UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática

Universitat Politècnica de València

Extensión del lenguaje Jason para la gestión de expectativas

Proyecto Final de Grado

Grado en Ingeniería Informática

Autor: Joaquín Taverner Aparicio

Tutores: Emilio Vivancos Rubio
Vicente Botti Navarro
Bexy Alfonso Espinosa

5 de julio de 2016

Agradecimientos

Agradezco a mis tutores Vicente Botti, Emilio Vivancos y Bexy Alfonso por su apoyo, paciencia y orientación a lo largo del desarrollo de este proyecto, y por hacerme sentir como uno más del grupo desde el primer día.

Abstract

Multi-agent systems are becoming increasingly important in the area of information technology as they provide scalability, adaptability, versatility, autonomy and have a high fault tolerance to faults. But in most of the current multi-agent systems, the process of reasoning is merely rational, forgetting the affective component inherent in any process of human reasoning. In recent years the introduction of emotions on software agents to simulate a more realistic human reasoning is acquiring a great relevance and many proposals have appeared to incorporate affective components into agents. In this final degree work, the proposed framework makes an extension of the *Jason* multi-agent programming language with the aim of allowing the use of expectations that represent an anticipatory mental component in the process of thinking. The expectations in an agent are the foundation on which it will generate emotions (hope, fear, satisfaction, disappointment) depending on which of these expectations are or not satisfied. This process of creation and evaluation of expectations will allow to modify the affective state of the agent, and therefore influence later in its decision-making process. This extension of *Jason*, is a module that will be included in the architecture for affective agents *GenIA*^{3}. In this work we will carry out the parts that correspond to the modification of the lexical, syntactical and semantical analyzer as well as the entire system of transitions between the states of the *Jason* agents reasoning cycle to include the management of expectations. It was also carried out the design and implementation of the verification system to check fulfilment of expectations that can change the affective state of the agent. Finally an emotional agent in *Jason* has been designed that allows to check the correct operation of the realized extension.

Keywords: Jason, agent, emotion, expectation.

Resumen

Los sistemas multi-agente están adquiriendo cada vez mayor importancia en el área de la tecnología de la información ya que proporcionan escalabilidad, adaptabilidad, versatilidad, autonomía y tienen una alta tolerancia a fallos. Pero en la mayoría de los sistemas multi-agente actuales, el proceso de razonamiento de los agentes es meramente racional, olvidando el componente afectivo inherente a todo proceso de razonamiento humano. En los últimos años la introducción de emociones para simular un razonamiento más humano en agentes software está adquiriendo una gran relevancia y han aparecido numerosas propuestas para incorporar componentes afectivos en los agentes. Dentro de este marco, en este trabajo final de grado, se propone realizar una extensión del lenguaje de programación de sistemas multi-agente *Jason* con el objetivo de permitir el uso de expectativas que representen un componente mental anticipatorio en el proceso de razonamiento. Las expectativas en un agente son una de las bases sobre las que se generarán emociones (esperanza, miedo, satisfacción, decepción) en función de que dichas expectativas se vean o no satisfechas. Este proceso de creación y evaluación de expectativas permitirá modificar el estado afectivo del agente, y por lo tanto, influir posteriormente en su proceso de toma de decisiones. Esta extensión de *Jason*, es un módulo que será incluido en el desarrollo de la arquitectura para agentes afectivos *GenIA*^{3}. En este trabajo se realizarán las partes que corresponden a la modificación del analizador léxico-sintáctico y semántico así como todo el sistema de transiciones entre estados de los agentes *Jason* para incluir el manejo de las expectativas. También se realizan el diseño y la implementación del sistema de comprobación del cumplimiento de expectativas que permita modificar el estado afectivo del agente. Finalmente se ha diseñado un agente emocional en *Jason* que permite comprobar el correcto funcionamiento de la extensión realizada.

Palabras clave: Jason, agente, emoción, expectativa.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En los últimos años hemos asistido a un aumento en la cantidad de investigaciones que se realizan sobre lenguajes orientados a sistemas multi-agente[1][10]. La mayoría de estas investigaciones se han centrado tradicionalmente en la búsqueda de soluciones racionales que maximizan la calidad o utilidad de los resultados[6]. Sin embargo, en algunos dominios este tipo de agentes racionales no resultan del todo útiles. Por ejemplo, a la hora de simular personajes virtuales, se hace necesario lograr comportamientos que se asemejen lo máximo posible a los comportamientos humanos. Los comportamientos humanos no son meramente racionales sino que están influenciados en mayor o menor medida, por características afectivas tales como la personalidad, las emociones o el estado de ánimo actual del individuo. Por tanto para que los agentes inteligentes tengan una conducta más similar a la humana se hace necesario la incorporación de características afectivas en el proceso de razonamiento y en el comportamiento. Muchas representaciones y formalizaciones de agentes afectivos se basan en la perspectiva cognitiva de las emociones. Estas formalizaciones modelan el proceso de evaluación, la dinámica de las emociones o el efecto que tienen sobre los procesos cognitivos y de comportamiento de los agentes. Sin embargo, no existen muchas que realicen una formalización general de la interrelación entre los procesos racionales y los procesos afectivos. Una de las formalizaciones que sí tiene en cuenta estos procesos es la arquitectura *GenIA*^{3}[6].

La arquitectura *GenIA*^{3} esta basada en teorías psicológicas y neurológicas ampliamente aceptadas en el ámbito de las emociones. *GenIA*^{3} está construido sobre una arquitectura tradicional BDI, que ofrece componentes para representar rasgos afectivos como la personalidad, las emociones y el

[1]Los sistemas multi-agente son sistemas formados por varios agentes inteligentes que interactúan entre ellos.

estado de ánimo. Al estar basada en una arquitectura BDI, *GenIA*^{3} obedece a las teorías de la motivación y la generación de la acción, donde el curso de las acciones a ejecutar se deciden de acuerdo con los objetivos del agente. La teoría afirma que los agentes BDI tienen un compromiso con sus intenciones, con el propósito de realizar aquellas acciones que ellos creen que pueden conducir a sus deseos.

