sistemas PRS en JAVA, que posee licencia de código abierto, está bien documentado y cuenta con una importante comunidad de usuarios.

[1] (Bordini & Hübner 2007)

Índice general

Capítulo 1

Introducción

La idea de programación orientada a agentes fue introducida por Yoav Shoam (Shoam 1993). El desarrollo de sistemas multiagentes ha recibido gran atención dentro de las Ciencias de la Computación en los últimos años, alegando que la tecnología emergente de esta área va influir en el diseño de sistemas computacionales situados en ambientes muy dinámicos e impredecibles. Desde esa perspectiva, la programación orientada a agentes pueda convertirse en una potente herramienta de desarrollo de sistemas computacionales en un futuro no muy lejano. En los últimos años ha crecido el interés en modelar sistemas complejos como sistemas multiagentes (Shoam 1993, Wooldridge & Jennings 1995, Wooldridge 2001). Dentro de las arquitecturas más notorias para dar soporte a los agentes que componen estos sistemas, se encuentra la arquitectura BDI (B: Belief, D: Desiere, I: Intention) propuesta por Rao & Georgeff (1991) y Rao & Georgeff (1995). Esta arquitectura de agentes ha sido una de las más estudiada y utilizada en distintas aplicaciones reales de importancia (Casali, Godo & Sierra 2008).

Los sistemas de razonamiento procedural (Procedural Reasoning System) originalmente desarrollado Georgeff & Lansky (1987), fue posiblemente la primera implementación de una arquitectura basada en el paradigma BDI. Desde la primera implementación de un sistema PRS, se han desarrollado distintas versiones como el sistema d-MARS (D'Inverno et al. 1998), una versión en C++ denominada Open-PRS (Ingrand 2004), y respecto a implementaciones en Java destacamos dos versiones actualmente utilizadas: JACK (Georgeff & Ingrand 1989) y JASON (Bordini & Hübner 2007).

Entre las ventajas principales de PRS se destacan: un conjunto de herramientas y métodos para la representación y ejecución de planes y procedimientos. Una arquitectura PRS consiste básicamente de: (1) una base de datos con las creencias (believes) actuales sobre el mundo (su entorno), la cual es actualizada para reflejar cambios; (2) un conjunto de objetivos o deseos (goals) actuales a alcanzar; (3) una librería de planes o procedimientos, describen las secuencias particulares de acciones y pruebas que pueden realizarse para alcanzar ciertos objetivos para reaccionar ante ciertas situaciones; y (4) una estructura de intenciones, que consiste de un conjunto ordenado (parcialmente) de todos los planes elegidos para ejecución. Los PRS han sido utilizados para modelar sistemas complejos en distintos dominios del mundo real (Georgeff & Lansky 1987, Georgeff & Ingrand 1989, Georgeff et al. 1999).

Una de las limitaciones tanto el modelo BDI (Rao & Georgeff 1995) como los sistemas basados en PRS (D'Inverno et al. 1998), es que no contemplan una forma explícita de representar la incertidumbre del entorno, estos se han planteado considerando una lógica bi-valuada para dar soporte a toda la in-

formación que utiliza un agente para la toma de decisiones (representada en sus estados mentales: B, D e I). Para flexibilizar el modelo BDI, Casali et al. (Casali, Godo & Sierra 2004, Casali, Godo & Sierra 2011) han presentado el modelo BDI graduado (g-BDI) que permite modelar un agente que trate la incertidumbre del entorno y represente sus preferencias graduadas. Este framework a partir del cuál se pueden definir distintas arquitecturas de agentes g-BDI se han especificado utilizando sistemas multicontexto (Ghidini & Giunchiglia 2001) y se han definido las lógicas adecuadas para representar cada uno de sus componentes (Wooldridge & Jennings 1995).

En este trabajo se propone analizar posibles extensiones de los sistemas PRS para representar estructuras de datos graduadas en sus creencias (Beliefs), deseos (Goals) y ver como a partir de estos elementos, las intenciones (Intentions) también tendrán distintos grados, los que llevarán al agente a optar por una acción a seguir.

Dado que existen distintas implementaciones de sistemas PRS (D'Inverno et al. 1998), se decide trabajar con JASON (Bordini & Hübner 2007) para aplicar esta extensión, por poseer licencia de código abierto y estar desarrollado en lenguaje JAVA, lo que permite alta portabilidad en el sistema y también en las implementaciones desarrolladas.

Se observa el avance en la modelización de la incertidumbre (g-BDI) para agentes pero el estado del arte se encuentra bastante lejos de tener una implementación genérica de estos aspectos. Por lo cual se han realizado extensiones a la herramienta JASON para poder implementar la incertidumbre de una forma práctica.

Capítulo 2

Conceptos Relacionados

2.1. Arquitecturas y modelos de agentes

En este capítulo se presentan conceptos relacionados al desarrollo de este trabajo. En primer lugar se observarán las distintas arquitecturas de agentes y se presenta la noción de agente. Probablemente la forma más general en la que se puede describir un agente corresponde a un sistema de software o hardware con las siguientes propiedades (Wooldridge & Jennings 1995, Wooldridge 2001):

- Autonomía: los agentes operan sin la directa intervención de humanos y tienen cierto control sobre sus acciones y estado interno.
- Habilidad social: interactúan con otros agentes (eventualmente humanos) mediante algún lenguaje de comunicación.
- Reactividad: perciben su entorno y lo modifican según sus necesidades.
- Pro-actividad: los agentes no solamente responden a su entorno sino

que también tienen un comportamiento en el cual toman la iniciativa de realizar sus objetivos personales.

Una noción más fuerte dentro de los trabajos de IA le agregan a las características mencionadas conceptos aplicados más comúnmente a los humanos. En esta dirección se puede caracterizar a un agente como ser intencional, como ser creencias, deseos, intenciones y obligaciones, así como también algunos trabajos consideran sus emociones.

Si se tiene en cuenta lo anterior, se valorarán las arquitecturas de agentes, las cuales representan el paso de la especificación a la implementación. La cuestión será cómo construir sistemas que satisfagan las propiedades que se esperan de un agente. La ingeniería de software orientada a agentes determina la interconexión de los módulos de software/hardware que permiten a un agente exhibir la conducta que le hayamos enunciado. Frente a otras tecnologías con componentes fijos como la de objetos (atributos y métodos) o la de sistemas expertos (motor de inferencias, base de conocimiento y otros elementos), en los agentes nos encontramos con una gran variedad de arquitecturas.

Maes (Maes 1991) propone la siguiente definición sobre arquitectura de agentes:

"(Es una metodología particular para construir agentes.) Especifica como el agente puede ser descompuesto como un conjunto de módulos y como éstos se interrelacionan. El conjunto total de módulos y sus interacciones brindan las respuestas a la preguntas de cómo los datos sensados y los actuales estados

mentales del agente determinan sus acciones y el futuro estado mental del mismo."

Existen diferentes propuestas para la clasificación de arquitecturas de agentes. Tomando como referencia la clasificación propuesta por Wooldridge (Wooldridge 2001) se puede considerar las siguientes clases concretas de arquitecturas:

- Deliberativas: se encuentran dentro de la IA simbólica. Las arquitecturas deliberativas de agentes suelen basarse en la teoría clásica de planificación de inteligencia artificial: dado un estado inicial, un conjunto de operadores y un estado objetivo, la deliberación del agente consiste en determinar qué pasos debe encadenar para lograr su objetivo, siguiendo un enfoque descendente (topdown).

- Reactivas: estas arquitecturas cuestionan la viabilidad del paradigma simbólico y proponen una arquitectura que actúa siguiendo un enfoque conductista, con un modelo estímulo-respuesta. Las arquitecturas reactivas no tienen un modelo del mundo simbólico como elemento central de razonamiento y no utilizan razonamiento simbólico complejo, sino que siguen un procesamiento ascendente (bottom-up), para lo cual mantienen una serie de patrones que se activan con ciertas condiciones de los sensores y tienen un efecto directo en los actuadores.

- Híbridas: Estas arquitecturas combinan módulos reactivos con módulos deliberativos. Los módulos reactivos se encargan de procesar los

estímulos que no necesitan deliberación, mientras que los módulos deliberativos determinan que acciones deben realizarse para satisfacer los objetivos locales y cooperativos de los agentes. Las arquitecturas híbridas pueden ser horizontales (donde todas las capas tienen acceso a los datos del entorno y a realizar acciones en el entorno) y verticales (una capa tiene el acceso a los datos del entorno y a realizar acciones en él).

- Arquitecturas de razonamiento práctico (agentes BDI): Es un modelo que decide momento a momento que acción seguir en orden a satisfacer los objetivos. Esta arquitectura incluye dos importantes procesos: deliberación -decide en cada caso que objetivos perseguir- y razonamiento de medios-fines (means-ends) -determinando la forma en que se buscaran dichos objetivos. Luego de la generación de los objetivos el agente debe elegir uno y comprometerse a alcanzarlo. Los objetivos que decide alcanzar son sus intenciones, las cuales juegan un papel crucial en el proceso de razonamiento práctico.

La arquitectura Belief-Desire-Intention (BDI) es una de las más relevantes dentro de esta línea y fue originada en el trabajo de la Rational Agency Project en el Stanford Research Institute a mediados del los 80' (Rao & Georgeff 1991, Rao & Georgeff 1995). El origen del modelo fue debido a la teoría sobre razonamiento práctico en humanos desarrollada por M. Bratman (Bratman 1987) que se enfoca particularmente en el rol de las intenciones. Para este trabajo se utiliza puntualmente la última arquitectura mencionada (BDI), la cual se desarrolla con mayor detalle a continuación.

2.1.1. Arquitectura BDI

La arquitectura BDI (Belief, Desire, Intention) está caracterizada porque los agentes que la implementan están dotados de los estados mentales que representan las Creencias , Deseos e Intenciones. Este modelo es uno de los más difundidos y estudiados dentro de los modelos de agentes.

Una arquitectura construida con base en el modelo de agentes BDI, incluye las siguientes estructuras de datos:

- Creencias (Beliefs): representan el conocimiento que el agente tiene sobre sí mismo y sobre el entorno.
- Deseos (Desires): son los objetivos que el agente desea cumplir.
- Intenciones (Intentions): se puede considerar como un subconjunto de deseos consistentes entre sí que el agente decide alcanzar. Las intenciones derivan en las acciones que ejecutará el agente en cada momento.

El proceso de razonamiento práctico de un agente BDI puede ser ilustrado siguiendo a Weiss (1999) en el esquema presentado en la Figura 2.1. En dicha figura además de los tres componentes principales del modelo (B, D e I) aparecen otros componentes que se describen a continuación:

- Una función de revisión de creencias (brf, belief revision function), que toma la entrada sensada y el estado actual de creencias del agente para determinar un nuevo conjunto de creencias: (donde p representa conjunto) $brf : p(Bel) \times P- > p(Bel)$

- Una función generadora de opciones (generate options), que determina qué opciones están disponibles para el agente (sus deseos), en base a las creencias actuales y sus actuales intenciones: $options : p(Bel) \times p(Int) -> p(Des)$

- Una función de filtrado (filter), que representa el proceso de deliberación del agente en la que el agente determina sus intenciones, basado en sus actuales creencias, deseos e intenciones: $filter : p(Bel) \times p(Des) \times p(Int) -> p(Int)$

- Una función selectora de acciones (actions), la cual determina en cada caso qué acción llevar a cabo en base a las intenciones vigentes: $execute : p(Int) -> A$

Hay varias razones que contribuyen a la importancia del modelo BDI. Este modelo está provisto con componentes esenciales que le permiten abordar aplicaciones del complejo mundo real. Estos sistemas reales suelen estar planteados en entornos dinámicos e inciertos.

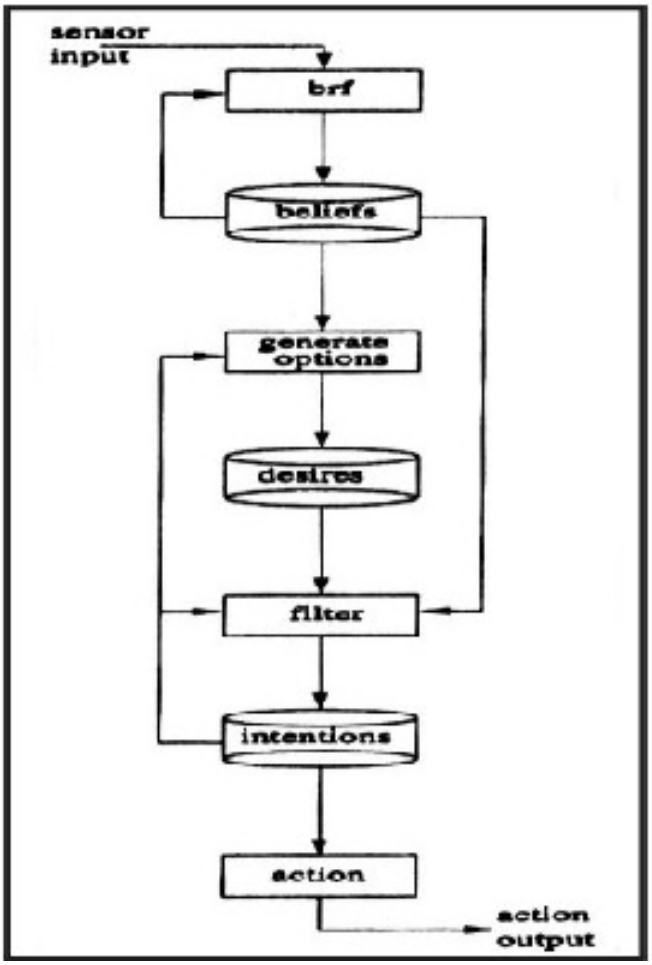


Figura 2.1: Diagrama esquemático de la arquitectura genérica belief-desire-intention

Fuente: Weiss, 1999, p58.

El modelo BDI es también interesante porque se ha realizado sobre él un gran esfuerzo para su formalización a través de las lógicas BDI, la cual ha sido aceptada por la comunidad científica (Rao & Georgeff 1995). Además, la importancia del modelo no se suscribe al marco teórico, sino que desde 1980 se han presentado diferentes desarrollos de esta arquitectura. Entre

las implementaciones podemos mencionar por ejemplo a IRMA (Intelligent Resourcebounded Machine Architecture) (Bratman, Israel & Pollack 1988), PRS (Procedural Reasoning Systems) (Wooldridge 1996), dMARS(distribute Multi-Agent Reasoning System) (D'Inverno et al. 1998).

En el modelo BDI (Rao & Georgeff), sus distintas variantes de este modelo y muchas de sus implementaciones están construidas sobre lógicas bivaluadas. Estos presentan limitaciones al momento de representar mundos inciertos, ya que carece de una forma de representar una estimación acerca de la posibilidad o probabilidad de que el mundo planteado sea el real. Esto ha llevado a Casali, Godo & Sierra (Casali, Godo & Sierra 2005, Casali, Godo & Sierra 2011) a pensar en un modelo multivaluado -graduado- que mejore la performance del agente utilizando grados en las actitudes mentales (B, D, I). Esta tesina se plantea también en esta dirección.

2.1.2. PRS (Procedural Reasoning Systems)

La arquitectura PRS desarrollada por Georgeff y Lansky (Georgeff & Lansky 1987) fue quizás la primera arquitectura basada en el paradigma BDI y se ha convertido en una de las más conocidas. Ha sido utilizada en varias aplicaciones. Una de las más importantes de esta arquitectura es en control de trafico aéreo (Georgeff & Ingrand 1989), para ayudar a prevenir el congestionamiento de tráfico determinando el orden en que deben aterrizar los aviones. También fue aplicado al monitoreo de diferentes subsistemas en naves espaciales de la NASA (Ljungberg & Lucas 1992). Un agente con arquitectura PRS trata de alcanzar cualquier meta que se proponga basándose

en sus creencias sobre el mundo (entorno). También puede simultáneamente reaccionar ante la ocurrencia de algún nuevo evento. De esta forma, PRS provee un marco en el cual los comportamientos de tipo "*goal-directed*" y "*event-directed*" pueden ser fácilmente integrados.

Los sistemas PRS consisten en un conjunto de herramientas y métodos, para la representación y ejecución de planes. Estos planes o procedimientos son secuencias condicionales de acciones las cuales pueden ejecutarse para alcanzar ciertos objetivos, o reaccionar en situaciones particulares.