Partiendo de la arquitectura *GenIA*^{3}, el objetivo de este trabajo se centra en el desarrollo del módulo de gestión de las expectativas en agentes *Jason*. Posteriormente, este trabajo será incorporado al resto de la implementación de la arquitectura *GenIA*^{3} con el fin de crear una plataforma para agentes afectivos. Concretamente en este trabajo se realizarán las partes que corresponden a la modificación del analizador léxico, sintáctico y semántico así como todo el sistema de transiciones entre estados de los agentes *Jason* para incluir el manejo de las expectativas. También se realizan el diseño y la implementación del sistema de comprobación del cumplimiento de expectativas que permita modificar el estado afectivo del agente. Finalmente se ha diseñado un agente emocional en *Jason* que permite comprobar el correcto funcionamiento de la extensión realizada.

El trabajo está organizado en cinco capítulos: el primer capítulo está destinado a la introducción del trabajo y a la revisión de los conceptos previos. El segundo capítulo está dividido en dos secciones principales, la primera trata sobre la incorporación de las emociones en *Jason* y la segunda se centra en las expectativas. En el capítulo tercero se hace un recorrido por las diferentes etapas de la implementación de las expectativas en *Jason*. El capítulo cuarto presenta un ejemplo para comprobar el correcto funcionamiento de la extensión realizada. Por último en el capítulo quinto se exponen las conclusiones obtenidas durante el desarrollo del trabajo así como las futuras ampliaciones.

1.1. Concepto de Agente Inteligente

El concepto de agente software no es sencillo de definir, pues el debate sobre que constituye un agente está todavía en curso. Sin embargo se puede decir de forma generalizada que un agente es un sistema reactivo que desarrolla su actividad dentro de un entorno y presenta cierto grado de autonomía en el sentido en que se le delega una tarea, y el propio sistema determina la mejor manera de lograr esta tarea[41]. Los agentes deben alcanzar su objetivo reaccionando a los cambios que se produzcan en su entorno averiguando por sí mismos cuál es la mejor forma para lograrlos, en lugar de que el programa-

dor tenga que especificar a bajo nivel cómo conseguirlos[42][35]. Para poder interactuar con el entorno estos agentes deben ser capaces de analizar dicho entorno mediante el uso de sensores, y deben tener un repertorio de posibles acciones que pueden realizar con el fin de modificar su entorno usando los actuadores como puede verse en la figura 1.1. Además del entorno, los agentes tienen una serie de propiedades generales:

- Autonomía: El agente debe ser capaz de operar de forma independiente con el fin de alcanzar los objetivos que se le delegan. Por lo tanto, un agente autónomo toma decisiones independientes sobre cómo lograr sus objetivos delegados. Sus decisiones (y por tanto sus acciones) están bajo su propio control.

- Reactividad: El agente debe ser sensible a los cambios que se produzcan en su entorno y reaccionar tomando las decisiones adecuadas en cada momento.

- Proactividad: El agente debe ser capaz de exhibir un comportamiento dirigido a un objetivo. Asume el pleno control de su conducta de modo activo, lo que implica la toma de iniciativa en el desarrollo de acciones que se traduzcan en el cumplimiento de sus objetivos. La proactividad descarta por completo los agentes pasivos, que nunca tratan de hacer nada.

- Capacidad social: Se refiere al intercambio de información crucial para la toma de decisiones en los agentes, esto es, que los agentes deben intercambiar creencias, planes y objetivos para conseguir un fin común.

1.2. Arquitectura BDI

Las teorías de agentes son especificaciones formales de dichos agentes. Existen cuatro clases generales de arquitecturas para agentes[16]: las basadas en la lógica donde las decisiones se toman a través de un proceso de razonamiento puramente lógico, las basadas en agentes reactivos que realizan una acción en función de la situación actual, las basadas en deseos, creencias e intenciones, los cuales son tomados en cuenta para la toma de decisiones y por último las basadas en capas que realizan un razonamiento mediante el uso de diferentes niveles de abstracción. Dentro de este marco una de las más importantes y usadas es la arquitectura *BDI* que son las siglas de *Beliefs,*

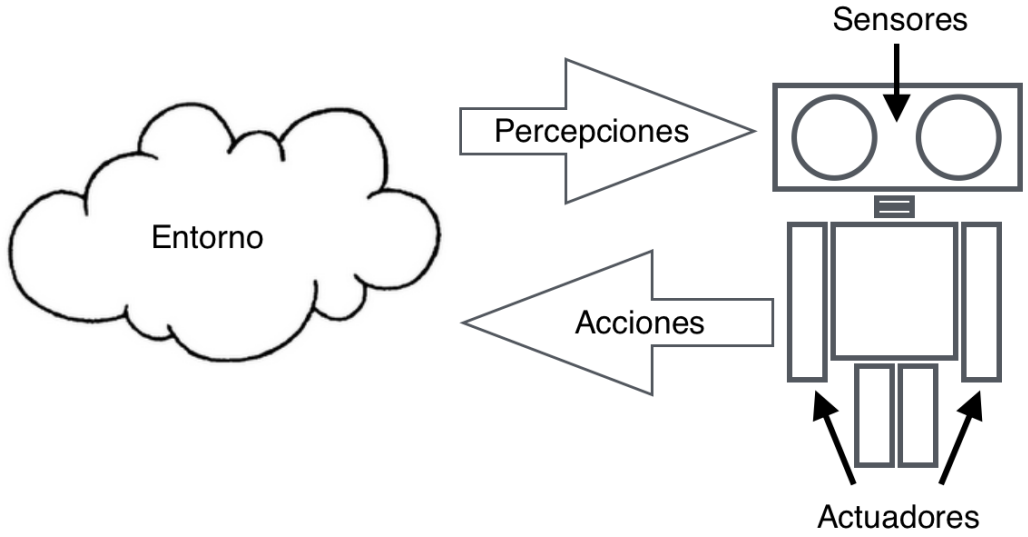

Figura 1.1: Comportamiento de un agente en su entorno.

Desires, Intentions. La arquitectura *BDI* se fundamenta en la teoría filosófica del "razonamiento práctico"[2][13] [12] en el que las acciones se deciden en consonancia con los objetivos. En la arquitectura *BDI* los agentes están diseñados como un sistema intencional donde al agente se le atribuye un conjunto de actitudes mentales. Los agentes están formados por un conjunto de creencias (que en *Jason* se denomina base de creencias), deseos e intenciones. Las creencias representan la información que tiene el agente sobre el estado de su entorno. Esta información no tiene por que ser completa como razonaron *Rao* y *Georgeff* en su definición de la lógica BDI[30, 31]. Los deseos son objetivos que el agente quiere conseguir, por ello tienen preferencia. Las intenciones son compromisos que adquiere el agente, es decir aquellos objetivos que selecciona para ejecutarlos. Los agentes toman la decisión de que acción realizar en función del estado en el que se encuentren.

Los agentes *BDI* llevan a cabo el proceso de razonamiento a través de cuatro procesos más específicos que son: la revisión de las creencias (donde se comprueban las percepciones, los mensajes nuevos y se contrastan con las creencias actuales para obtener nuevas creencias), la generación de opciones (las opciones son deducidas por el agente a partir de la base de creencias y las intenciones actuales), el filtro (determina las nuevas intenciones en relación con la base de las creencias actuales, los deseos y las intenciones), y la

[2]El razonamiento práctico se centra en dos procesos: la deliberación donde el agente decide qué hacer mediante la determinación de los objetivos que se persiguen y el razonamiento sobre los medios para llegar al fin ("*means-end reasoning*")[11].

selección de acciones (que determina la acción a realizar teniendo en cuenta las intenciones actuales)[4].

1.3. Jason: una extensión de *AgentSpeak*

Jason es un intérprete basado en *Java* de una extensión de *AgentSpeak*. *AgentSpeak* es un lenguaje de programación orientado a agentes con una arquitectura *BDI*[10]. Un agente en *Jason* consta de un conjunto de creencias, un conjunto de planes[3] y un conjunto de objetivos.

- Las creencias representan el conocimiento que tiene el agente del estado actual del entorno donde se encuentra. No son estáticas, de hecho suelen estar en constante cambio durante la ejecución del agente. Hay tres fuentes principales que modifican la base de creencias: los mensajes, las percepciones y el propio razonamiento del agente. Los mensajes son emitidos por otros agentes y pueden modificar, añadir o eliminar creencias en el agente que los recibe. Las percepciones vienen del entorno y cuando el agente percibe un cambio en el entorno se produce una modificación en su base de creencias. Por último cuando un agente realiza una acción puede modificar sus creencias. Por ejemplo, si se plantea el agente como un robot físico que se encuentra en una habitación donde hay una lata que está en el suelo y su función es recogerla, existirá al menos una creencia de que la lata está en el suelo por ejemplo `situacion(lata,suelo)`[4]. Cuando el robot ejecuta la acción de coger la lata puede modificar su creencia para indicar que la lata ya no esta en el suelo si no que la tiene el robot. Para poder realizar este cambio se elimina la creencia `situacion(lata,suelo)` y se añade una nueva creencia `situacion(lata,robot)`.

  Cuando se definen las creencias, también se pueden añadir reglas que permitan deducirlas. Estas reglas constan de una parte izquierda con la creencia que se deducirá, y una parte derecha con el conjunto de condiciones que deben cumplirse. La estructura es la siguiente:

[3]Una de las ideas clave en AgentSpeak es que el agente ejecuta estos planes sobre la marcha con el fin de alcanzar planes globales más complejos que le permitan lograr sus objetivos.

[4]Este tipo de estructura en *Jason* se denomina predicado, tanto los paréntesis como los parámetros son opcionales, por ejemplo `situacion` también es un predicado válido. El número de parámetros y su tipo también son opcionales.

```
creencia :- condición .
```

En las condiciones pueden ponerse otras creencias, acciones internas o expresiones mediante una sintaxis parecida a *Prolog*. Por ejemplo si se quiere deducir la posición de la lata en el entorno, se puede tener la regla que la deduzca dada la situación de la lata y la posición de la lata obtenida por la percepción del entorno:

```
lata(X,Y) :- situacion(lata,suelo) &
posLata(X,Y)[source(S)] & S == percept.
```

En caso de que la lata esté en el suelo y la creencia `posLata(X,Y)` que tiene asociada una anotación que, como se explicará más adelante, le indica cual es la fuente de la información, y proviene de las percepciones entonces se cumple y por tanto la creencia `lata(X,Y)` será deducida, y las variables "`X`" e "`Y`" obtendrán sus valores a partir de la creencia `posLata`.

- Los objetivos son los logros que debe alcanzar el agente. Existen dos tipos: los *Achivement Goals* y los *Test Goals*. Los *Achivement Goals* son objetivos a alcanzar por el agente que serán verdaderos una vez alcanzados. Para denotarlos se emplea el símbolo "`!`". Siguiendo el ejemplo anterior, el agente podría tener un objetivo del estilo `!recoger(lata)` que significa que el agente tiene como objetivo recoger una lata y deberá alcanzarlo mediante la aplicación de los planes que tenga disponibles. En cambio los *Test Goals* se expresan mediante el símbolo "`?`" y se usan para realizar consultas a la base de creencias. Siguiendo con el ejemplo, si se quiere saber la situación de la lata en un momento determinado de la ejecución se puede realizar la siguiente pregunta `?situacion(lata,X)`, donde la variable "`X`" se unificará con el valor de la creencia que tenga el agente, por ejemplo si tiene la creencia `situacion(lata,suelo)` "`X`" cogerá el valor "`suelo`".