Una arquitectura PRS consiste básicamente de: (1) una base de datos con las Creencias (beliefs) actuales sobre el mundo (su entorno), la cual es actualizada para reflejar cambios; (2) un conjunto de objetivos o deseos (goals) actuales a alcanzar; (3) una librería de planes o procedimientos, describen secuencias particulares de acciones y pruebas que pueden realizarse para alcanzar ciertos objetivos o para reaccionar ante ciertas situaciones; y (4) una estructura de intenciones, que consiste de un conjunto ordenado (parcialmente) de todos los planes elegidos para ejecución. Estos componentes se ilustran en la Figura 2.2.

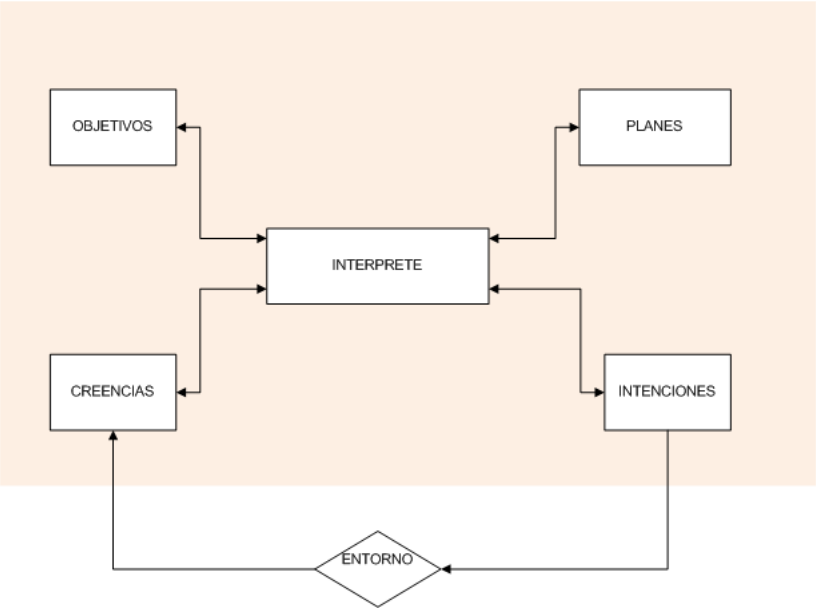


Figura 2.2: PRS. Una arquitectura basada en el paradigma BDI.

Un intérprete (mecanismo de inferencia) manipula estos componentes. Recibe nuevos eventos y objetivos internos; selecciona un plan (procedimiento) apropiado teniendo en cuenta los nuevos eventos, objetivos y creencias; ubica el plan dentro de la estructura de intenciones (grafo); elije un plan (intención) en la estructura y finalmente ejecuta un paso del plan activo. Esto puede resultar en una acción primitiva o en la formulación de un nuevo objetivo. El sistema interactúa con el entorno a través de su base de datos y de las acciones básicas o primitivas.

Las estructuras de la Figura 2.2 pueden tener distintas representaciones, dependiendo de la plataforma o implementación de PRS elegida. En general la lógica utilizada para la representación del conocimiento es prácticamente la misma para todas las implementaciones PRS. Se observarán con más detalle

en el proximo capítulo los componentes principales de un PRS. (Creencias, Objetivos, Planes)

2.2. Modelo BDI-graduado (g-BDI)

Las distintas arquitecturas de agentes desarrolladas hasta el momento, han sido planteadas para manejar información básicamente bivaluada. Si bien el modelo BDI (Rao & Georgeff 1991), reconoce explícitamente que el conocimiento de un agente sobre el mundo es incompleto, no plantea usar la cuantificación para dicho fin.

En el modelo BDI-graduado presentado en Casali et al.(Casali, Godo & Sierra 2004, Casali, Godo & Sierra 2011), se plantea un modelo general de agente BDI graduado especificando una arquitectura que pueda tratar con la incertidumbre del entorno y actitudes mentales graduadas. De esta forma los grados de las creencias van a representar en que medida el agente cree que una fórmula es cierta, los grados de los deseos permiten representar el nivel de preferencia y el grado de las intenciones darán una medida costo-beneficio que le representa al agente alcanzar cada objetivo.

Para especificar la arquitectura de un agente BDI graduado, se utiliza la noción de sistema multicontextos. Estos sistemas han sido planteados por **?** y utilizado para la especificación de agentes BDI (Parsons, Sierra & Jennings 1998).

Una especificación multicontexto de agente consta básicamente de unidades o contextos, lógicas y bridge rules. De esta forma, un agente es definido

como un grupo de contextos interconectados: $< \{C_i\}_{i\epsilon I}, \Delta_{br} >$, donde cada contexto $\{C_i\}_{i\epsilon I}$ queda definido por $C_i =< L_i, A_i, \Delta_i >$ donde L_i es un lenguaje, A_i es un conjunto de axiomas y Δ_i son reglas de inferencia, los cuales definen la lógica que utiliza el contexto. Para definir un sistema concreto se establece una teoría $T_i \subset L_i$ asociada a cada unidad. Δ_{br} son reglas de inferencia donde las premisas y conclusiones pertenecen a distintos contextos, por ejemplo la regla:

$$\frac{C_1 : \psi, C_2 : \gamma}{C_3 : \theta}$$

significa que podemos deducir θ en el contexto C_3 si las fórmulas ψ y γ son deducidas en C_1 y C_2, respectivamente. En estos sistemas la deducción está a cargo de dos tipos de reglas de inferencia, las reglas internas Δ_i de cada contexto y las bridge rules Δ_{br}. Las reglas internas infieren consecuencias dentro de una teoría, mientras que las bridge rules permiten exportar los resultados de una teoría a otra.

El modelo de agente utilizado en el modelo g-BDI tiene contextos para representar sus creencias (BC), deseos (DC) e intenciones (IC). También se pueden considerar otras unidades funcionales como por ejemplo una unidad dedicada a la comunicación (CC), la cual establece una interface única y bien definida con el entorno, representando los sensores y actuadores del agente. Además se incluye un contexto de planificación (PC) que es el encargado de encontrar un plan para cambiar de un estado del mundo a otro nuevo.

También se consideran un conjunto básico de reglas puentes Δ_{br}. Las cuales nos permiten realizar deducciones donde premisas y conclusiones pertenecen a distintos contextos. Luego el modelo de agente es:

$Ag =< \{BC, DC, IC, CC, SC, PC\}, \Delta_{br} >$, cuyo esquema puede verse en la figura 2.3. Donde se indican los distintos contextos y las reglas puentes ((1) al (6)).

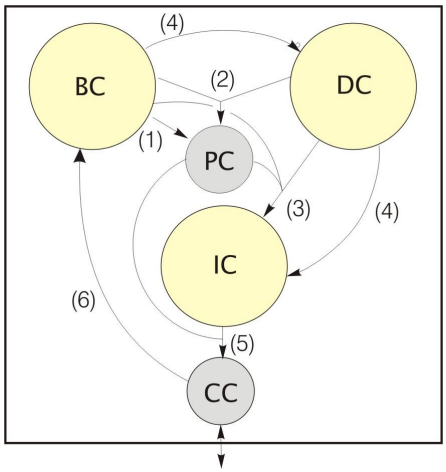


Figura 2.3: Modelo multicontexto de un agente BDI graduado.

Cada contexto tiene su lógica asociada, lo que significa que se define un lenguaje, con su semántica y su sistema deductivo. Para representar y razonar con grados en las creencias, deseos e intenciones, se ha elegido utilizar lógicas modales multivaluadas. En particular se siguió la propuesta de Godo, Esteva & Hajek (2000), donde el razonamiento bajo incertidumbre se trata definiendo teorías modales adecuadas sobre lógicas multivaluadas.

Presentamos la idea básica considerando el contexto BC donde los grados de belief son modelados por ejemplo, como probabilidades. Luego, para

cada fórmula clásica (bi-valuada), podemos considerar la formula modal $B\gamma$ denotando "γ es probable". Esta fórmula $B\gamma$ es entonces una formula fuzzy (multivaluada), la cual puede ser verdadera en cierto grado, dependiendo de la probabilidad de γ. En particular, se puede considerar como grado de verdad de $B\gamma$ a la probabilidad de γ.

Se necesita entonces, un marco lógico multivaluado para razonar sobre las formulas modales $B\gamma$'s, pero de forma que mantenga el modelo de incertidumbre que se elija para representar los grados de creencia. Es decir, un marco, donde además de los axiomas de la lógica multivaluada particular, se expresen los axiomas que garanticen una correcta interpretación del modelo de incertidumbre. En la propuesta realizada por Casali, Godo & Sierra (2004) se utiliza como lógica multivaluada la lógica de Lukasiewicz, por permitir modelizar distintas medidas de incertidumbre, en particular las probabilidades. El mismo marco lógico puede aplicarse a la representación y razonamiento con grados en los deseos e intenciones.

Las características generales de los distintos componentes de un agente BDI graduado se pueden ver en Casali, Godo & Sierra (2004) y Casali, Godo & Sierra (2011). Inspirados en este modelo de agentes g-BDI, el objetivo de esta tesina es evaluar cuales de las ideas planteadas en este modelo pueden ser implementadas en una versión de sistemas PRS y en particular, se trabajará con JASON.

Capítulo 3

La Arquitectura de JASON

3.1. Introducción a JASON

En esta sección se hará una reseña de los conceptos más importantes de la herramienta JASON los cuales se tratarán en más profundidad en las siguientes secciones. Si bien JASON puede usarse mediante su integración con sistemas multiagentes como por ejemplo, utilizando JADE (Bellifemine et al. 2005), este trabajo se centrará en el comportamiento individual de los agentes, considerando que estos se vincularán con otros.

El lenguaje interpretado por JASON es una extensión del AgentSpeak(L), un lenguaje abstracto originalmente diseñado por Rao & Georgeff (1995) y secuencialmente extendido por un grupo de expertos encabezados por Rafael Bordini. AgentSpeak(L) tiene una concisa notación y es una eficiente e inteligente extensión de la programación lógica para sistemas BDI. La arquitectura BDI (Beliefs-Desires-Intentions) es una de las mejores aproximaciones para el desarrollo de agentes cognitivos (Rao & Georgeff 1995). También provee un framework elegante y abstracto para la programación de este tipo de agentes.

Los tipos de agentes especificados con AgentSpeak (L) son referidos a veces como sistemas de planificación reactivos.

JASON a comparación de otros sistemas de agentes BDI posee la ventaja de ser multiplataforma al estar desarrollado en el lenguaje JAVA, está disponible su código Open Source y es distribuido bajo la licencia GNU LGPL. Además de interpretar el lenguaje AgentSpeak (L) original, JASON posee las siguientes características:

- Strong Negation.
- Manejo de planes de falla. (planes "*especiales*"que se ejecutan exepcionalmente en situación de falla)
- Comunicación entre agentes.
- Anotaciones de las creencias en las fuentes de información.
- Soporte del desarrollo de entornos (Programados en JAVA).
- Anotaciones en las etiquetas o labels de los planes que pueden ser usadas por funciones selectoras modificadas o extendidas.
- Posibilidad de correr un sistema multiagente distribuido sobre la red.
- Extendible (funciones selectoras, funciones de verdad y la arquitectura completa).
- Librería de acciones internas esenciales.
- Acciones internas extensibles por el usuario, programadas en JAVA.

3.1.1. Arquitectura de agente en AgentSpeak

Un agente en **AgentSpeak (L)** es creado especificando un conjunto de creencias (beliefs) y un conjunto de planes (plans). Otros elementos relevantes son los objetivos (goals) del agente y los eventos disparadores (trigger events) que sirven para representar la parte reactiva de un agente.

- Creencias: Representarán las creencias del agente respecto a su entorno. Un átomo de creencia (belief atom) es un predicado de primer orden. Los belief atoms y sus negaciones son literales de creencia (belief literals). Un conjunto inicial de creencias es tan sólo una colección de átomos de creencia.

- Objetivos: Representarán los objetivos del agente, AgentSpeak (L) distingue sólo dos tipos de objetivos (goals): *achivement goals* y *test goals*. Estos dos son predicados prefijados por los operadores '!' y '?' respectivamente.

    - El *achivement goal* denota que el agente quiere alcanzar un estado en el mundo donde el predicado asociado sea verdadero (en la práctica, esto con lleva a la ejecución de subplanes).
    - Un *test goal* devuelve una unificación asociada con un predicado en el conjunto de creencias del agente, si no hay asociación simplemente falla.

- Un evento disparador (*trigger event*) define que eventos pueden iniciar la ejecución de un plan. Un evento puede ser interno, cuando un sub-

objetivo tiene que ser logrado, o externo, cuando es generado por actualizaciones de creencias debido a una percepción del ambiente. Hay dos tipos de eventos disparadores, relacionados con el hecho de que agregan o quitan actitudes mentales (creencias u objetivos). Estos son prefijados por '+' y '-' respectivamente.

- Planes: Los planes son acciones básicas que un agente puede realizar sobre su ambiente. Estas acciones están también definidas como predicados de primer orden, pero con símbolos especiales, llamados símbolos de acción, usados para distinguirlos de otros predicados. Un plan está formado por un evento disparador (denotando el propósito del plan), seguido por una conjunción de literales de creencia representando un contexto. Para que el plan sea aplicable, el contexto debe ser una consecuencia lógica de las actuales creencias del agente. El resto del plan es una secuencia de acciones básicas o subobjetivos que el agente tiene que lograr (o testear) cuando el plan es elegido para su ejecución.

Ejemplo:

+concierto (A,V) : Gusta (A)
<- !reservar_tickets (A,V).

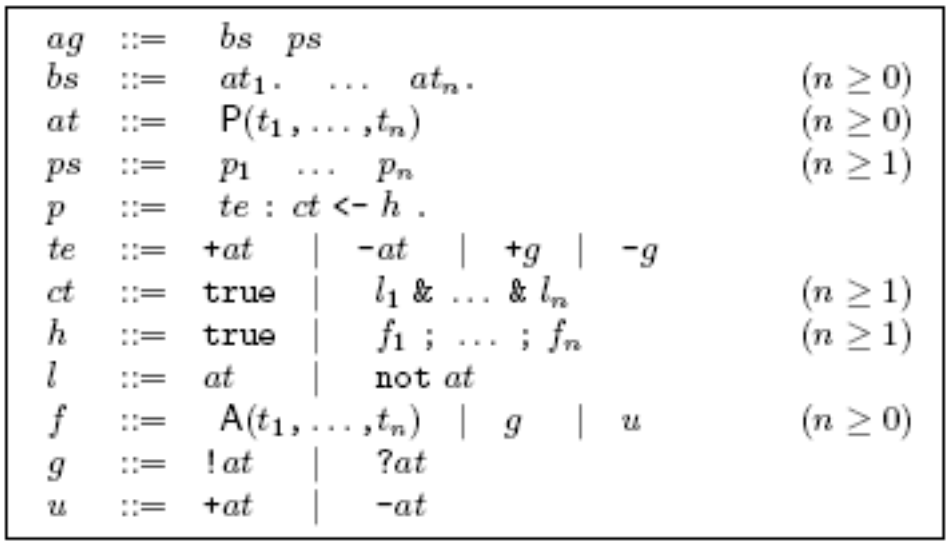
+!reservar_tickets(A,V) : ∼ ocupado (tel)
<-llamar (V).
...;

!escoger_asientos (A,V).

Aquí se muestra un ejemplo de un plan en AgentSpeak (L). Nos dice que cuando se obtiene el conocimiento de un concierto A que se va a desarrollar en el lugar V (representado por $concierto(A, V)$) y si gusta al agente el concierto A ($Gusta(A)$), entonces se tendrá como nuevo objetivo reservar tickets para este evento ($!reservar\_tickets(A, V)$). Luego, cuando el objetivo es agregado a los beliefs, lo que se tendrá es: si el teléfono no está ocupado, llamar a V (lugar de reserva), más otras acciones (denotadas por '...'), finalizadas por el subobjetivo de escoger los asientos.

3.1.2. Sintaxis del AgentSpeak(L)

La sintaxis de **AgentSpeak(L)** está definida por la gramática que se muestra en la siguiente Figura (3.1):



ag	::=	$bs \quad ps$	
bs	::=	$at_1. \quad \ldots \quad at_n.$	$(n \geq 0)$
at	::=	$\mathsf{P}(t_1, \ldots, t_n)$	$(n \geq 0)$
ps	::=	$p_1 \quad \ldots \quad p_n$	$(n \geq 1)$
p	::=	$te : ct \texttt{<-} h .$	
te	::=	$\texttt{+}at \mid \texttt{-}at \mid \texttt{+}g \mid \texttt{-}g$	
ct	::=	$\texttt{true} \mid l_1 \texttt{ \& } \ldots \texttt{ \& } l_n$	$(n \geq 1)$
h	::=	$\texttt{true} \mid f_1 \texttt{ ; } \ldots \texttt{ ; } f_n$	$(n \geq 1)$
l	::=	$at \mid \texttt{not } at$	
f	::=	$\mathsf{A}(t_1, \ldots, t_n) \mid g \mid u$	$(n \geq 0)$
g	::=	$\texttt{!}at \mid \texttt{?}at$	
u	::=	$\texttt{+}at \mid \texttt{-}at$	

Figura 3.1: Sintaxis de AgentSpeak
(Bordini & Hübner 2007)

Un agente **ag** es especificado como un conjunto de creencias **bs** (la base de creencia inicial) y un conjunto de planes **ps** (la librería de planes del agente). Las fórmulas atómicas **at** del lenguaje son predicados $P(t1, ..., tn)$ donde **P** es un símbolo de predicado, **A** es un símbolo de acción, y $t_1, ..., t_n$ son términos estándar de lógica de primer orden.

Un plan en **AgentSpeak (L)** esta definido por $p ::= te : ct < -h$ **p**, donde **te** es un evento disparador, **ct** es el contexto del plan y **h** es una secuencia de acciones, objetivos, o actualizaciones de creencias; **te:ct** es la cabeza del plan y **h** es el cuerpo. Un conjunto de planes está dado por **ps** como una lista de planes. Un evento disparador puede ser el agregado o eliminado de una creencia (+**at** y **-at** respectivamente), o el agregado o quitado de un objetivo (+**g** y **-g** respectivamente). Finalmente, el cuerpo del plan **h** puede

estar constituido por una acción $A(t_1, ..., t_n)$, un goal **g** o la actualización de las creencias **+at** y **-at**. En el capitulo 3.2 se verá en profundidad la sintaxis de JASON y las extensiones que proponemos en este documento.

3.1.3. Semántica Informal

El intérprete de **AgentSpeak (L)** también maneja un conjunto de eventos y un conjunto de intenciones que será la lista de acciones que el agente decide ejecutar. Su funcionamiento requiere tres funciones de selección. La **función de selección de eventos** (SE), que selecciona un sólo evento del conjunto de eventos. Una **función de selección de opciones** (SO), que selecciona un plan del conjunto de planes aplicables. Por último, una **función de selección de intenciones** (SI). Estas funciones selectoras pueden definirse según el comportamiento de cada agente que las utilice. Las intenciones son cursos de acciones que un agente se compromete a hacer para manejar cierto evento. Cuando un evento es interno, consecuencia de una acción ejecutada por un agente este es acompañado por la intención que lo generó. Al estar el agente interactuando con el ambiente a traves de sus percepciones recibirá nuevas creencias que serán eventos externos. Estos eventos externos crean nuevas intenciones, representando focos de atención separados.

Como se verá más adelante, los eventos internos son generados por el comportamiento interno del sistema o ejecución del mismo y los externos, son los eventos percibidos por los agentes.

La figura 3.2 describe cómo trabaja un intérprete de **AgentSpeak (L)**:

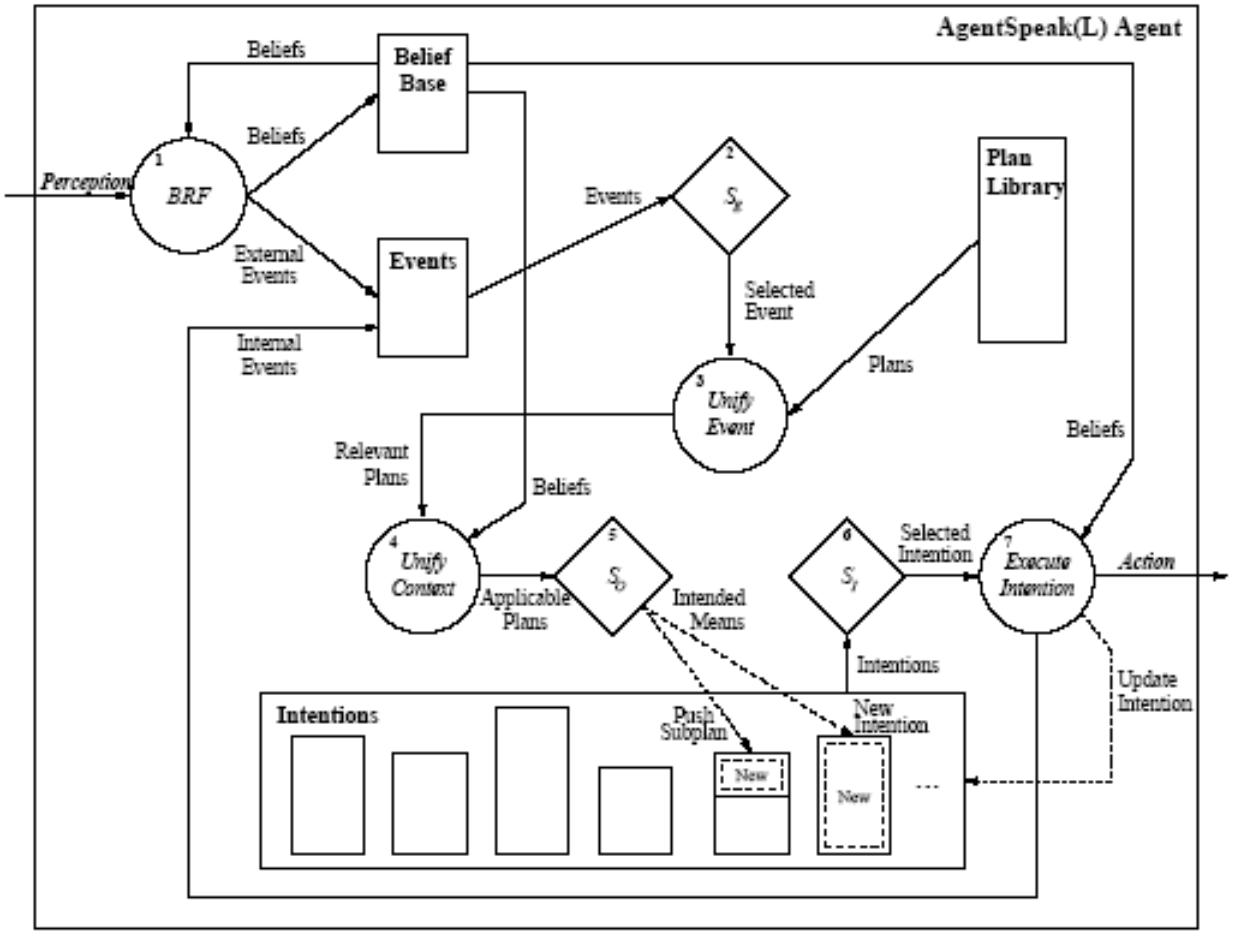


Figura 3.2: Diagrama del intérprete del agente AgentSpeak (L)
Fuente: Bordini & Hübner, 2007, p8.

En cada ciclo de interpretación de un programa de agente, **AgentSpeak (L)** actualiza la creencia del agente y con ello la lista de eventos, que pudo ser generado por una percepción del ambiente (evento externo) o por la ejecución de intenciones(acciones), lo cual produce un evento interno (ver en la figura (1)). Se asume que las creencias son actualizadas por las percepciones. La función de revisión de creencias no es parte del intérprete pero es un

componente necesario de la arquitectura del agente.

Después de que el selector de eventos (SE) haya seleccionado el evento (2), **AgentSpeak (L)** tiene que unificar este evento con eventos disparadores en la cabeza de los planes (3). Esto genera un conjunto de planes relevantes con sus respectivos contextos, se eliminan aquellos cuyos contextos no se satisfagan con la base de creencias del agente, para formar un conjunto de planes aplicables (4). Este conjunto de planes se denominan «opciones». Estas opciones se generarón a partir de un evento externo o interno. Entonces el selector de opciones SO elige uno de estos planes (5), luego pone este plan con alguna intención existente (si el evento fue interno) o crea una nueva intención (si el evento fue externo).

Todo lo que queda es seleccionar una intención (6) para ser ejecutada en el ciclo (7). La función de selección de intenciones (SI) se encarga de esto. La intención incluye un plan y la primer fórmula es tomada para su ejecución.

3.2. Gramática de Jason

A continuación se presenta la gramática actual de JASON y las diferencias entre la misma y la de AgentSpeak (L). (Bordini & Hübner 2007)

Siempre se deberá tener en cuenta que el estado de un agente, a lo largo de su ciclo de vida, está representado por un conjunto de creencias (beliefs) y un conjunto de planes (plans) y su comportamiento estará reflejado en el comportamiento de las funciones selectoras. Por lo cual se centrará el analisis de estas estructuras para no perder de foco el objetivo.

La sintaxis de JASON representada en BNF es la siguiente:

$$agent -> (init\_bels \mid init\_goals)^* plans \quad (3.1)$$

$$init\_bels -> beliefs\ rules \quad (3.2)$$

$$beliefs -> (literal \text{“.”})^* \quad (3.3)$$

$$rules -> (literal \text{“:-”} log\_expr \text{“.”})^* \quad (3.4)$$

$$init\_goals -> (\text{“!”} literal \text{“.”})^* \quad (3.5)$$

$$plans -> (plan)^* \quad (3.6)$$

$$plan -> [@atomic\_formula] triggering\_event [: context] [\text{“<-”} body] \text{“.”} \quad (3.7)$$

$$triggering\_event -> (\text{“+”} \mid \text{“-”}) [\text{“!”} \mid \text{“?”}] literal \quad (3.8)$$

$$literal -> [\sim] atomic\_formula \quad (3.9)$$

$$context -> log\_expr \mid \text{“}true\text{”} \quad (3.10)$$

$$log\_expr -> simple\_log\_expr \mid \text{“}not\text{”} log\_expr \mid log\_expr \text{“\&”} log\_expr \mid log\_expr \text{“|”} log\_expr$$
$$\mid \text{“(“}log\_expr\text{”)”} \quad (3.11)$$

$$simple\_log\_expr -> (literal \mid rel\_expr \mid < VAR >) \quad (3.12)$$

$$body -> body\_formula (\text{“;”} body\_formula)^* \quad (3.13)$$
$$\mid \text{“}true\text{”}$$

$$body\_formula -> (\text{“!”} \mid \text{“?”} \mid \text{“+”} \mid \text{“-”} \mid \text{“-+”}) literal \mid atomic\_formula \mid < VAR >$$
$$\mid rel\_expr \quad (3.14)$$

$$atomic\_formula- >(< ATOM > | < VAR >)[\text{“(”}list\_of\_terms\text{“)”}][\text{“[”}list\_of\_terms\text{“]”}] \quad (3$$

$$list\_of\_terms- >term(\text{“,”}term)^* \quad (3.16)$$

$$term- >literal|list|arithm\_expr| < VAR > | < STRING > \quad (3.17)$$

$$list- >\text{“[”}[term(\text{“,”}term)^*[\text{“|”}(list| < VAR >)]]\text{“]”} \quad (3.18)$$

$$rel\_expr- >rel\_term(\text{“} < \text{”}|\text{“} <= \text{”}|\text{“} > \text{”}|\text{“} >= \text{”}|\text{“} == \text{”}|\text{“}$$

$$== \text{”}|\text{“} = \text{”})rel\_term \quad (3.19)$$

$$rel\_term- >(literal|arithm\_expr) \quad (3.20)$$

$$arithm\_expr- >arithm\_term[(\text{“} + \text{”}|\text{“} - \text{”})arithm\_expr \quad (3.21)$$

$$arithm\_term- >arithm\_factor[(\text{“} * \text{”}|\text{“/”}|\text{“}div\text{”}|\text{“}mod\text{”})arithm\_term] \quad (3.22)$$

$$arithm\_factor- >arithm\_simple[\text{“} * *\text{”}arithm\_factor] \quad (3.23)$$

$$arithm\_simple- > < NUMBER > | < VAR > |\text{“} - \text{”}arithm\_simple$$

$$|\text{“(”}arithm\_expr\text{“)”} \quad (3.24)$$

Una de las diferencias más importante que distingue a JASON de la sintaxis de AgentSpeak (L) (ver Sección 3.1.2) es la inclusión de las anotaciones tanto en los beliefs como en los planes. Las anotaciones en los beliefs pueden verse en la regla 3.9 y los anotaciones en los planes en la regla 3.7.

Estas anotaciones están descriptas por (3.15):

$$atomic\_formula- > (< ATOM > | < VAR >)[\text{“(”}list\_of\_terms\text{“)”}][\text{“[”}list\_of\_terms\text{“]”}]$$

Dado que una fórmula atómica en JASON podría ser: [MIANOTACION] por lo tanto usando la regla (3.3) se puede generar un belief con anotación

como por ejemplo: X [MIANOTACION]. También son válidas en JASON fórmulas atómicas $p(t_1, \ldots, t_n)$, $n >= 0$, $\sim p(t_1, \ldots, t_n)$ donde '$\sim$' denota strong negation (negación fuerte). La negación por falla es usada en los contextos de los planes y se denota por 'not' precediendo un literal. El contexto es por lo tanto una conjunción de literales (regla 3.10). Los términos en JASON pueden ser variables, listas (estilo PROLOG), como también números enteros, de punto flotante o Strings (regla 3.17). Además, cualquier fórmula atómica puede ser tratada como un término y las variables como literales. Como en PROLOG, los operadores relacionales infijos son permitidos en los contextos de un plan y '_' es usada como variable anónima.

En JASON las fórmulas atómicas pueden contener anotaciones, esta es una lista de términos encerradas en corchetes inmediatamente siguiendo una fórmula. Dentro de una base de creencias (beliefs), las anotaciones son usadas, por ejemplo, para registrar las fuentes de la información, se verá en este trabajo que se utilizarán para representar un valor asociado al grado de creencia (belief degree) de que esa fórmula es cierta.

Debido a las anotaciones, las unificaciones se vuelven más complicadas. Cuando se intenta unificar una formula atómica con anotaciones at_1 $[s_{11}, \ldots, s_{1N}]$ con otra fórmula atómica at_2 $[s_{21}, \ldots, S_{2M}]$ no solamente at_1 debe unificar con at_2 sino que $\{s_{11}, \ldots, s_{1N}\}$ debe estar incluido $\{s_{21}, \ldots, S_{2M}\}$. Si por ejemplo, at_1 aparece en un contexto de un plan y at_2 en la base de belief. No solamente at_1 debe unificar con at_2 sino que todas las listas de términos de la anotación asociada a at_1 deben ser corroboradas por la anotación asociada a at_2.

Como caso más general, además de que un agente esté representado por un conjunto de beliefs iniciales y un conjunto de planes, los agentes pueden contener goals iniciales que pueden aparecer entre los beliefs iniciales de un agente. Los goals iniciales se representan como beliefs con un prefijo '!' y no pueden contener variables.

Otro cambio sustancial de JASON es que la base de belief puede contener reglas estilo PROLOG y la misma sintaxis usada en el cuerpo de una regla puede ser usada en el contexto de los planes.

Los operadores "+" y "-" en los cuerpos de los planes son usadas para poner o sacar beliefs que trabajan como "notas mentales" del agente mismo. El operador "+" agrega un belief después de remover, si existe, la primer ocurrencia del belief en la base de belief. Por ejemplo , $+a(X+1)$ primero remueve $a(\_)$ y agrega $a(X+1)$.

Las variables pueden ser usadas en lugares donde una formula atómica es esperada, incluyendo triggering events, contextos y cuerpos de los planes. En el contexto o el cuerpo también se puede ubicar una variable en vez de una formula atómica pero manteniendo las anotaciones explícitas, observar los siguientes ejemplos de unificación:

Sea X una variable y p una constante:

$X[a,b,c] = p[a,b,c,d]$ (unifican y X es p)

$p[a,b] = X[a,b,c]$ (unifican y X es p)

$X[a,b] = p[a]$ (no unifican) pues $[a,b]$ no está incluido en $[a]$

$p[a,b] = X[a]$ (no unifican) pues $[a,b]$ no está incluido en $[a]$

Notar que el término "true" en el contexto o el cuerpo de un plan (para

denotar contexto o cuerpo vacío) puede ser omitido y reescrito de la siguiente forma:

	Reescrito como:
$+e : c < -true.$	$+e : c.$
$+e : true < -!g.$	$+e < -!g.$
$+!e : true < -true.$	$+!e.$

Los planes tienen etiquetas representados por [$@atomic\_formula$] (3.7). Una etiqueta de un plan puede ser cualquier fórmula atómica, incluyendo anotaciones. Las anotaciones dentro de las etiquetas o labels de un plan pueden ser utilizadas para la implementación de funciones selectoras para planes más complejos.