- Los planes son una secuencia de acciones que debe llevar a cabo el agente para realizar una acción. Los planes tienen tres partes: el evento disparador del plan, el contexto y el cuerpo, estos dos últimos son opcionales. La estructura para los planes es la siguiente:

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

El evento disparador responde a la necesidad de los agentes de ser proactivos y reactivos de forma que, mientras están tomando decisiones para cumplir sus objetivos, puedan atender a cambios en su entorno. Cuando se cumple un evento disparador el plan que lleva asociado pasa a ser relevante y podrá ser escogido para su ejecución durante la fase de razonamiento. Hay dos tipos principales de eventos disparadores, los de adición y los de eliminación. Los de adición disparan su plan cuando se añade una creencia o un objetivo. Los de eliminación disparan su plan cuando se elimina una creencia o un objetivo. En la tabla 1.1 pueden verse los distintos eventos disparadores.

Notación	**Definición**
`+creencia`	Cuando se añada la creencia
`-creencia`	Cuando se elimine la creencia
`+!objetivo`	Cuando se añada el *achivement-goal*
`-!objetivo`	Cuando se elimine el *achivement-goal*
`+?test`	Cuando se añada el *test-goal*
`-?test`	Cuando se elimine el *test-goal*

Cuadro 1.1: Tipos de eventos disparadores.

El contexto esta formado por un conjunto de condiciones que se deben cumplir para que el plan pase a ser aplicable. Estas condiciones deben de poder ser deducidas de la base de creencias, y permiten realizar una distinción entre planes, de forma que puedan existir varios planes para un mismo evento disparador, pero cada plan puede tener un contexto distinto. Como el contexto es opcional, si no se define será siempre cierto. También se permite el uso de expresiones relacionales utilizando una sintaxis bastante parecida a *Prolog*. Igual que ocurre con los eventos disparadores, también existen distintas condiciones que pueden verse en la tabla 1.2. Estas condiciones pueden ser agrupadas mediante el operador "`&`".

El cuerpo del plan contiene aquellas acciones que han de ser ejecutadas cuando un evento coincide con el evento disparador y el contexto del plan es cierto. Cada acción irá separada por un punto y coma. Hay seis tipos diferentes de acciones que se pueden introducir en un plan:

- **Acciones:** Las acciones son la forma que tienen los agentes de interactuar con el entorno. Siguiendo con el ejemplo del robot, es

Notación	Definición
`creencia`	El agente cree que la creencia es cierta
`~creencia`	El agente cree que la creencia es falsa
`not creencia`	El agente no cree que la creencia es cierta
`not ~creencia`	El agente no cree que la creencia es falsa

Cuadro 1.2: Tipos de condiciones para el contexto.

necesario que existan acciones que permitan realizar acciones sobre el hardware del robot, por ejemplo si el robot cuenta con una pinza para coger las latas, habrá una función del estilo `pinza(abrir)` para abrir la pinza. Todas las variables que se empleen en una acción deben de estar previamente instanciadas. Como puede verse, la estructura de las acciones es la misma que la de las creencias: un predicado. La distinción entre ambas se produce por su localización. Las acciones solo pueden estar en los cuerpos de los planes, y en los cuerpos de los planes las creencias van acompañadas de un símbolo de inserción, borrado o modificación.

Las acciones deben ser conocidas por el programador, ya que para cada aplicación se deberán desarrollar las acciones correspondientes que serán implementadas en un fichero de entorno. Cuando se ejecuta una acción el agente queda suspendido. Para saber si esa acción ha tenido éxito, el entorno devuelve una variable de tipo lógico que será cierta en caso de que la acción haya podido ejecutarse adecuadamente.

- **Acciones internas:** Por lo que respecta a la sintaxis, se diferencian de las acciones en que se emplea un punto delante, por ejemplo `.lenght(X)` que devuelve la longitud de "`X`" si es una lista. En cuanto a su función, se diferencian de las acciones en que, mientras estas últimas se ejecutan en el entorno, que debe verse como algo externo al agente, las acciones internas realizan su ejecución dentro del código del agente, por tanto pueden acceder a los métodos de las clases del propio agente. Esto permite al programador poder extender el lenguaje y emplear métodos para controlar diferentes parámetros dentro del propio agente. Existe una serie de funciones internas predeterminadas en *Jason* como por ejemplo `.max` que devuelve el valor máximo de una lista de valores o `.println` que permite imprimir mensajes por consola.

- ***Achivement Goals:*** Cuando aparece un *Achivement Goals* dentro del cuerpo de un plan se añade como objetivo. Si existe un plan que tenga como evento disparador este objetivo, se añade a la lista de planes relevantes y podrá ser seleccionado para su ejecución. Estos objetivos pueden tener éxito si se ejecuta correctamente su plan asociado, pero también pueden producir fallos debido a que se haya producido un fallo en el plan asociado o a que no exista ningún plan para ese objetivo. Cuando un objetivo no tiene éxito produce un fallo del plan al que pertenece. Cuando aparece un objetivo en un plan si tiene delante el símbolo "`!`" el plan queda suspendido hasta que el objetivo termine, tanto si tiene éxito como si no. Pero no siempre interesa que los planes queden suspendidos ya que puede ser que el nuevo objetivo no sea prioritario para ese plan. Para solucionarlo se coloca una segunda exclamación "`!!`", de esta forma el plan añade el objetivo pero continua desarrollándose.
- ***Test Goals:*** Permiten conocer el estado de una creencia en cualquier punto del cuerpo del plan. Aunque en el contexto se realiza una comprobación del estado de las creencias, es posible que durante la ejecución del plan se modifique la base de creencias. Igual que en el caso anterior pueden ser de éxito o de fallo. En este caso el fallo se produce si no se puede deducir el test de la base de creencias, pero a diferencia de los *Achivement Goals* no necesita que haya un plan asociado.
- **Notas mentales:** Las notas mentales permiten modificar la base de creencias. Se pueden añadir, modificar o eliminar creencias. Para la modificación no es necesario que la creencia exista previamente en la base de creencias, si no está, simplemente la añade. La estructura puede verse en la tabla 1.3.

Notación	**Definición**
`+creencia`	Se añade la creencia a la base de creencias del agente
`-creencia`	Se elimina la creencia de la base de creencias del agente
`-+creencia`	Se modifica la creencia de la base de creencias del agente

Cuadro 1.3: Tipos de notas mentales.