Finalmente, las acciones internas pueden ser usadas tanto en el contexto como en el cuerpo de un plan. Cualquier símbolo de acción empezando con "." o conteniendo "." denota una acción interna. Las mismas son acciones definidas por el usuario las cuales son ejecutadas internamente por el agente. Se denominan internas para marcar una distinción con las acciones que aparecen en el cuerpo de un plan que denotan las acciones que un agente puede realizar para cambiar el entorno compartido.

Las anotaciones estructuralmente serán la base de la inclusión de los grados dentro de G-JASON que veremos más adelante. Funcionalmente se deberán cambiar tanto las unificaciones como las funciones selectoras para poder soportar los cambios sintácticos y semánticos de las "nuevas anotaciones" que se verán en el próximo capítulo.

Capítulo 4

G-JASON

A partir del conocimiento de la arquitectura de agentes que se pueden definir en JASON y con la inspiración del modelo g-BDI, se ha trabajado en esta tesina en una extensión de JASON que permita representar grados en las creencias (belief) y en los planes. Esto ha significado una extensión en su sintaxis y en su semántica. Al incluir los grados mencionados, para que los mismos tengan el valor semántico deseado, se tienen que modificar las funciones selectoras descriptas en las secciones 3.1.3 (ver figura 3.2 puntos 2, 5, 6) y los procesos de unificacioń para los eventos y los contextos de los planes (ver diagrama 3.2 puntos 3, 4).

Como los elementos fundamentales de JASON son los beliefs y los planes, se define una extensión graduada para estos elementos y cómo se utilizan para decidir la intención y la acción del agente.

1. Grados en las creencias (Beliefs)

   Se agregan grados en las creencias, representados por valores numéricos en el interválo [0,1] para representar el grado de certeza de que un hecho

sea cierto, para ello se utilizan las anotaciones. Luego, una creencia queda definida por:

$$X[degOfCert(valor_x)]$$

donde X representará una fórmula atómica (representando un hecho) y $valor_x$ su grado de certeza.

Por ejemplo, la creencia de que hay un viento leve con una medida de credibilidad de 0.7 se representará como: viento_leve [degOfCert(0.7)]. Este grado de creencia podrá representar distintas medidas de incertidumbre como por ejemplo la probabilidad. Este grado impactará en el orden de selección de los eventos debido a que la función selectora SE tomará el de mayor grado primero.

2. Grados en los Planes

   El plan en G-JASON está formalizado por:

   @label[planRelevance(gradoLabel)] te [degOfCert(gradoTE)]: ct [degOfCert(gradoCT)] <-body;

   Se permite asociar a cada plan tres valores: el grado en la anotación del label (gradoLabel), en el triggering event (gradoTE) y en el contexto (gradoCT) para que tengan un valor semántico o de carácter funcional.

   - El gradoLabel representa la medida de éxito del plan (planRelevance) en G-JASON. Este valor influye en la selección de intenciones y en la selección de opciones. De modo de que si dos planes

han pasado los filtros debido a las unificaciones se ejecutará el que tenga el «gradoLabel» más alto.

- El gradoTE es un grado aplicado al evento disparador e influye en la selección de eventos y en la unificación de eventos. Dependiendo de este grado se podrá considerar un plan como relevante una vez ejecutado el proceso de unificación de evento.
- El gradoCT es un grado en el contexto del plan e influirá en la unifación del contexto. Dependiendo de este grado se podrá considerar un plan como aplicable una vez ejecutado el proceso de unificación del contexto.

Tanto el gradoTE aplicado al evento disparador, como el gradoCT aplicado al contexto, representan precondiciones adicionales: en que grado estas creencias deben ser ciertas en la base actual para que el plan sea aplicable.

A continuación se describen estas funcionalidades de G-JASON con más detalle.

4.1. La semántica informal de G-JASON

Las modificaciones realizadas se van a describir siguiendo el flujo de la máquina de estados de JASON que se sigue utilizando en G-JASON planteando las distintas modificaciones en sus distintos componentes y procesos.

1. En el caso de los beliefs, G-JASON elegirá el evento que tenga mayor grado de creencia en la base para luego activar los planes que unifiquen

con estos eventos. Esta funcionalidad de la selección de eventos es realizada por la selectora de eventos en la cual también se deberá tener en cuenta los grados de los beliefs y los grados de los eventos (ver diagrama 3.2 punto 2).

2. Se extiende la unificación para los eventos (ver diagrama 3.2 punto 3) "si el trigger event de un plan (TE) unifica con un belief para que el plan se convierta en relevante debe también satisfacerse su grado":

   Si se tiene el belief en la base X[degOfCert(gradoB)] y un plan en la librería de planes con TE: X[degOfCert(gradoTE)]

   Debe ser $gradoB \geq gradoTE$ $(Regla1)$

   Por ejemplo, si se tiene en la base viento_leve[degOfCert(0.9)] y un plan en la librería de planes con TE: viento_leve[degOfCert(0.7)] este plan se considerará relevante y deberá considerarse en la unificación de contexto.

   Luego, no solamente deberán unificar los beliefs (versión original de JASON) sino que además deberán unificar las respectivas anotaciones (grados), en otro caso el plan no se considerará como relevante y será filtrado.

   Para esto se ha modificado el desarrollo del proceso de "Unificación de evento" que se verá en profundidad en la próxima sección.

3. En forma similar al punto anterior, se modificó el proceso de unificación de los contextos (ver diagrama 3.2 punto 4). Si el contexto de un

plan unifica con un belief de la base de creencias, para que el plan se convierta en aplicable debe también suceder que unifiquen los grados esto es:

Si se tiene el belief en la base Y[degOfCert(gradoB)] y un plan relevante con contexto: Y[degOfCert(gradoCT)]

Debe ser $gradoB \geq gradoCT \quad (Regla2)$

Por ejemplo, si se tiene en la base soleado[degOfCert(0.8)] y un plan relevante con contexto: soleado[degOfCert(0.7)] este plan se considerará aplicable.

En conclusión, no solamente deberán unificar los beliefs (versión original de JASON) sino que además deberán unificar las respectivas anotaciones (grados). Esto obligó a reformular la unificación esta vez para los contextos.

4. En el caso de las funciones selectora de opciones (SO) y la selectora de intenciones (SI) (ver diagrama 3.2 punto 5,6) para establecer tanto las opciones como las intenciones, G-JASON se fijará en los grados representados en las anotaciones de los labels de los planes (gradoLabel).

   @label[planRelevance(gradoLabel)] te [degOfCert(gradoTE)]: ct [degOfCert(gradoCT)] <-body;

   Cuando la máquina de estados selecciona sobre el conjunto de planes aplicables para crear las intenciones o actualizar las existentes, se basará en el plan de grado más alto que eligió la selectora de opciones (SO)

previamente. La intención creada o actualizada en el proceso será la que posee la opción asociada de mayor grado. En el caso que la máquina de estados desee ejecutar una acción, tomará la acción de la intención con el grado más alto que seleccionó la selectora de intenciones (SI) previamente. La acción ejecutada, será la contenida por la intención con mayor grado.

A continuación se verá como extender la sintaxis de JASON para obtener las nuevas funcionalidades. Luego se presentan las modificaciones soportadas por la nueva versión de JASON (G-JASON).

4.1.1. La Gramática de G-JASON

En la extensión G-JASON las palabras "planRelevance" (probabilidad de éxito de un plan) y "degOfCert" (grado de certeza de un hecho) serán palabras reservadas y los grados serán representados de manera flotante (en el intervalo [0,1]).

Se necesita para poder soportar los cambios planteados, modificar JASON redefiendo la sintaxis para los beliefs, planes, trigger event (te) y contextos (ct).

1. La formalización de los beliefs se extiende para permitir que tengan un grado de certeza:

   $beliefs->$ (**literalC** ".") $*$ (*en la sintaxis original* *regla* 3.3)

   $literalC->$ [" $\sim$ "]$atomic\_formula$ **anotC**

   $anotC->$ "[$degOfCert$(" $<valor>$ ")]"

$valor -> dig1, dig2$

$dig1 -> 0|1$

$dig2 -> 0|1|...|9$

2. Para agregar grados en el triggering event de cada plan se especializa la anotación para representar el grado de certeza:

   $triggering\_event -> (\ "+"\ |\ "-"\ )\ [\ "!"\ |\ "?"\ ]$ **literalC**
   (*en la sintaxis original* *regla* 3.8)

   Con lo cual se permite que el evento disparador (TE) en el plan, sea graduado.

3. Para el contexto se realiza el cambio de manera más directa reutilizando anotC: (definida en el punto 1)

   $context -> log\_expr$ **anotC** |"*true*" (*en la sintaxis original* *regla* 3.10)

4. Se agrega el grado(planRelevance) que se necesita dentro del plan para la selección y filtrado de las intenciones y opciones:

   $plan -> "@" atomic\_formula$ **anotG** $triggering\_event[: context][" <- " body]"."$
   (*en la sintaxis original* *regla* 3.7)

   $anotG -> "[planRelevance(" < valor > ")]"$

Con estas modificaciones realizadas en la sintaxis se está en condiciones de implementar en JASON las extensiones deseadas, se describe en la próxima sección el código de las extensiones y cómo impactarán en el nuevo comportamiento del agente.

4.2. Implementación de las extensiones. Modificaciones al código de Jason: G-JASON

Para poder lograr las extensiones se deben realizar modificaciones que afectan el tratamiento original de Jason tanto sobre los beliefs como sobre los planes. Las modificaciones realizadas se han implementado en las funciones selectoras. Ha sido también necesario modificar el proceso de unificación de evento y el proceso de unificación de contexto para poder aprovechar el valor de las anotaciones y los labels utilizados así como también poder unificar los grados agregados tanto en beliefs como en los planes. Se observa el tratamiento de los beliefs dentro del código, recordando que tanto los beliefs como los planes iniciales son parte de la representación de un agente en JASON. Como se vio en la sección 3.1.3, el funcionamiento de JASON puede ser representado por una máquina de estados (presentada en [Figura 3.2]). Se presentan los cambios propuestos en G-JASON según el orden en el ciclo de interpretación de un programa de agente, siguiendo el orden del flujo de ejecución en la máquina de estados.

4.2.1. Selección de eventos

Debido a que se requiere extender los beliefs para agregar grados, el grado en los beliefs del agente queda representado mediante el uso de "anotaciones" de JASON como se explicó en la sintaxis de G-JASON, sin modificar el código original.

Durante la ejecución del proceso de inicialización de estructuras en la

máquina de estados de JASON se cargan el conjunto de eventos y el library plan (conjunto de todos los planes). El conjunto de eventos se ha generado a partir de los beliefs definidos para el agente y la librería de planes se ha generado con todos los planes que contiene el agente. Tanto los planes como los beliefs iniciales para este caso están contenidos en un archivo de extensión *.asl, donde se define el estado inicial del agente (en el Anexo A se puede observar un ejemplo de este archivo).

1. Opción utilizando grados en los beliefs

   Se requiere que el grado en los beliefs influya en la ejecución de la máquina de estados JASON. Los beliefs generan los eventos que luego unificarán con los eventos disparadores de los planes. Por lo cual modificando el método "selectora de eventos" se logrará hacer influir directamente este grado en el comportamiento de la máquina. En este caso se eligió que el método "selectora de eventos" seleccione el evento con mayor grado de certeza (degOfCert).

   El código en la versión original de JASON ejecuta el método "*selectEvent*" sobre una estructura de tipo cola (input del método) y devuelve el primer evento que encuentra (retorna un evento). La nueva versión de *selectEvent* se puede observar en el Anexo B. La misma también recibe como parámetro una cola de eventos pero retorna el evento que posee el mayor grado de certeza (defOfCert).

   Si se generaron los eventos:
   $E1 = [+sol[degOfCert(0.8)]]$

$E2 = [+viento[degOfCert(0.7)]]$

La selectora retorna el evento E1 debido a que es el evento con mayor grado de certeza.

2. Activando prioridades

   Para manejar la reactividad del agente de una forma más directa, se agrega la posibilidad de activar un archivo «*Prioridades*» donde se ordenan los eventos según su prioridad. En cierta forma, cuando esta prioridad es mayor, los eventos serán más reactivos dado que la selectora de eventos los considerará primero en caso que este activada en el sistema la *prioridad*.

   Para el tratamiento de este archivo se ha modificado el proceso de selección de eventos nuevamente. Se determinará en estas propiedades el orden en el cual se seleccionarán los eventos, quedando sin efecto en este caso la selección de eventos observada anteriormente donde se selecciona el evento con mayor grado.

   Por ejemplo en una situación donde se requiere establecer el orden de importancia de tres hechos (gas, luz, vibración), donde gas es más importante que luz y luz más importante que vibración.

   Se establece este orden de precedencia en el archivo de *Prioridades* y no en la base de belief, dado que si la prioridad esta activa, el valor de precedencia en la selección de eventos lo fijará lo definido en el archivo.

   Es decir si se tiene en la base de belief de la representación del agente,

los hechos:

$vibracion[0, 9]$

$gas[gr0, 5]$

Pero en las prioridades el orden esta establecido por: $(gas, luz, vibracion)$.

Se seleccionará:

a) Si el archivo de prioridades está activo: se selecciona *gas* por el orden establecido en las prioridades.

b) Si no está activo: se selecciona *vibracion* por tener el grado más alto.

4.2.2. Proceso de unificación de eventos

Como se observa en la selección de eventos, los eventos se generan a partir de los beliefs iniciales definidos para el agente (por ejemplo en el archivo Agente.asl). Continuando con el ciclo de interpretación de un programa de agente, se procede a unificar el evento seleccionado por la selectora de eventos contra los eventos disparadores contenidos en los planes de la librería de planes. (library plan: se carga la estructura a partir del estado inicial del agente contenido en archivo de tipo *.asl). Si se tiene un *Plan P*:

$@PlanP{+}sensor\_termico(TEMP)[degOfCert(0.7)] : (TEMP < 300) <- encender\_alarma; llamar\_supervisor$,

y un evento E

$E = [+sensor\_termico(100)[degOfCert(0.8)]]$:

El evento E y el plan P unifican según la unificación original definida para JASON en el proceso de unificación de eventos. (sensor_termico(100) del evento unifica con sensor_termico(TEMP) del plan y quedará TEMP=100).

Se extendió la unificación debido al agregado de los grados en los eventos disparadores en la sintaxis de G-JASON (Caso 2 de la sintaxis de G-JASON). Con la extensión de la sintaxis se filtrarán los planes donde el grado del evento disparador del plan elegido sea menor al grado del evento. En el ejemplo el grado del evento E es 0.8 y el grado del evento disparador del plan P es 0.7 por lo tanto el plan P no será filtrado y será considerado de tipo relevante.

Para lo cual se extendió el método *protected boolean* $unifyTerms(Termt1g,$ $Termt2g)$ en la clase Unifier. Las nuevas versiones de los métodos se pueden observar en el Anexo E.

En conclusión se necesita un tratamiento especial cuando el functor es "$degOfCert$' y es agregado al evento disparador y se implementó esta modificación en el código.

4.2.3. Proceso de unificación de los contextos

El proceso de unficación de eventos arroja un conjunto de planes relevantes los cuales serán evaluados en el proceso de unificación de contexto. Originalmente en JASON se evalúa que el contexto sea verdadero.

Si se tiene un plan relevante P:

$@PlanP{+}sensor\_termico(TEMP)[degOfCert(0.7)] : (TEMP < 300) <$ $-encender\_alarma; llamar\_supervisor,$

y un belief B

$B = [+sensor\_termico(100)[degOfCert(0.8)]]$:

Como observamos en la unificación anterior TEMP tomara el valor 100, con lo cual la validación del contexto TEMP <300 será verdadera (100 <300).

Se necesita extender el proceso de unificación de contexto debido al agregado de los grados en el contexto como se ha visto en la sintaxis de G-JASON.

Como se planteó en G-JASON, se filtrará el plan relevante donde el grado del contexto sea menor al grado del belief. En el ejemplo anterior, el grado del belief B es 0.8 y el grado del contexto del plan relevante P es 0.6 por lo tanto el plan P no será filtrado y se considerará de tipo aplicable.