- **Expresiones:** Igual que ocurría en el contexto, dentro de un plan también pueden aparecer expresiones relacionales que adquieran valores de cierto o falso, siguiendo la misma sintaxis que en el contexto. Además se pueden asignar valores a variables con el operador "=". Por ejemplo `N = .lenght(A)`, que unificará "`N`" con el valor del tamaño de la lista "`A`".

Otro elemento importante de los planes son las etiquetas. Las etiquetas se pueden interpretar como un nombre que se le asigna a un plan. No es necesario explicitarlas ya que *Jason* le asigna a cada plan un nombre por defecto. Cuando a un plan se le asigna un nombre, se permite al programador poder seleccionar ese plan en cualquier punto de la ejecución. Una de las cosas más importantes de las etiquetas es que pueden llevar asociadas distintas anotaciones. Esto permite que puedan existir distintos planes con un mismo evento disparador, incluso con el mismo contexto pero con distintas etiquetas, que permiten realizar una selección de planes más clara. Incluso se pueden usar en el propio código interno del agente para realizar distinciones entre planes. Cuando se añaden las etiquetas los planes quedan con la siguiente estructura:

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

Por ejemplo una etiqueta podría ser `@plan[source(self)]`. Esta etiqueta crea una anotación para el plan `source(self)` que podrá ser empleada en el método *selecOption* para determinar si el plan es aplicable. A continuación, siguiendo con el ejemplo del robot, se muestra el plan asociado al evento disparador `!recoger(lata)`:

```
1 +!recoger(lata) : posRobot(X,Y) & posLata(X,Y)
2 <-
3     !agacharse;
4     !pinza(cerrar);
5     -posLata(X,Y);
6     -+situacion(lata,robot).
```

Este plan tiene como contexto la condición de que debe poderse deducir de la base de creencias que el robot se encuentra en una posición determinada, y que la lata se encuentra en la misma posición, ya que ambas condiciones comparten las mismas variables. En caso afirmativo este plan será aplicable. Cuando se ejecute realizará la acción de agacharse, y espera a que concluya, si tiene éxito ejecutará la acción de cerrar la pinza. Por último elimina la creencia de que la lata se encuentra en esa posición, y modifica la creencia `situacion` para que la lata deje de estar en el suelo y pase a estar en el robot.

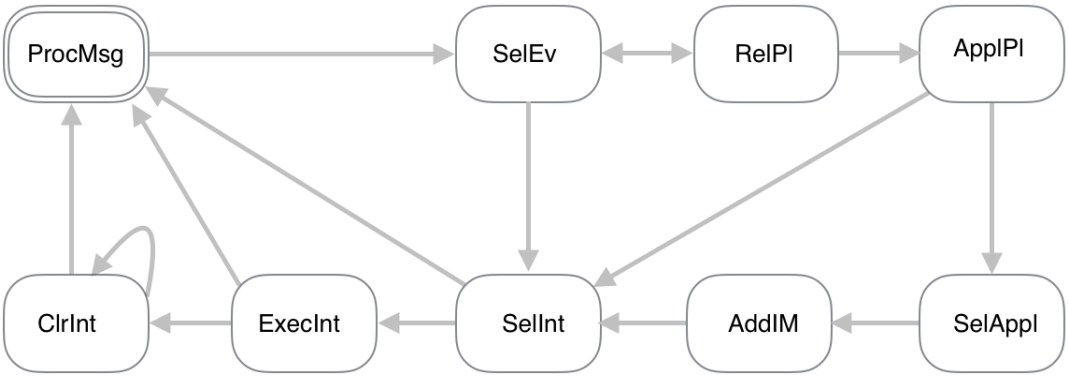

Figura 1.2: Ciclo de razonamiento *Jason* original.

En *Jason* los agentes tienen un ciclo de razonamiento propio. En cada ciclo de razonamiento el agente comprueba los mensajes que recibe e interactúa con el entorno (percepción). Estas acciones pueden añadir creencias u objetivos que generarán eventos que pueden llevar asociados planes. Del conjunto de planes aplicables se selecciona uno que será el que se ejecutará.

El ciclo de razonamiento (figura 1.2) es un autómata finito que consta de los siguientes estados: *ProcMsg* estado inicial en el que se procesan los mensajes recibidos, *SelEv* donde se selecciona el evento, *RelPl* aquí se realiza la selección de los planes que son relevantes, *ApplPl* en este paso se hace una comprobación de cuales de los planes relevantes son aplicables, *SelAppl* se selecciona un plan a aplicar, en el estado *AddIM* se añade la nueva intención al conjunto de intenciones, *SelInt* es el estado donde se selecciona una intención, *ExecInt* en este paso se ejecuta la intención, *ClrInt* aquí es donde se eliminan las intenciones que se hayan ejecutado en el paso anterior.

1.4. El concepto de la emoción

Las emociones tienen muchas facetas que involucran sentimientos, experiencias, fisiolofías, comportamientos, conocimientos y conceptualizaciones [26]. Cuando se habla de cualquier cosa relacionada con la informática siempre se piensa en procesos meramente racionales que siguen una cierta lógica. Pero el ser humano normalmente no actúa de manera racional. Las decisiones que toma un ser humano están influenciadas por sus emociones y sus características afectivas[6]. La filosofía contemporánea sostiene que las emociones

juegan un papel importante en el razonamiento práctico[5] [28]. De forma que las emociones quedan ligadas a los procesos racionales contrariamente a la visión tradicional que trataba las emociones como fenómenos "pasivos"[4]. Las emociones, por tanto, tienen un peso importante en los procesos de razonamiento que permiten la toma de decisiones. Bajo este punto de vista, las emociones están muy ligadas a la racionalidad y también es una tendencia de las áreas de investigación, como la psicología o la neurociencia.

Las emociones también están relacionadas con la memoria[4]. Existe evidencia de que las personas recuerdan mejor los fenómenos que han sido emocionalmente significativos[17]. La memoria también afecta a las emociones, de forma que un evento que en el pasado generó una emoción, condiciona la emoción producida cuando se repite de nuevo el evento. En la relación de la emoción y la memoria dos procesos importantes se pueden destacar. En primer lugar el proceso a cargo de determinar qué y cómo los recuerdos emocionales se almacenan, y en segundo lugar, el proceso que determina cómo estos recuerdos evolucionan con el tiempo ya que las emociones inicialmente tienen un impacto fuerte en los individuos que con el tiempo va disminuyendo. La generación de comportamiento reactivo generalmente se realiza principalmente empleando a las experiencias pasadas y la memoria emocional. Lo mismo ocurre con la toma de decisiones, que también se lleva a cabo mediante la evaluación de las expectativas.

Las ciencias psicológicas y cognitivas han sentado los cimientos para futuras investigaciones en computación emocional. Teorías como las conductistas evolutivas han inspirado a los enfoques computacionales actuales en un mayor o menor grado. En los últimos años han ido apareciendo diferentes estudios para incorporar las emociones en los procesos software [29, 6, 18, 14]. Cuando se quiere que los agentes software tengan un razonamiento similar al humano se hace necesario incluir las emociones en su proceso de razonamiento. Pero representar las emociones dentro de un proceso software resulta una tarea bastante compleja y normalmente los desarrolladores se ven en la obligación de crear sus propios modelos de emoción desde cero. Por eso se hace necesario la creación de normas y directrices sistemáticas. Sin embargo hay muy pocas propuestas en este campo que tengan en cuenta la posterior implementación en agentes software. La mayoría no ofrecen una arquitectura de agentes afectivos. Por eso, nace la necesidad de crear nuevos modelos de agen-

[5]La mayoría de las teorías de simulación de las emociones se basan en el razonamiento práctico. Aristóteles definió el razonamiento práctico como el razonamiento que concluye en una acción. El razonamiento es un proceso mental, que tiene lugar en la mente del razonador y su conclusión debe ser un estado mental[13].

te que permitan la programación de agentes emocionales. Como por ejemplo, *GenIA*^{3}[6], que propone una arquitectura pensada para la incorporación de los agentes afectivos en *Jason*.

El proceso de desarrollo de un modelo computacional puede ayudar a probar y concretar distintas teorías psicológicas, obligando a cumplir compromisos sobre la forma abstracta en la que se realizan las construcciones teóricas. Los modelos computacionales proporcionan un laboratorio sobre el cual se pueden experimentar, a través de la simulación, distintas teorías que permitan a los investigadores derivar predicciones que pueden probarse posteriormente con los datos humanos. Los experimentos realizados mediante simulación a menudo resultan más eficientes que los realizados con humanos, ya que las máquinas no están condicionadas a priori a la experimentación. Por otra parte, se eliminan las preocupaciones éticas que son centrales para cualquier investigación que involucra la evocación de emociones en sujetos humanos.

Los modelos emocionales para agentes inteligentes deben dotarlos de la capacidad de razonar mediante el uso de las emociones, igual que ocurre en los seres humanos. El modelo emocional debe ser capaz de evaluar todas las situaciones que el agente pueda encontrar y también debe proporcionar una estructura para las variables que influyen en la intensidad de una emoción. Uno de los modelos con mayor relevancia en las investigaciones relacionadas con la aplicación de las emociones en los procesos de razonamiento software es el modelo *OCC*[2, 8]. *OCC* es un modelo estándar para la síntesis de emociones. Este modelo especifica veintidós categorías de emociones basadas en reacciones a distintas situaciones. Los aspectos cuantitativos de las emociones se describen mediante tres términos: la potencia, el umbral y la intensidad. Por ejemplo, para la intensidad existen veinticinco variables distintas. Para cada emoción existen una serie de variables de intensidad que le afectan. Las variables de intensidad están divididas en locales y globales. Las locales solo afectan a ciertas emociones, mientras que las globales afectan a todas las emociones. Sólo cuatro variables están identificadas como variables globales: el sentido de la realidad, la proximidad psicológica, lo inesperado, y la activación fisiológica[37]. Un ejemplo del uso práctico de *OCC* es el proyecto *ALMA*[18]. En *ALMA* se desarrollan personajes virtuales interactivos que simulen el comportamiento humano. Estos personajes interactivos tienen un perfil de personalidad y muestran las emociones y los estados de ánimo en tiempo real.

Del conjunto de las variables globales, para este trabajo la que tiene el

interés principal es la variable que mide el valor de lo inesperado. Esta variable se emplea para medir cuan inesperada es una situación para el agente, es decir cuando se presenta esta situación en que medida se produce sorpresa en el agente. La sorpresa puede ser positiva o negativa, es decir, existen tanto sorpresas agradables como desagradables. Además, debe tenerse en cuenta que inesperado se considera que es una expectativa que mira hacia atrás, es decir, cuando se evaluá una situación, para poder conocer el valor de cuan inesperada es esa situación, hay que comprobar el grado en el que los eventos generados con anterioridad permitían la deducción de lo que ha sucedido. Esto significa que lo inesperado de un evento, sólo se evalúa después de haber sido percibido. De este modo lo inesperado de un evento no depende de consideraciones previas a la percepción del evento. Debido a que la sorpresa hace que la emoción sea más intensa, este aspecto se ha incorporado en el modelo *OCC* como una variable que influye en la intensidad de las emociones. La variable de lo inesperado da una medida de valor de lo esperado (o sea, de las expectativas), ya que se complementan. De esta forma se puede evaluar el valor de lo inesperado comprobando el cumplimiento de lo que sí es esperado. Por ejemplo, si el agente está esperando que dentro de una hora le llamen por teléfono, y cuando pasa la hora no le han llamado no se cumple la expectativa, por tanto el valor de lo inesperado será el complementario de esa expectativa, es decir, que no se haya producido la llamada incrementa el valor de lo inesperado.