Se extendió nuevamente el método *protected boolean* $unifiesTerms()$ en la clase Unifier, pero esta vez para el tratamiento del contexto. Las nuevas versiones de los métodos se pueden observar en el Anexo E. En conclusión, se implementó un tratamiento especial cuando el functor es "$degOfCert$" y se agrega al contexto.

4.2.4. Selección de opciones

La unificación de contextos arrojará un conjunto de planes aplicables los cuales fueron filtrados en la extensión del proceso. Los planes aplicables serán el conjunto de entrada de la selección de opciones, la selectora de opciones elige uno de estos planes aplicables y luego vincula este plan con alguna intención existente (en caso de el evento fue interno, generado a partir de un belief) o se creará una nueva intención que contenga esta opción con su respectivo plan asociado. (3.1.3) Las opciones estarán representadas por los planes aplicables asociados.

En G-JASON se extendió los planes de JASON agregando grados representados por el uso de anotaciones en los "labels" de JASON sin modificar el código original como se explica en la sintaxis de G-JASON.

Se requiere que el grado en los planes aplicables influya en la ejecución de la máquina de estados JASON. Para lo cual se modificó el método "selectora de opciones" para que seleccione la opción con mayor posibilidad de éxito (planRelevance).

El código en la versión original de JASON ejecuta el método "*selectOption*" sobre una estructura de tipo lista (input del método) y devuelve la primer opción que encuentra (retorna una opción). La nueva versión de *selectOption* se puede observar en el Anexo C. La misma también recibe como parámetro una lista de opciones pero retorna la opción que posee el mayor grado de éxito (chanceOfSucess).

Luego, si se presentan las opciones:

P1=@$Plan1[planRelevance(0.85)]+sensor\_termico(TEMP)[degOfCert(0.7)]$: $(TEMP < 300)[degOfCert(0.6)] < -encender\_alarma; llamar\_supervisor$,

P2=@$Plan2[planRelevance(0.65)]+sensor\_termico(TEMP)[degOfCert(0.5)]$: $(TEMP < 100)[degOfCert(0.3)] < -encender\_calentador; encender\_turbina$,

La selectora retorna la opción P1 debido a que es la opción (plan) con mayor posibilidad de éxito (planRelevance).

Esta extensión es posible debido a los cambios realizados en la sintaxis de los planes (Caso 4 de la sintaxis de G-JASON).

4.2.5. Selección de intenciones

En la selección de opciones del proceso se crearon o actualizaron el conjunto de intenciones que posee el agente. El conjunto de intenciones será la entrada del método de selección de intenciones, la selectora de intenciones eligirá una intención la cual será ejecutada por el proceso de ejecución para terminar el ciclo de interpretación del agente. Cada intención estará representada por el plan que cada una de ellas contiene (plan aplicable que fue elegido como opción) y el cuerpo de ese plan contiene una lista de acciones para su posterior ejecución. El orden de ejecución de acciones de una intención se mantiene según la versión original de JASON.

Al igual que en la selección de opciones, la selectora de intenciones tendrá en cuenta el valor del grado en los planes representado por la anotación (*planRelevance*). Por lo cual modificando el método "selectora de intenciones" se logra hacer influir directamente este grado del plan en el comportamiento de la máquina de estados. En este caso se elige que el método "selectora de intenciones" se rija por seleccionar la intención con mayor posibilidad de éxito (planRelevance).

El código en la versión original de JASON ejecuta el método "*selectIntention*" sobre una estructura de tipo cola (entrada del método) y devuelve la primer intención que encuentra (retorna una intención). La nueva versión de *selectIntention* se puede observar en el Anexo D. La misma también recibe como parámetro una cola de intenciones pero retorna la intención que posee el mayor grado (chanceOfSucess).

Si se generaron las intenciones representadas por:

P1=@$PLAN1[planRelevance(0.85)]+sensor\_termico(TEMP)[degOfCert(0.7)]$: $(TEMP < 300)[degOfCert(0.6)] < -encender\_alarma; llamar\_supervisor$,

P2=@$PLAN2[planRelevance(0.65)]+temperarura\_horno(TEMP)[degOfCert(0.5)]$: $(TEMP < 100)[degOfCert(0.3)] < -encender\_ventiladores; abrir_escape$,

La selectora retornará la intención P1 debido a que es la que posee la mayor posibilidad de éxito.(planRelevance) Esta extensión es posible debido a los cambios realizados en la sintaxis de los planes (Caso 4 de la sintaxis de G-JASON).

Una vez seleccionada la intención, la máquina de estados ejecuta una de las acciones contenidas en el cuerpo del plan.

4.2.6. Caso de Estudio

En este capítulo se concluye con un caso de estudio que muestra el potencial de las extensiones anteriormente descriptas.

Se observarán tres formas de trabajo de un agente: JASON original, G-JASON y G-JASON con la configuración de prioridades activada.
Se desea modelar la supervisión de un horno rotativo utilizado para la fundición de metales. El horno además posee un motor que le permite girar lentamente alrededor de su eje principal. Este tipo de hornos es utilizado para la fundición de cobre, latón, bronce y aluminio. Se estará analizando constantemente tres variables fundamentales en la operación del horno:

- Temperatura del horno
- Presión de los gases en el interior del horno
- Vibración del motor de rotación

Para esta tarea se cuenta con tres sensores estrategicamente situados. En este contexto se implementará un agente de supervisión que contará con las lecturas de los tres sensores y a partir de las mismas deberá tomar acciones preventivas. Por lo tanto, los sensores abastecerán al agente de creencias sobre las lecturas y sus grados de certeza asociados los cuales dependerán de las precisiones de los instrumentos utilizados. Estas precisiones pueden variar de acuerdo al rango de valores de las variables en los cuales el horno esta trabajando. En base a las lecturas de los sensores y las acciones preventivas se modelarán los planes del agente. En un momento de tiempo t1, el horno se encuentra trabajando a una temperatura de 700 grados (B1), a una presión de 80 bars (B2) y a un nivel de vibración de 8 (B3). La precisión de las mediciones a esa temperatura del horno es del 70 % para las lecturas de temperatura, 90 % para las lecturas de presión y un 60 % para las lecturas de vibración. Los hechos sensados ingresarán de acuerdo a la sincronización de los lectores. El agente además posee acciones de supervisión de acuerdo a las lecturas de los sensores. Si la lectura de temperatura supera los 300 grados se deberá tomar la medida de encender el ventilador para que el horno comience a enfriarse, esta acción se encuentra modelada en el plan P1 (alerta temperatura). Si la temperatura supera los 600 grados, el horno comienza a operar en un estado crítico de temperatura por lo cual se debe

apagar el horno para lograr un enfriamiento total, esta acción se encuentra modelada en el plan P2 (urgencia de temperatura) y posee mayor relevancia que el plan P1. Para el caso de la variable presión, si la lectura supera los 35 bars se deberá cerrar una válvula de inyección del horno (plan P3) y si la presión supera los 70 bars se deberá encender una alarma general, dado el peligro de explosión en el horno y se deberán tomar medidas de precaución (plan P4). Las fallas de presión son las causas más frecuentes de accidentes por lo cual el control de esta variable es la más relevante para este sistema de horno. También se deberá supervisar el trabajo del motor de rotación del horno, a cierta cantidad de tiempo de trabajo empieza a sufrir el desgaste de sus piezas por lo cual comienza a producir un nivel elevado de vibración. Para evitar una rotura inminente del motor, a un nivel superior de vibración 8, se procede con el frenado del mismo (plan P5).

En los tres casos de ejemplo se utilizará una representación del agente que modele una situación del horno. El agente tendrá una base de creencias (belief) y un conjunto de planes iniciales.

1. Caso JASON original:

   La representación para el caso JASON original se observa a continuación:

   B1= $sensor\_termico(700)$

   B2= $sensor\_presion(80)$.

   B3= $sensor\_vibracion(8)$.

P1= $@alerta\_temperatura\ + sensor\_termico(TEMP)$:

$TEMP > 300\ < -prende\_ventilador.$

P2= $@urgencia\_temperatura\ + sensor\_termico(TEMP)$:

$TEMP > 600\ < -apagar\_horno.$

P3= $@alerta\_presion\ + sensor\_presion(PRES)$:

$PRES > 35\ < -cierra\_valvula.$

P4= $@urgencia\_presion\ + sensor\_presion(PRES)$:

$PRES > 70\ < -enciende\_alarma.$

P5= $@manejo\_vibracion\ + sensor\_vibracion(NVL)$:

$NVL > 5\ < -frena\_motor.$

Se observa en primer lugar, el comportamiento del agente en modo JASON original. Se debe recordar que en este caso no se tienen en cuenta los grados de creencias de las distintas mediciones relevadas por los sensores. Las creencias generan el conjunto de eventos y los planes iniciales forman la librería de planes del agente. Los eventos se evalúan en el orden que ingresan en el sistema. Los pasos de la ejecución para este caso se resumen en:

a) Se tomará el evento generado a partir de B1, dado que la selectora de eventos toma el primero.

b) El evento unifica con los eventos disparadores de los planes P1 y P2 con lo cual los convierte a los mismos en relevantes.

c) Los planes P1 y P2 unifican el contexto contra B1, con lo cual se generan dos opciones a partir de los planes.

d) La selectora de opciones seleccionará la primera opción que es la generada por P1 y se crea una intención.

e) Se ejecuta la acción de la intención generada a partir de P1: **prende_ventilador**.

f) Se selecciona el segundo evento que es el generado a partir de B2.

g) El evento unifica con los eventos disparadores de los planes P3 y P4 con lo cual los convierte en relevantes.

h) Los planes P3 y P4 unifican el contexto contra B2, con lo cual se generan dos opciones a partir de los planes.

i) La selectora seleccionara la opción de P3 por ser la primera y se crea una intención.

j) Se ejecuta la acción de la intención generada a partir de P3: **cierra_valvula**.

k) Se tomará el evento generado a partir de B3.

l) El evento unifica con el evento disparador de P5 y lo convierte en relevante.

m) El plan P5 unifica el contexto contra B3 y se agrega a los planes aplicables.

n) La selectora de opciones toma la primera entre las opciones: P5.

ñ) Se ejecuta la acción de la intención generada en base a P5: **frena_motor**.

Para este caso, el orden de las acciones resultantes a ejecutar por el agente es:

- prende_ventilador
- cierra_valvula
- frena_motor

Dado el orden de sincronización de los sensores, se ejecutarón la acción relacionada a la variable «temperatura» primero. Se ejecutó antes el plan de menor relevancia para el horno P1 sobre P3 consecuencia también del orden de ejecución de los planes en el comportamiento de JASON original. Además el plan con mayor relevancia para el agente (P4) no se ejecutó.

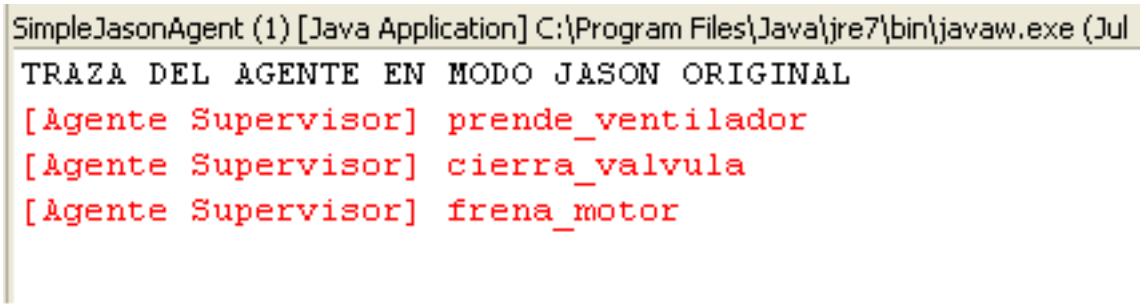


Figura 4.1: Traza de ejecución JASON original

2. Caso G-JASON:

A continuación se observa la ejecución del agente en modo G-JASON, por lo tanto se agregarán grados a la representación del agente, que permiten mayor expresividad permitiendo representar cualidades expuestas en el caso de estudio. Los grados en los beliefs (degOfCert) se agregan dependiendo de la precisión de medición que posee cada sensor. Por lo tanto cuanto más preciso sea el sensor, más precisa será su lectura y mayor será el grado de certeza que representa la creencia generada. En el caso de estudio se sabe que la precisión de los sensores de temperatura son de 70 % a una temperura de 700 grados por lo tanto el grado de certeza de la lectura de temperatura (B1) será de 0,7. Se realiza lo mismo con las lecturas de los sensores de presión (B2) y vibración (B3). Para los planes se agregarán grados (planRelevance) para dar relevancia a la necesidad de urgencia de ejecutar ciertas acciones por el agente supervisor, para poder manejarlo se agrega al plan su efectividad o grado del plan. Dada la relevancia de la presión para los hornos, se agregan los grados de relevancia más altos a los planes relacionados a la presión P3 (0,9) y P4 (0,85) respectivamente. En un segundo orden de relevancia estará modelada la temperatura (planes P1 y P2) y en último orden de relevancia se modelará la vibración (plan P5).

B1= $sensor\_termico(700)[degOfCert(0.7)]$

B2= $sensor\_presion(80)[degOfCert(0.9]$.

B3= $sensor\_vibracion(8)[degOfCert(0.6)]$.

P1= @$alerta\_temperatura[planRelevance(0.7)]$

$+sensor\_termico(TEMP)[degOfCert(0.5)]$:

$TEMP > 300[degOfCert(0.6)] < -prende\_ventilador.$

P2= @$urgencia\_temperatura[planRelevance(0.8)]$

$+sensor\_termico(TEMP)[degOfCert(0.5)]$:

$TEMP > 600[degOfCert(0.6)] < -apagar\_horno.$

P3= @$alerta\_presion[planRelevance(0.85)]+sensor\_presion(PRES)[degOfCert(0.7)]$:

$PRES > 35[degOfCert(0.8)] < -cierra\_valvula.$

P4= @$urgencia\_presion[planRelevance(0.9)]+sensor\_presion(PRES)[degOfCert(0.7)]$:

$PRES > 70[degOfCert(0.8)] < -enciende\_alarma.$

P5= @$manejo\_vibracion[planRelevance(0.6)]+sensor\_vibracion(NVL)[degOfCert(0.4)]$:

$NVL > 5[degOfCert(0.5)] < -frena\_motor.$

Los pasos de la ejecución para este caso se resumen en:

a) Se tomará el evento generado a partir de B2, dado que la selectora de eventos toma el evento con mayor grado. (degOfCert(0.9))

b) El evento unifica con los eventos disparadores de los planes P3 y P4 dado que el grado del belief B2 (degOfCert(0.9)) es mayor al de los eventos disparadores tanto del plan P3 ($sensor\_presion(PRES)[degOfCert(0.7)]$) como del plan P4 ($sensor\_presion(PRES)[degOfCert(0.7)]$), con lo cual los convierte a los mismos en planes relevantes.

c) Los planes P3 y P4 unifican el contexto contra B2 dado que el grado del belief B2 (degOfCert(0.9)) es mayor que el de los contextos de los planes tanto del plan P3 ($PRES > 35[degOfCert(0.8)]$) como del plan P4 ($PRES > 70[degOfCert(0.8)]$), con lo cual se generan dos opciones a partir de estos planes.

d) La selectora de opciones seleccionará la opción con mayor grado de plan, que es la generada por P4 ($@urgencia\_presion[planRelevance(0.9)]$) y se crea una intención.

e) Se ejecuta la acción de la intención generada a partir de P4: **enciende_alarma**.

f) Se selecciona el evento que posee el mayor grado, el evento relacionado a B1. (degOfCert(0.7))

g) El evento unifica con los eventos disparadores de los planes P1 y P2 dado que el grado del belief (degOfCert(0.7)) es mayor al de los eventos disparadores (degOfCert(0.5)) en ambos planes, con lo cual los convierte a los mismos en planes relevantes.

h) Los planes P1 y P2 unifican el contexto contra B1, se generan dos opciones a partir de los planes.

i) La selectora seleccionará la opción de mayor grado P2 (planRelevance(0.8)) entre (P2[planRelevance(0.8)], P1[planRelevance(0.7)]) y se crea una intención.

j) Se ejecuta la acción de la intención generada a partir de P2: **apaga_horno**.

k) Se tomará el evento que posee el mayor grado, el evento relacionado a B3.

l) El evento unifica con el evento disparador de P5 dado que el grado del belief B3(degOfCert(0.6)) es mayor al de el evento disparador (degOfCert(0.4)), con lo cual lo convierte en relevante.

m) El plan P5 unifica el contexto contra B3 y se agrega a los planes aplicables.

n) La selectora de opciones toma la única opción que resta P5 y genera una intención: P5[planRelevance(0.6)].

ñ) Se ejecuta la acción de la intención generada en base a P5: **frena_motor**.

Para este caso, el orden de las acciones resultantes a ejecutar por el agente es:

- enciende_alarma
- apaga_horno
- frena_motor

Se observa que la acción *enciende_alarma* es la primera en ejecutarse. Esta acción esta relacionada con el manejo de presión modelada por el belief B2 (grado de certeza 0,9) el cual es el más preciso. Además se ha ejecutado primero el plan P4 (relevancia de 0,9), por lo tanto también se ha ejecutado el plan más relevante. A diferencia del caso anterior,

el agente supervisor ha mejorado al atender primero las lecturas más precisas (sensor de presión) y ejecutar las acciones en el orden más relevante relacionadas a la lectura más precisa. En el caso anterior (JASON Original) se ejecuto la acción relacionada a la temperatura (lectura de menor precisión) dado el orden de llegada de las lecturas de los sensores.

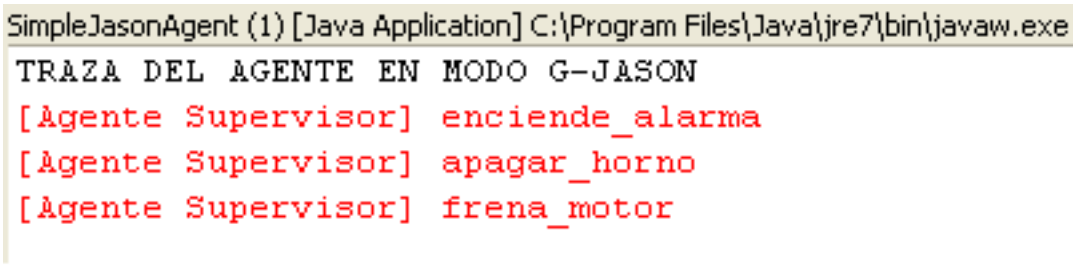


Figura 4.2: Traza de ejecución G-JASON

3. Caso G-JASON con prioridades

   Finalmente se observa el comportamiento del agente habilitando la configuración de las prioridades. Este archivo permite ordenar la importancia de los hechos sensados de forma independiente al orden que ingresan en el sistema por la sincronización de sensores (caso JASON original) o por el grado de certeza de los hechos sensados (caso G-JASON). El orden de tratamiento de los eventos será el que definan las «*prioridades*». Igualmente se seguirán utilizando los grados de certeza en los beliefs para los procesos de unificación de eventos y contexto. El archivo de propiedades *Prioridades.properties* contiene:

datosPrioridad = *true*

1 *sensor_vibracion*

2 *sensor_termico*

3 *sensor_presion*

Cada diezmil (**10000**) toneladas de operación del horno se determina que el desgaste del motor está en su punto máximo, por lo cual el manejo de la vibración es un problema crítico para el agente supervisor que debe tratarse prioritariamente. Suponiendo este escenario, *sensor_vibracion* tendrá la máxima prioridad sobre *sensor_termico* y *sensor_presion*. Los pasos de la ejecución para este caso se resumen en:

a) Se tomará el evento generado a partir de B3, dado que *sensor_vibracion* posee la máxima prioridad. (1)

b) El evento unifica con el evento disparador del plan P5 dado que el grado del belief (0.6) es mayor que el del evento disparador (0.4), con lo cual lo convierte en relevante.

c) El plan P5 unifica el contexto contra B3 dado que el grado del belief (0.6) es mayor que del contexto del plan (0.5), con lo cual se genera una opción a partir de este plan.

d) La selectora de opciones seleccionará la opción con mayor grado, que es la generada por P5 y se crea una intención.

e) Se ejecuta la acción de la intención generada a partir de P5: **frena_motor**.

f) Se selecciona el evento que posee la mayor prioridad, el evento relacionado a B1. (2)

g) El evento unifica con los eventos disparadores de los planes P1 y P2 dado que el grado del belief (0.7) es mayor al de los eventos disparadores (0.5), con lo cual los convierte a los mismos en relevantes.

h) Los planes P1 y P2 unifican el contexto contra B1, dado que el grado del belief (0.7) es mayor que el de los contextos de los planes (0.6), con lo cual se generan dos opciones a partir de los planes.

i) La selectora seleccionara la opción de mayor grado P2 (0.8) y se crea una intención.

j) Se ejecuta la acción de la intención generada a partir de P2: **apaga_horno**.

k) Se tomará el evento que posee la máxima prioridad, el evento relacionado a B2. (3)

l) El evento unifica con los eventos disparadores de los planes P3 y P4 dado que el grado del belief (0.9) es mayor al de los eventos disparadores (0.7), con lo cual los convierte en relevantes.

m) Los planes P3 y P4 unifican el contexto contra B2 dado que el grado del belief (0.9) es mayor que el de los contextos de los planes (0.8) y se agregan a los planes aplicables.

n) La selectora de opciones toma la opción con mayor grado entre las opciones: P3 y P4.

ñ) Se ejecuta la acción de la intención generada en base a P4: **enciende_alarma**.

Para este caso, el orden de las acciones resultantes a ejecutar por el agente es:

- frena_motor
- apaga_horno
- enciende_alarma

A pesar de que el sensor de vibración posee la menor precisión, dado que en este momento se determinó que la prioridad más alta la tiene el manejo de la vibración, se ejecuta la acción *"frena_motor"* relacionada a la máxima prioridad modelada. En este caso se puede observar la ventaja de poseer prioridades para modelar una situación crítica, de forma más reactiva por encima de la precisión de los sensores como en el caso de G-JASON. Se pudo definir gracias a las prioridades atender con máxima reactividad la lectura de vibración. Se observa como difiere el orden de ejecución de acciones para los tres diferentes casos. (JASON, G-JASON, G-JASON con prioridades).
Cono conclusión general, al utilizar G-JASON se logra mayor expresividad y mejores resultados (se atiende primero la lectura más precisa) dado que el orden de lectura y certeza es independiente al orden que se producen las lecturas de los mismos (JASON original). En el caso del uso de prioridades, se permitió dejar explícito las prioridades de atención de las lecturas, con

lo cual se logró mayor reactividad para el manejo de un evento crítico. (se analiza primero la lectura con mayor prioridad para el agente)

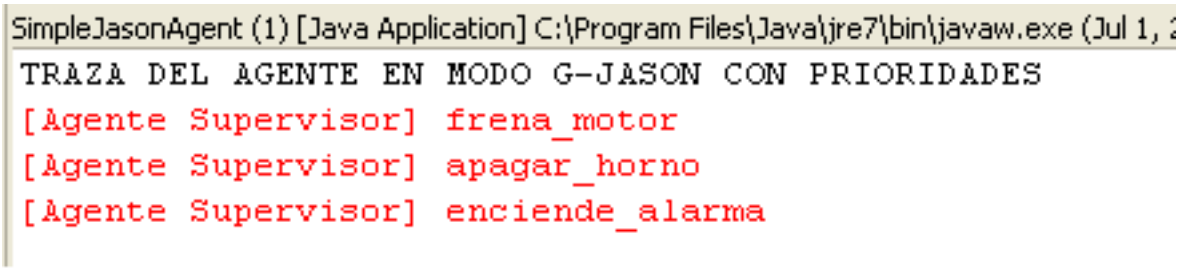


Figura 4.3: Traza de ejecución G-JASON con prioridades

Capítulo 5

Conclusiones y trabajo futuro

Se ha realizado una extensión de JASON tanto semántica como sintáctica para incluir grados en las creencias (beliefs) y en los planes (en distintos componentes). Se agregarón dos tipos de grados: grado de certeza (degOfCert) y relevancia de los planes (planRelevance). El grado de certeza se agregó en las creencias, en el evento disparador (TE) del plan y en el contexto de los planes. El grado de relevancia de los planes (planRelevance) aplica a todo el plan. También se ha agregado el concepto de prioridades para poder modelar situaciones que requieren mayor reactividad de las creencias. Al incluir las prioridades y grados mencionados, para que los mismos tengan el valor semántico deseado, se tuvieron que modificar las funciones selectoras y los procesos de unificacioń para los eventos y los contextos de los planes. Se implementaron todas las modificaciones necesarias, en una nueva versión de JASON que denominamos G-JASON por ser JASON graduado. A través de un caso de estudio se pudo comprobar que con esta extensión se aumenta la expresividad de JASON pudiendo representar distintas situaciones que

involucran incertidumbre en los hechos y diferentes relevancias de planes obteniendo mejores resultados que en JASON original.

En este prototipo se ha mostrado que estructuras de JASON se pueden utilizar y extender para introducir los grados en los distintos componentes básicos de un agente JASON (beliefs y planes). Aclaramos que a estos grados se le podría dar otra semántica y por lo tanto obtener otros comportamientos en los distintos agentes. Respecto al modelo BDI graduado que ha inspirado este trabajo, ha quedado pendiente modelar grados en los deseos (goals de JASON) ya que en la estructura de los sistemas PRS estos elementos no son considerados en los componentes básicos y no tienen un representación adecuada para modelizar grados de preferencias. La contribución de esta tesina ha sido presentada en el XIV Workshop de Agentes y Sistemas Inteligentes WASI - CACIC 2013 (Biga & Casali).

Apéndice A

Representación de un agente para JASON

Contenido de un archivo de tipo .asl interpretado por JASON:

$no\_llueve[degOfCert(1.0)]$.

$viento\_leve[degOfCert(0.8)]$.

$sol[degOfCert(0.9)]$.

$@lpasearrio[chanceOfSucess(0.7)] + sol[degOfCert(0.7)] :$

$viento\_leve[degOfCert(0.8)] < -do(50); .print(invito\_amigos\_rio)$.

$@ljugarfutbol[chanceOfSucess(0.8)] + no\_llueve[degOfCert(1.0)] :$

$viento\_leve[degOfCert(0.5)] < -do(60); .print(invito\_amigos\_futbol)$.

Apéndice B

Código JAVA de la nueva versión de la selectora de eventos

```
public Event selectEvent(Queue<Event>events) {
        // make sure the selected Event is removed from 'events' queue
        /*
        * @Author:Adrian Biga
        * En la version original:El evento es extraido de la cola de eventos, la funcion pool retorna
        * y remueve la cabeza de la cola.No tiene en cuenta las anotaciones de los beliefs para
        * esto.
        *
        */
        Iterator<Event>it=events.iterator();
        while(it.hasNext()){
                Event e = it.next();
                System.out.println("Selectora de eventos (literales): "
```

```
                + e.getTrigger().getLiteral().getAnnots());
            if(datosUmbral == true){
                ordenarListaEventosUmbral(e);
            }else{
                ordenarListaEventos(e);
            }
        }
        //return events.poll();(version anterior)
        Event selected = selectEventN(events);
        events.remove(selected);
        return selected;

}


private void ordenarListaEventos (Event e){
    ListTerm l = e.getTrigger().getLiteral().getAnnots();
    List <Term>lis = l.getAsList();
    Iterator <Term>it =lis.iterator();
    Double f = -1.0;
    while(it.hasNext()){
        Term t = it.next();
        String s = t.toString();
        System.out.println("Terminos del evento "+s);
```

```
            int i = s.indexOf("degOfCert");
            //encontro la parte del termino que posee "degOfCert"
            if(i!=-1 ){
                if(s.charAt(i+10) != '1'){
                    String sub = s.substring(i+10,i+13);
                    f =new Double(sub);
                    this.mapEvents.put(e, f);
                }
                else {
                    this.mapEvents.put(e, 1.0);
                }
            }
        }
        if(f.equals(-1.0)){
                this.mapEvents.put(e, 1.0);
        }
}


private Event selectEventN(Queue<Event>events){
        Iterator<Event>it = events.iterator();
        Event temp = null;
        Event max = null;
        if(it.hasNext()){
```

```
        temp =it.next();
        max = temp;
    }
    while(it.hasNext()){
            Event e =it.next();
            if(mapEvents.get(e) >mapEvents.get(temp)){
                max = e;
                temp = max;
            }
        }
        System.out.println("Evento con el grado mas alto:" + max.toString());
        return max;
    }
```

Apéndice C

Código JAVA de la nueva versión de la selectora de opciones

```
public Option selectOption(List<Option>options) {
    /*
        * @Author:Adrian Biga
        * La opcion es extraida de una lista y "remove" la retorna.No tiene en
        * cuenta las anotaciones ni los labels para esto.
        *
        */
    if (options != null && !options.isEmpty()) {
            Iterator<Option>it=options.iterator();
            while(it.hasNext()){
                Option o = it.next();
                System.out.println("Selectora de opciones (literales):" +
                    o.getPlan().toString() + "..." + o.getUnifier().toString());
```

```
                    ordenarListaOpciones(o);
            }
                //return options.remove(0);
                    Option selected = selectOptionN(options);
                    options.remove(selected);
                    return selected;
        } else {
            return null;
        }
}


private void ordenarListaOpciones (Option o){
    //Obtengo el predicado del plan
    // Pred pred = o.getPlan().getLabel();
        System.out.println("Anotaciones del label del plan para Opciones: "+
                o.getPlan().getLabel().getAnnots());
        ListTerm l = o.getPlan().getLabel().getAnnots();
        List <Term>lis = l.getAsList();
        Iterator<Term>it =lis.iterator();
        Double f = -1.0;
        while(it.hasNext()){
            Term t = it.next();
            String s = t.toString();
```

```
                System.out.println("Terminos de la opcion: "+s);
                //int i = s.indexOf("degOfCert");
                int i = s.indexOf("planRelevance");
                //encontro la parte del termino que posee "chanceOfSucess"
                if(i!=-1 ){
                    if(s.charAt(i+15) != '1'){
                        String sub = s.substring(i+15,i+18);
                        f =new Double(sub);
                        this.mapOptions.put(o, f);
                    }
                    else {
                        this.mapOptions.put(o, 1.0);
                    }
                }
            }
            if(f.equals(-1.0)){
                        this.mapOptions.put(o, 1.0);
            }
        }


private Option selectOptionN(List<Option>options){
        Iterator<Option>it = options.iterator();
        Option temp = null;
```

```
        Option max = null;
        if(it.hasNext()){
            temp =it.next();
            max = temp;
        }
        while(it.hasNext()){
                Option e =it.next();
                if(mapOptions.get(e) >mapOptions.get(temp)){
                    max = e;
                    temp = max;
                }
        }
        System.out.println("Opcion con el grado mas alto:" + max.toString());
        return max;
}
```

Apéndice D

Código JAVA de la nueva versión de la selectora de intenciones

```
public Intention selectIntention(Queue<Intention>intentions) {
    // make sure the selected Intention is removed from 'intentions'
    // and make sure no intention will "starve"!!!
    //return intentions.poll();
    /*
      * @Author:Adrian Biga
      * La intencion es extraida de una cola y "poll" la retorna.No tiene en
      * cuenta las anotaciones ni los labels para esto.
      *
      */
      Iterator<Intention>it=intentions.iterator();
          while(it.hasNext()){
                Intention i = it.next();
```

```
                System.out.println("Selectora de intenciones (como termino):"
                  + i.getAsTerm());
                ordenarListaIntenciones(i);
            }
            Intention selected = selectIntentionN(intentions);
            intentions.remove(selected);
            return selected;
}


private void ordenarListaIntenciones(Intention i){
    Term ter = i.getAsTerm();
    String sTerm = ter.toString();
    Double temp = 0.0;
    Double max = 0.0;
    Double f = -1.0;
    System.out.println("Terminos de la intencion: "+ sTerm);
       //Puede aparecer mas de una ves ,por lo cual nos tenemos que quedar con el maximo
    int j = sTerm.indexOf("planRelevance");
    String sub = sTerm.substring(j+15,j+18);
    if(j == -1){
        this.mapIntentions.put(i,1.0);
    }
    else {
```

```
if(j!=-1 ){
    if(sTerm.charAt(j+15) != '1'){
        f =new Double(sub);
        //this.mapOptions.put(o, f);
        temp = f;
        max = temp;
    }
    else {
        temp = 1.0;
        max = temp;
        //this.mapOptions.put(o, 1.0);
    }
}
int index = j + 15;
while(sTerm.indexOf("chanceOfSucess",index)!= -1){
    j = sTerm.indexOf("chanceOfSucess",index);
    if(sTerm.charAt(j+15) != '1'){
        f =new Double(sub);
        if(f >temp){
           max = f;
           temp = max;
        }
    }
    else {
```

```
                    if(1.0 >temp){
                       max = 1.0;
                       temp = max;
                    }
                    //this.mapOptions.put(o, 1.0);
                }
            index = j + 15;
            }
        }
        this.mapIntentions.put(i,max);
}


private Intention selectIntentionN(Queue <Intention>intentions){
    Iterator<Intention>it = intentions.iterator();
    Intention temp = null;
    Intention max = null;
    if(it.hasNext()){
       temp =it.next();
       max = temp;
    }
    while(it.hasNext()){
            Intention e =it.next();
            if(mapIntentions.get(e) >mapIntentions.get(temp)){
```

```
                    max = e;
                    temp = max;
                }
            }
            System.out.println("Intencion con el grado mas alto:" + max.toString());
            return max;
        }
```