La personalidad juega un factor muy importante en las emociones. El término personalidad hace referencia a la manera característica en la que una persona piensa, siente, se comporta y se relaciona con otros. Es el único rasgo afectivo que se mantiene a largo plazo y refleja las diferencias individuales en las características mentales. La personalidad puede definirse en un espacio dimensional definiendo los distintos rasgos que la componen. Una de las teorías más avaladas en este ámbito es la de los *Cinco Grandes*[33]. La teoría de los *Cinco Grandes* divide la personalidad en cinco dimensiones. Cada dimensión se corresponde con un aspecto del individuo y se mantiene estable a largo de su ciclo de vida. Estos aspectos determinan la manera en que el individuo debe responder a los distintos estímulos durante su vida. Los aspectos que forman las cinco dimensiones son:

- Afabilidad: es la amabilidad hacia los otros, e incluye rasgos como altruismo, ternura, confianza y modestia.

- Meticulosidad: esta capacidad permite controlar los impulsos logrando que el comportamiento esté dirigido a alcanzar los objetivos.

- Extroversión: capacidad de relacionarse con los demás y mostrar abiertamente los sentimientos.

- Neuroticismo: o inestabilidad emocional. Esta relacionado con la inestabilidad e inseguridad emocional, la ansiedad y el estado continuo de preocupación y tensión.

- Apertura a la experiencia: describe la amplitud, profundidad, originalidad y complejidad de la vida mental y aquella derivada de la experiencia de una persona.

En la representación de las emociones es fundamental mostrar los distintos estados de ánimo. El estado de ánimo es la disposición que muestra un individuo en un momento determinado[38]. Los estados de ánimo son diferentes de las emociones o los sentimientos en cuanto a que son menos específicos, menos intensos y tienen menos probabilidades de ser desencadenados por un estímulo o evento en particular. Además los estados de ánimo son menos volátiles que las emociones, es decir varían menos con el tiempo y tienen una duración mayor[27]. Los estados de ánimo típicamente tienen un valor positivo o negativo, es lo que comúnmente se conoce como buen o mal humor. El estado de ánimo suele sufrir oscilaciones en el tiempo como respuesta a eventos positivos o negativos. El estado de ánimo es un estado interno que depende de cada individuo. Cuando un estado de ánimo se prolonga en el tiempo se denomina estado de equilibrio, humor dominante o estado fundamental de ánimo. Este estado es al que tiende el sujeto cuando no tiene estímulos externos que puedan alterarlo. Este estado de equilibrio depende de los rasgos de la personalidad, es decir la personalidad predispone a ciertos estados de ánimo. Por ejemplo, un sujeto con una personalidad nerviosa puede tener como estado de equilibrio un estado con un nivel alto de exaltación, activación y nerviosismo.

Los estados de ánimo, como se ha mencionado anteriormente, son más duraderos que las emociones y no tienen por que estar relacionados directamente con un único estímulo. Las condiciones en las que se modifica el estado de ánimo son: la aparición de un evento ligeramente positivo o negativo, el desplazamiento de un evento inductor de una emoción, el recuerdo de una experiencia emocional y la inhibición de la respuesta emocional en presencia de un evento emocional inductor[25]. Existen diferentes interpretaciones a la hora de modelar los estados de ánimo. Uno de los modelos más relevantes es el de Mehrabian y Russell(1974)[1] cuya estructura *PAD (Pleasure, Arousal, Dominance)*, procedente de la psicología ambiental, describe las respuestas

emocionales del individuo ante el entorno, a través de tres dimensiones numéricas: placer, excitación y dominación[9].

Placer	**excitación**	**dominación**
feliz-infeliz	animado-decaído	controlador-controlado
contento-enfadado	exaltado-calmado	influyente-influenciado
encantado-descontento	entusiasmado-sereno	contenido-afectado
alegre-triste	nervioso-tranquilo	importante-temeroso
ilusionado-desilusionado	activo-pasivo	dominante-dócil
entretenido-aburrido	sorprendido-indiferente	autónomo-guiado

Cuadro 1.4: Las dimensiones PAD de Mehrabian–Russell.

Las dimensiones de *PAD* pueden tener cualquier valor numérico, generalmente se emplean cuatro valores para cada dimensión, con lo que se pueden representar sesenta y cuatro valores posibles para las emociones. El significado de las tres dimensiones es el siguiente: el placer denota cuan agradable o desagradable resulta un estímulo, la excitación es la activación que le produce el estímulo y la dominación es el nivel de dominio o sumisión que representa el estado de ánimo. Por ejemplo, el miedo es un estado de sumisión mientras que la ira es un estado de dominación. Un ejemplo del uso del modelo *PAD* para el cálculo de los distintos estados de ánimo en un personaje virtual es el proyecto *ALMA*[18]. *ALMA* realiza un cálculo del estado de ánimo mediante el uso de unas fórmulas que implican el uso de la personalidad representada mediante los cinco grandes[33]:

$$Placer = 0{,}21 * E + 0,59 * A + 0,19 * N$$

$$Excitacion = 0,15 * AE + 0,30 * A - 0{,}57 * N$$

$$Dominacion = 0,25 * AE + 0,17 * M + 0{,}60 * E - 0{,}32 * A$$

donde E representa la variable Extroversión, A representa la Afabilidad, N el Neuroticismo, M la Meticulosidad y AE es la Apertura a la Experiencia.

De esta forma en *ALMA* se calcula el estado de equilibrio en base a la personalidad del personaje virtual. Para realizar las oscilaciones en los estados de ánimo se emplean las emociones, de forma que cuando se produce un nuevo estímulo que genera una emoción se evalúa el nuevo estado de ánimo. Para poder realizar este proceso de evaluación *ALMA* ha creado una relación directa entre las emociones y el espacio *PAD*. Para cada emoción se

le han asignado tres valores correspondientes a las tres variables del espacio *PAD*. Cuando llega una nueva emoción el estado de ánimo actual se genera en base al estado de ánimo anterior modificándolo con los valores asignados a esa emoción. De esta forma se impiden los cambios bruscos entre los distintos estados de ánimo y también esto permite distinguir la varianza del estado de ánimo en función del tipo de emoción.