Apéndice E

Código JAVA de la nueva versión de la unificación de eventos y de contextos

```
protected boolean unifyTerms(Term t1g, Term t2g) {
    // if args are expressions, apply them and use their values
    if (t1g.isArithExpr()) {
        t1g = (Term) t1g.clone();
        t1g.apply(this);
    }
    if (t2g.isArithExpr()) {
        t2g = (Term) t2g.clone();
        t2g.apply(this);
    }
    final boolean t1gisvar = t1g.isVar();
    final boolean t2gisvar = t2g.isVar();
    // both are vars
```

```
if (t1gisvar && t2gisvar) {
        VarTerm t1gv = (VarTerm) t1g;
        VarTerm t2gv = (VarTerm) t2g;
        // get their values
        Term t1vl = function.get(t1gv);
        Term t2vl = function.get(t2gv);
        // if the variable value is a var cluster, it means it has no value
        if (t1vl instanceof VarsCluster)
            t1vl = null;
        if (t2vl instanceof VarsCluster)
            t2vl = null;
        //both has value, their values should unify
        if (t1vl != null && t2vl != null) {
            return unifiesNoUndo(t1vl, t2vl);
        }
        // only t1 has value, t1's value should unify with var t2
        if (t1vl != null) {
            return unifiesNoUndo(t2gv, t1vl);
        }
        // only t2 has value, t2's value should unify with var t1
        if (t2vl != null) {
            return unifiesNoUndo(t1gv, t2vl);
        }
        // both are var with no value, like X=Y
```

```
        // we must ensure that these vars will form a cluster
        //if (! t1gv.isUnnamedVar() && ! t2gv.isUnnamedVar()) {
            VarTerm t1c = (VarTerm) t1gv.clone();
            VarTerm t2c = (VarTerm) t2gv.clone();
            VarsCluster cluster = new VarsCluster(t1c, t2c, this);
            if (cluster.hasValue()) {
                // all vars of the cluster should have the same value
                for (VarTerm vtc : cluster) {
                    function.put(vtc, cluster);
                }
            }
//}
return true;
}
// t1 is var that doesn't occur in t2
if (t1gisvar) {
        VarTerm t1gv = (VarTerm) t1g;
        // if t1g is not free, must unify values
        Term t1vl = function.get(t1gv);
        if (t1vl != null && !(t1vl instanceof VarsCluster))
            return unifiesNoUndo(t1vl,t2g);
        else if (!t2g.hasVar(t1gv))
            return setVarValue(t1gv, t2g);
        else
```

```
            return false;
        }
// t2 is var that doesn't occur in t1
if (t2gisvar) {
        VarTerm t2gv = (VarTerm) t2g;
        // if t1g is not free, must unify values
        Term t2vl = function.get(t2gv);
        if (t2vl != null && !(t2vl instanceof VarsCluster))
            return unifiesNoUndo(t2vl,t1g);
        else if (!t1g.hasVar(t2gv))
            return setVarValue(t2gv, t1g);
        else
            return false;
}
// both terms are not vars
// if any of the terms is not a structure (is a number or a
// string), they must be equal
if (!t1g.isStructure() —— !t2g.isStructure())
        return t1g.equals(t2g);
// both terms are structures
Structure t1s = (Structure)t1g;
Structure t2s = (Structure)t2g;
// different arities
final int ts = t1s.getArity();
```

```
if (ts != t2s.getArity())
        return false;
final boolean t1islit = t1g.isLiteral();
final boolean t2islit = t2g.isLiteral();
final boolean t1isneg = t1islit && ((Literal)t1g).negated();
final boolean t2isneg = t2islit && ((Literal)t2g).negated();
// if both are literal, they must have the same negated
if (t1islit && t2islit && t1isneg != t2isneg)
        return false;
// if one term is literal and the other not, the literal should not be negated
if (t1islit && !t2islit && t1isneg)
        return false;
if (t2islit && !t1islit && t2isneg)
        return false;
// if the first term is a predicate and the second not, the first should not have annots
if (t1g.isPred() && !t2g.isPred() && ((Pred)t1g).hasAnnot())
        return false;
// different functor
/* AB
* Hay que modificar porque trabajaremos con distintos functor
*/
if (t1s.getFunctor() != null && !t1s.getFunctor().equals(t2s.getFunctor())
        && (!t1s.getFunctor().equals("planRelevance")
        —— !t2s.getFunctor().equals("degOfCert")))
```

```
        return false;
// unify inner terms
// do not use iterator! (see ListTermImpl class)
for (int i = 0; i <ts; i++)
        //AB: Modificamos la  unificacion
        if(t1s.getFunctor().equals(t2s.getFunctor())
          && t1s.getFunctor().equals("degOfCert")){
          Float f1= new Float(t1s.getTerm(i).toString());
          Float f2= new Float(t2s.getTerm(i).toString());
          if(f2 <f1)
                    return false;
          }else if(t1s.getFunctor().equals("chanceOfSucess")
                    && t2s.getFunctor().equals("degOfCert")) {
          /AB:Modificacion para los contextos
                    Float f1= new Float(t1s.getTerm(i).toString());//grado en contexto
                    Float f2= new Float(t2s.getTerm(i).toString());//grado en belief
                    if(f2 <f1)
                              return false;
          }
else {
        if (!unifiesNoUndo(t1s.getTerm(i), t2s.getTerm(i)))
                    return false;
}
// if both are predicates, the first's annots must be subset of the second's annots
```

```
¿if (t1g.isPred() && t2g.isPred())
        if ( ! ((Pred)t1g).hasSubsetAnnot((Pred)t2g, this))
                return false;
return true;
}
```

Apéndice F

Paso a paso en detalle de la máquina de estados de JASON

Primera iteración de la máquina:

A) El primer método que se ejecuta es $TransitionSystem.\ applyProcMsg()$ este método esta más relacionado a los mensajes, es decir, a la comunicación entre agentes por lo cual en este análisis que aplica a un sólo agente no ejecuta nada.

B) Automáticamente se pasa al estado de la máquina $SelEv$, con lo cual se ejecuta posterioriormente el metodo $TransitionSystem.applySelEv()$, se puede ver el status actual de "Circumstance" (estructura interna del agente que define el estado) del agente en ese momento:

Circumstance:
$E = [+x(2)[source(self), degOfCert(0.7)], +x(0)[source(self), degOfCert(0.9)],$
$+x(1)[source(self), degOfCert(0.5)]]$
$I = [\,]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = null$

$SO = null$

$SI = null$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,]$

Por lo cual se observa que el conjunto de eventos ya está cargado producto de los beliefs definidos en el agente. Siguiendo con el método dentro de la misma se seleccionará uno de los mismos.

Como existe un evento, se pasa al estado $FindOp$, en otro caso se hubiera pasado directamente al estado $ProcAct$. Esto sucede porque no se encuentra customizado el metodo "*selectOption*" (Selectora de Opciones). (Esto no debería suceder porque necesitamos que se haga un tratamiento especial de los planes). Con lo cual se esta salteando tres pasos de la máquina de estado enumerados debajo:

1. case $RelPl$: $applyRelPl(); break;$

   Este caso nos seleccionará un conjunto de planes relativos generados por el método $TransitionSystem.relevantPlans$ los cuales dependen

del TE elegido por la *selectEvent*. (Selectora de Eventos)

2. case $ApplPl$: $applyApplPl(); break;$

   Este caso seleccionará un conjunto de planes aplicables generados por el método $TransitionSystem.applicableplans$ a partir de los planes relativos y pondrá el estado como $SelAppl$.

3. case $SelAppl$: $applySelAppl(); break;$

   Este caso arrojará una "opción" producto de la función *selectOption* dependiendo del conjunto de los planes aplicables y pasará al estado $ProcAct$ si no existe opción o pasará al estado $AddIM$ si hay una opción.

Se observará a continuación el funcionamiento del método $applyFindOp()$ que opera en base al método $planLibrary.getCandidatePlans$ dependiendo del evento seleccionado por la selectora de eventos. (en la primera iteración el evento será $+x(0)[source(self),\ degOfCert(0.9)]$) Esta función trabaja sobre esta lista de planes candidatos:

$[@l\_\_0[source(self)]+x(N) : (N >= 3) < -do(50); .print(end1)., @l\_\_1[source(self)]+$
$x(N)[degOfCert(0.7)] : (N < 3) < -do(50); .print(end4)., @l\_\_2[source(self)]+$
$x(N) : (N < 3) < -do(0); .print(end2)., @degOfCert[degOfCert(0.7), source(self)]+$
$x(N) : (N < 3) < -do(20); .print(end3)., @l\_\_3[source(self)] + x(N) : (N <$
$3) < -do(60);\ .print(end5).]$

Se observará en todos los planes que se agregan en el input ASL, un label genérico agregado por Jason para dar formato: $@l\_\_3[source(self)]$ por otro

lado se puede observar que el label agregado en el input es mantenido y no modificado. ($@degOfCert[degOfCert(0.7)$)

La primer iteración dentro de los planes candidatos al realizarse la unificación y evaluar N toma el valor 0.($valor\ de\ x\ en\ el\ evento$). Por lo cual el primer plan es descartado y no se transforma en una opción. (Recordar que los contextos deben evaluar a verdadero y 0 no es mayor o igual que 3).

En la segunda iteración se observa cómo si la cumple y es generada la opción. (ver el SO)

Circumstance:

$E = [+x(2)[source(self), degOfCert(0.7)], +x(1)[source(self), degOfCert(0.5)]]$

$I = [\,]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(0)[source(self), degOfCert(0.9)]$

$SO = (@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)., \{N = 0\})$

$SI = null$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,].$

Al haberse generado la opción, pasa automáticamente a $AddIM$. (al encontrar una opción pasa directamente al otro estado) Se ejecutará $applyAddIM()$ (IM: Intention Mean) dentro de la misma se tiene en cuenta la opción seleccionada y el evento. Por ejemplo la opción: $(@l\_\_2[source(self)] + x(N) : (N < 3) < -do(0); .print(end2)., \{N = 0\})$ es el plan seleccionado más la unificación de N con el valor 0 (unificador) y el evento: $+x(0)[source(self), degOfCert(0.9)]$. Como no se tiene ninguna Intention creada se procede a la creación de la misma. Se observa como queda Circunstance:

Circumstance:

$E = [+x(2)[source(self), degOfCert(0.7)], +x(1)[source(self), degOfCert(0.5)]]$

$I = [@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./\{N = 0\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(0)[source(self), degOfCert(0.9)]$

$SO = (@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)., \{N = 0\})$

$SI = null$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,]$

Y pasa al estado $ProcAct$ donde se ejecutará la opción $applyProcAct()$ su funcionalidad está relacionada con las acciones, revisa la estructura feedback actions (FA) en Circumstance y define el estado $SelInt$.

Se ejecuta la función $applySelInt$, la misma observa si Circumstance posee Intentions y ejecuta la selectora $selectIntention$ para seleccionar una.

$(@l\_\_2[source(self)] + x(N) : (N < 3) < -do(0); .print(end2)./\{N = 0\}enestecaso)$

Como posee intenciones, toma el estado $ExecInt$ sino volverá al estado inicial $StartRC$. La función que se ejecutará será $applyExecInt$ donde fijará que tipo de regla posee cada acción. Por ejemplo, para el plan elegido se tienen dos acciones: $do(0); .print(end2)$. En la función se observarde una acción a la vez, se evalua $do(0)$ que es de tipo "action".

Circumstance:

$E = [+x(2)[source(self), degOfCert(0.7)], +x(1)[source(self), degOfCert(0.5)]]$

$I = [\,]$

$A =< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, false >$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(0)[source(self), degOfCert(0.9)]$

$SO = (@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)., \{N = 0\})$

$SI = @l\_2[source(self)]{+}x(N) : (N < 3) < -do(0); .print(end2)./\{N = 0\}$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,]$.

En este paso la Circumstance en el conjunto A se tendrá una nueva tupla $< accion, intencion >$ y en este caso tomará el estado por defecto $ClrInt$. Se ejecutará $applyClrInt$ donde se le pasa la intención. Esta función es para remover la intención de SI. Se remueve la misma si esta en estado finalizado, es decir todas las acciones fueron ejecutadas. A continuación se sale del bucle y se procederá a ejecutar la acciones pendientes. Se ejecuta $do(0)$ que es la acción qye había quedado registrada en A y Circumstance queda de la siguiente manera. (Observar que cambio PA y FA)

Circumstance:

$E = [+x(2)[source(self), degOfCert(0.7)], +x(1)[source(self), degOfCert(0.5)]]$

$I = [\,]$

$A =< do(0), @l\_2[source(self)]{+}x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(0)[source(self), degOfCert(0.9)]$

$SO = (@l\_\_2[source(self)]{+}x(N) : (N < 3) > -do(0); .print(end2)., \{N = 0\})$

$SI = @l\_\_2[source(self)]{+}x(N) : (N < 3) < -do(0); .print(end2)./\{N = 0\}$

$AI = null$

$PA = \{1 =< do(0), @l\_\_2[source(self)]{+}x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >\}$

$PI = \{\}$

$FA = [< do(0), @l\_\_2[source(self)]{+}x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >].$

Segunda iteración:

El bucle comenzará a ejecutarse de nuevo (*reasoningCycle*). Se comienza reseteando Circumstance en los siguientes conjuntos como en la primer iteración: A, RP, AP, SI, SO, SE. Con lo que queda:

Circumstance:

$E = [+x(2)[source(self), degOfCert(0.7)], +x(1)[source(self), degOfCert(0.5)]]$

$I = [\,]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = null$

$SO = null$

$SI = null$

$AI = null$

$PA = \{1 =< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >\}$

$PI = \{\}$

$FA = [< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >].$

y se comienza a ejecutar la máquina de estados nuevamente (ver el switch que describe a la máquina). En esta segunda iteración se resaltará solamente lo que más impacta a lo relativo a planes por lo cual se observa del estado *FindOp* en adelante.

En esta segunda iteración se comienza con el status de Circumstance:

Circumstance:

$E = [+x(1)[source(self), degOfCert(0.5)]]$

$I = [\,]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(2)[source(self), degOfCert(0.7)]$

$SO = null$

$SI = null$

$AI = null$

$PA = \{1 =< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >\}$

$PI = \{\}$

$FA = [< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$

$\{N = 0\}, true >]$.

Se observará de nuevo la función $applyFindOp()$, en base a la función $PlanLibrary.\ getCandidatePlans$ dependiendo del evento seleccionado por SE. (en la segunda iteración será $+x(2)$ $[source(self), degOfCert(0.7)]$) Esta función trabaja sobre esta lista de planes, la misma que en la primera iteración:

$[@l\_\_0[source(self)]+x(N) : (N >= 3) < -do(50); .print(end1)., @l\_\_1[source(self)]+$
$x(N)[degOfCert(0.7)] : (N < 3) < -do(50); .print(end4)., @l\_\_2[source(self)]+$
$x(N) : (N < 3) < -do(0); .print(end2)., @degOfCert[degOfCert(0.7), source(self)]+$
$x(N) : (N < 3) < -do(20); .print(end3)., @l\_\_3[source(self)] + x(N) : (N <$
$3) < -do(60);\ .print(end5).]$

La primer corrida dentro de los planes candidatos como al unificar y evaluar N toma el valor 2. Por lo cual el primer plan es descartado y no se transforma en una opción. El plan elegido esta vez es $@l\_\_1[source(self)]$ $+x(N)[degOfCert(0.7)] : (N < 3) < -do(50)$. Debido a que se unificaron las anotaciones.

Anotación en el evento:

$[source(self),\ degOfCert(0.7)]$

Anotación en el plan TE's:

$[degOfCert(0.7)]$ (Observar el SO)

Circumstance:

$E = [+x(1)[source(self), degOfCert(0.5)]]$

$I = [\,]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(2)[source(self), degOfCert(0.7)]$

$SO = (@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$
$.print(end4)., \{N = 2\})$

$SI = null$

$AI = null$

$PA = \{1 =< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >\}$

$PI = \{\}$

$FA = [< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >].$

Al haberse generado la opción se pasa automáticamente a $AddIM$. En resumen al encontrar una opción pasa directamente al otro estado. Se ejecutará $applyAddIM()$ (IM: Intention Mean) dentro de la misma se tiene en cuenta la opción seleccionada y el evento. Por ejemplo la opción: $@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50); .print(end4)., \{N = 2\}$ es el plan seleccionado más la unificación de N con el valor 0 (unificador) y el evento: $+x(0)[source(self), degOfCert(0.9)]$. Como no se encuentra ninguna Intention creada se procede a la creación de la misma. Se observa como queda Circunstance:

Circumstance:

$E = [+x(1)[source(self), degOfCert(0.5)]]$

$I = [@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50); .print(end4)./$
$\{N = 2\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(2)[source(self), degOfCert(0.7)]$

$SO = (@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$
$.print(end4)., \{N = 2\})$

$SI = null$

$AI = null$

$PA = \{1 =< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >\}$

$PI = \{\}$

$FA = [< do(0), @l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./$
$\{N = 0\}, true >].$

Y se pasa al estado $ProcAct$. Aquí se ejecuta la opción $applyProcAct()$, su funcionalidad esta relacionada con las acciones, revisa la feedback actions - ¿FA en Circumstance y setea el estado $SelInt$. En este caso se tiene feedback actions (FA), la cual se evalua antes de pasar al siguiente paso en este caso se agrega nuevamente a la estructura SI quedando:

Circumstance:

$E = [+x(1)[source(self), degOfCert(0.5)]]$

$I = [@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50); .print(end4)./$
$\{N = 2\}, @l\_\_2[source(self)]+x(N) : (N < 3) < -.print(end2)./\{N =$
$0\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(2)[source(self), degOfCert(0.7)]$

$SO = (@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$
$.print(end4)., \{N = 2\})$

$SI = @l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\}$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,].$

Se ejecuta la función *applySelInt*, la misma observa si Circumstance posee Intentions y ejecuta la selectora *selectIntention* para seleccionar una.($@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50); .print(end4)./\{N = 2\}enestecaso$)
Como posee intenciones toma el estado *ExecInt* sino volvería al estado inicial *StartRC*. La función que se ejecutará será *applyExecInt*. Se fijará que tipo de regla posee cada acción. Por ejemplo para el nuevo plan elegido se tienen dos acciones: $do(50); .print(end4)$. En la función se observará de a una acción a la vez, primero se evalua $do(50)$ que es de tipo "action".

Circumstance:

$E = [+x(1)[source(self), degOfCert(0.5)]]$

$I = [@l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\}]$

$A =< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N < 3) <$
$-do(50);$

$.print(end4)./\{N = 2\}, false >$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(2)[source(self), degOfCert(0.7)]$

$SO = (@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$

$.print(end4)., \{N = 2\})$

$SI = @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$

$.print(end4)./\{N = 2\}$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,].$

Ahora la Circumstance en el conjunto A tendrá una nueva tupla <acción, intención>, y en este caso tomará el estado por defecto $ClrInt$. Ahora se ejecutará $applyClrInt$ donde se le pasa la intención. Esta función es para remover la intención de SI. Se remueve si esta finalizado, es decir las acciones fueron ejecutadas. Ahora se sale del bucle y se procederá a ejecutar la acciones pendientes. Se ejecuta $do(50)$ que es lo que había quedado registrado en A y

Circumstance queda de la siguiente manera. (Ver que cambió PA y FA) y se observa que también quedo la primer intención seleccionada.

Circumstance:

$E = [+x(1)[source(self), degOfCert(0.5)]]$

$I = [@l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\}]$

$A =< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$

$.print(end4)./\{N = 2\}, false >$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(2)[source(self), degOfCert(0.7)]$

$SO = (@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$

$.print(end4)., \{N = 2\})$

$SI = @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N < 3) < -do(50);$

$.print(end4)./\{N = 2\}$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,].$

Tercer iteración:

El bucle comenzará a ejecutarse de nuevo ($reasoningCycle$). Comienza reseteando Circumstance en los siguientes conjuntos: A, RP, AP, SI, SO, SE. Con lo que queda la estructura:

Circumstance:

$E = [+x(1)[source(self), degOfCert(0.5)]]$

$I = [@l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = null$

$SO = null$

$SI = null$

$AI = null$

$PA = \{2 =< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N <$
3)

$< -do(50); .print(end4)./\{N = 2\}, true >\}$

$PI = \{\}$

$FA = [< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N <$
$3) < -do(50);$

$.print(end4)./\{N = 2\}, true >].$

Observar que la intención ejecutada en la primera iteración aparece en C debido a que posee acciones pendientes. En esta tercera iteración se resalta solamente lo que más impacta a lo relativo a planes por lo cual se verá del estado $FindOp$ en adelante. Donde se puede observar que quedará el último evento a evaluar.

En esta tercer iteración se comienza con el estado de Circumstance:

Circumstance:

$E = [\,]$

$I = [@l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = [+x(1)[source(self), degOfCert(0.5)]$

$SO = null$

$SI = null$

$AI = null$

$PA = \{2 =< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N <$
3)

$< -do(50); .print(end4)./\{N = 2\}, true >\}$

$PI = \{\}$

$FA = [< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N <$
$3) < -do(50);$

$.print(end4)./\{N = 2\}, true >].$

Se observa de nuevo la función $applyFindOp()$ en base a la función $PlanLibrary.getCandidatePlans$ y dependiendo del evento seleccionado por SE. (en la tercer iteración será $+x(1)$ $[source(self), degOfCert(0.5))$ Esta función trabaja sobre esta lista de planes, la misma que en la primera iteración:

$[@l\_\_0[source(self)]+x(N) : (N >= 3) < -do(50); .print(end1)., @l\_\_1[source(self)]+$
$x(N)[degOfCert(0.7)] : (N < 3) < -do(50); .print(end4)., @l\_\_2[source(self)]+$
$x(N) : (N < 3) < -do(0); .print(end2)., @degOfCert[degOfCert(0.7), source(self)]+$
$x(N) : (N < 3) < -do(20); .print(end3)., @l\_\_3[source(self)] + x(N) : (N <$
$3) < -do(60); .print(end5).]$

La primer corrida dentro de los planes candidatos como al unificar y evaluar N toma el valor 1. Por lo cual el primer plan es descartado y no se transforma en una opción. El plan elegido esta vez es $@l\_\_2[source(self)] +$ $x(N) : (N < 3) < -do(0); .print(end2)$ (Opción elegida). Debido a que no se unifican las anotaciones y es el primero que cumple con la condición siendo $N = 1$.

Circumstance:

$E = [\,]$

$I = [@l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(1)[source(self), degOfCert(0.5)]$

$SO = (@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)., \{N =$
$1\})$

$SI = null$

$AI = null$

$PA = \{2 =< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N < 3)$

$< -do(50); .print(end4)./\{N = 2\}, true >\}$

$PI = \{\}$

$FA = [< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N < 3)$

$< -do(50); .print(end4)./\{N = 2\}, true >].$

Al haberse generado la opción se prosigue automáticamente a $AddIM$. En resumen al encontrar una opción pasa directamente al otro estado. Se ejecutará $applyAddIM()$ (IM: Intention Mean) dentro de la misma se tiene en cuenta la opción seleccionada y el evento. Por ejemplo la opción: $@l\_\_2[source(self)] + x(N) : (N < 3) < -do(0); .print(end2)., \{N = 1\}$ es el plan seleccionado más la unificación de N con el valor 1 (unificador) y el evento: $+x(1)[source(self), degOfCert(0.5)]$. Como no tiene ninguna Intention creada se procede a la creación de la misma. Se observa como queda Circunstance:

Circumstance:

$E = [\,]$

$I = [@l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\},$

$@l\_\_2[source(self)] + x(N) : (N < 3) < -do(0); .print(end2)./\{N = 1\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(1)[source(self), degOfCert(0.5)]$

$SO = (@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)., \{N = 1\})$

$SI = null$

$AI = null$

$PA = \{2 =< do(50), @l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3)$

$< -do(50); .print(end4)./\{N = 2\}, true >\}$

$PI = \{\}$

$FA = [< do(50), @l\_\_1[source(self)] + x(N)[degOfCert(0.7)] : (N < 3)$

$< -do(50); .print(end4)./\{N = 2\}, true >].$

Se pasa al estado $ProcAct$. Aquí se ejecutará la opción $applyProcAct()$ su funcionalidad esta relacionada con las acciones, revisa la feedback actions $- > FA$ en Circumstance y definirá el estado $SelInt$. En este caso se tiene feedback actions (FA), la cual se evalúa antes de pasar al siguiente paso en este caso se agrega nuevamente a I quedando:

Circumstance:

$E = [\,]$

$I = [@l\_\_2[source(self)] + x(N) : (N < 3) < -.print(end2)./\{N = 0\},$

$@l\_\_2[source(self)] + x(N) : (N < 3) < -do(0); .print(end2)./\{N = 1\},$

$@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -.print(end4)./\{N = 2\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = +x(1)[source(self), degOfCert(0.5)]$

$SO = (@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)., \{N = 1\})$

$SI = @l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -.print(end4)./\{N = 2\}$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,].$

Se ejecuta la función $applySelInt$, la misma observa si Circumstance posee Intentions y ejecuta la selectora $selectIntention$ para seleccionar una, siempre la primera $@l\_\_2[source(self)]+x(N) : (N < 3) < -.print(end2)./\{N = 0\}$ en este caso) Como posee intenciones toma el estado $ExecInt$ sino volvería al estado inicial $StartRC$. La función que se ejecutará será $applyExecInt$. Se fijará que tipo de regla posee cada acción. Por ejemplo para el nuevo plan elegido se tiene solo una acción (que quedó pendiente de la primera iteración): $.print(end2)$.

Cuarta iteración: Se observará en esta iteración que se han terminado de ejecutar las dos acciones que disparo el primer TE y que no quedan más eventos para la selectora. Se observará como es el estado de C. Se observan las dos intenciones en I que poseen todavía acciones por ejecutar.

Circumstance:

$E = [\,]$

$I = [@l\_\_2[source(self)]+x(N) : (N < 3) < -do(0); .print(end2)./\{N = 1\},$

$@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -.print(end4)./\{N = 2\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = null$

$SO = null$

$SI = null$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,].$

Al no tener eventos se dirije directamente a $applyProcAct()$ y de ahí directo a $applySelInt()$.

Circumstance:

$E = [\,]$

$I = [@l\_\_1[source(self)]+x(N)[degOfCert(0.7)] : (N < 3) < -.print(end4)./$

$\{N = 2\}]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = null$

$SO = null$

$SI = @l\_\_2[source(self)]{+}x(N) : (N < 3) < -do(0); .print(end2)./\{N = 1\}$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,].$

Al tener intenciones, el selector modifica el estado de SI. Selecciona la primera

$@l\_\_2[source(self)] + x(N) : (N < 3) < -do(0); .print(end2)./\{N = 1\}$. La cual le queda por ejecutar $do(0); .print(end2)$, ejecutará primero $do(0)$. Las próximas iteraciones terminarán de ejecutar las acciones para poder eliminar las intenciones y de esta forma no quedarán acciones pendientes para ejecutar. Hasta llegar al estado de C:

Circumstance:

$E = [\,]$

$I = [\,]$

$A = null$

$MB = [\,]$

$RP = null$

$AP = null$

$SE = null$

$SO = null$

$SI =$

$AI = null$

$PA = \{\}$

$PI = \{\}$

$FA = [\,]$

Bibliografía

Bellifemine, Fabio, Federico Bergenti, Giovanni Caire & Agostino Poggi. 2005. *JADE - a java agent development framework*. New York: Springer chapter 5, pp. 125–147.

Biga, A. & A. Casali. 2013. Una extensión de agentes en JASON para razonar con incertidumbre: G-JASON. In *WASI-CACIC*. Mar del Plata: .

Bordini, R. & J. Hübner. 2007. *BDI Agent Programming in AgentSpeak Using JASON*. John Wiley and Sons.

Bratman, M. E. 1987. "Intention, Plans, and Practical Reason." *Harvard University Press* .

Bratman, M. E., D. J. Israel & M. E. Pollack. 1988. "Plans and resource bounded practical reasoning." *Computational Intelligence* 4:349–355.

Casali, A. 2009. "On intencional and Social Agents with graded Attitudes, Monografíes de L'Institut D'Investigacøen Intel.ligència Artificial.".

Casali, A., Ll. Godo & C. Sierra. 2004. Graded BDI Models For Agent Architectures. In *the Fifth International Workshop on Computational*

Logic in Multi-agent Systems, ed. J. Leite & P. Torroni. Lisboa: CLIMA V pp. 18–33.

Casali, A., Ll. Godo & C. Sierra. 2005. Graded BDI Models For Agent Architectures. In *Lecture Notes in Artificial Intelligence*, ed. Joao Leite & Paolo Torroni. Berling Heidelberg: Springer-Verlag pp. 126–143.

Casali, A., Ll. Godo & C. Sierra. 2008. "A Tourism Recommender Agent: From theory to practice." *Revista Iberoamericana de Inteligencia Artificial* 12:40:23–38.

Casali, A., Ll. Godo & C. Sierra. 2011. "A graded BDI agent model to represent and reason about preferences." *Artificial Intelligence, Special Issue on Preferences Artificial Intelligence* 175:1468–1478.

D'Inverno, M., D. Kinny, M. Luck & M. Wooldridge. 1998. A formal specification of dMARS. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, ed. M.P. Singh, A.S. Rao & M. Wooldridge. Montreal: Springer-Verlag.

Georgeff, M., B. Pell, M. Pollack, M. Tambe & M. Wooldridge. 1999. "The Belief-Desire-Intention Model of Agency." *Intelligent Agents* 1365.

Georgeff, M. & F. Ingrand. 1989. Monitoring and control of spacecraft systems using procedural reasoning. Technical report Australian Artificial Intelligence Institute Australia: .

Georgeff, M. P. & A. L. Lansky. 1987. Reactive reasoning and planning. In *AAAI-87*. Seattle: pp. 677–682.

Ghidini, C. & F. Giunchiglia. 2001. "Local Model Semantics, or Contextual Reasoning = Locality + Compatibility." *Artificial Intelligence* 127(2):221–259.

Godo, L., F. Esteva & P. Hajek. 2000. "Reasoning about probabilities using fuzzy logic." *Neural Network World* 10:811–824.

Howden, N., R. R'onnquista, A. Hodgson & A. Lucas. 2001. JACK Summary of an Agent Infrastructure. In *the 5th International Conference on Autonomous Agents*. Montreal: .

Ingrand, F.F. 2004. "OPRS development environment.".

Jennings, N.R. 1987. "On Agent-Based Software Engineering." *Artificial Intelligence* 117:277–296.

Krapf, A. & A. Casali. 2007. Desarrollo de Sistemas Inteligentes aplicados a redes eléctricas industriales. In *WASI-CACIC*. Corrientes: .

Ljungberg, M. & A. Lucas. 1992. The OASIS air-traffic management system. In *the Second Pacific Rim International Conference on Artificial Intelligence (PRICAI '92)*. Seoul: .

Maes, P. 1991. "The Agent network arquitecture (ANA)." *SIGART Bulletin* 2:115–120.

McGilton, W. H. 1996. "A white paper.The JAVA language environment.".

Parsons, S., C. Sierra & N. R. Jennings. 1998. "Agents that reason and negociate by arguing." *Journal of Logic and Computation* 8(3):261–292.

Rao, A. & M. Georgeff. 1991. Modeling Rational Agents within a BDI-Architecture. In *In proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, ed. R. Fikes & E. Sandewall. San Mateo: pp. 473–484.

Rao, A. & M. Georgeff. 1995. "BDI Agents from Theory to Practice." *AAII* .

Shoam, Y. 1993. "Agent-Oriented Programing." *Journal of Artificial Intelligence* 60(1):51–92.

Weiss, G.(editor). 1999. "Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence." *The MIT Press* .

Wooldridge, M. 1996. Practical reasoning with procedural knowledge: A logic of BDI agents with know-how. In *Proceedings of the International Conference on Formal and Applied Practical Reasoning, FAPR-96*, ed. D. M. Gabbay & H. J. Ohlbach. Berlin: Springer-Verlag pp. 663–678.

Wooldridge, M. 2001. *Introduction to Multiagent Systems.* John Wiley and Sons.

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