Capítulo 2

Emociones y expectativas en un agente *Jason*

2.1. Introducción

En este capítulo se presenta la arquitectura *GenIA*^{3}. El presente trabajo se basa en esta arquitectura, en la cual implementa el módulo de las expectativas. Posteriormente se introducen los conceptos previos sobre las expectativas en agentes inteligentes. Finalmente se presenta la motivación del trabajo partiendo de las descripciones realizadas.

2.2. Emociones en los agentes *Jason*

Actualmente existen diversos lenguajes que permiten modelar agentes inteligentes. Sin embargo no hay demasiados que proporcionen estructuras útiles para programar agentes afectivos empleando además razonamiento práctico. Existen varias investigaciones en el marco teórico que desarrollan modelos computacionales genéricos que permiten la definición de agentes afectivos con un cierto criterio filosófico y psicológico[34]. Pero la mayoría de estas investigaciones no proponen la forma de llevarlas a la práctica mediante el uso de algún lenguaje de agentes. Por tanto aún quedan muchos desafíos para estos enfoques que sean aplicables y prácticos. Para poder avanzar es necesario la creación de una formalización para agentes afectivos que permita modelarlos en un sistema real para poder probar los conceptos teóricos.

Los modelos de emociones en agentes inteligentes están diseñados para lograr imitar lo máximo posible el comportamiento humano. Existen muy pocos modelos y normalmente son de alto nivel, por lo que la implementa-

ción es bastante complicada. Los desarrolladores de lenguajes de agentes a menudo se enfrentan al reto de empezar desde cero cuando es necesario incluir las emociones en la representación de un agente y su comportamiento. La propuesta de la arquitectura *GenIA³* va acompañada de una estructura de lenguaje de agente que incluye los principales aspectos a considerar en relación a las emociones. Por esta razón se ha seleccionado *GenIA³* para la realización de este trabajo. *GenIA³* es una arquitectura de propósito general para agentes *BDI* afectivos. *GenIA³* se basa en teorías psicológicas y neurológicas ampliamente estudiadas y ofrece una visión integral del agente y su comportamiento teniendo en cuenta tanto los atributos y procesos racionales y afectivos. Esta arquitectura incluye una extensión del ciclo de razonamiento de *AgentSpeak*, la definición de su semántica operacional, así como la extensión de la sintaxis del lenguaje de agente *Jason*. Esta arquitectura permitirá realizar las comparaciones entre diferentes teorías psicológicas fácilmente. Por lo tanto nuevas propuestas se podrán adaptar a distintos dominios de aplicación y podrán emplear distintas teorías psicológicas.

Entre las características de *GenIA³* destacan: una mayor flexibilidad, una notación compacta y elegante que evita ambigüedades, permite utilizar diferentes teorías psicológicas, permite personalizar varios pasos del ciclo de razonamiento y del ciclo afectivo, permite definir un nivel de racionalidad (de forma que se puede indicar mediante un valor numérico el grado de racionalidad que se desea que tenga el agente), permite incluir características adicionales en la definición de la personalidad de los agentes y por último esta arquitectura es capaz, no solo de impulsar los objetivos a partir del estado afectivo, sino que además permite generar nuevos objetivos a través de las estrategias de afrontamiento[1] del agente.

2.2.1. Formalización de la arquitectura *GenIA³*

GenIA³ es una arquitectura de propósito general para agentes *BDI*. Para poder llevar a cabo la implementación de agentes afectivos en *Jason* empleando la arquitectura *GenIA³* se propone la formalización descrita a continuación.

[1]Las estrategias de afrontamiento son las respuestas, tanto físicas como fisiológicas, producidas por el individuo para hacer frente a una situación como por ejemplo la ansiedad o el estrés. Las estrategias de afrontamiento definen la forma en que un individuo reacciona a un evento que involucra cambios emocionales, y esas reacciones pueden ser o manifestaciones involuntarias o acciones más planificadas. Las reacciones pueden ser orientadas a cambiar las creencias, metas o intenciones de un individuo.

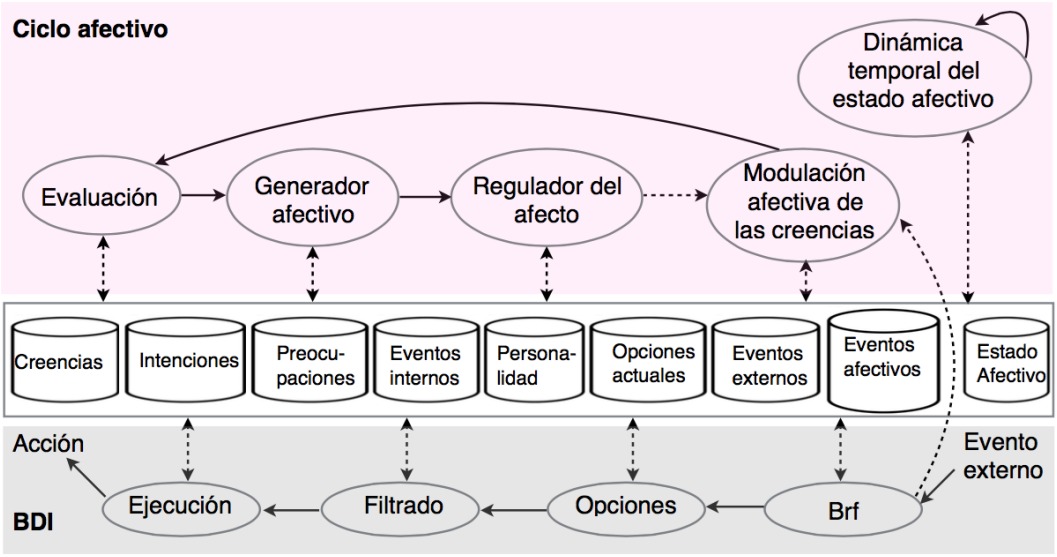

Figura 2.1: Estructura de la arquitectura *GenIA*^{3}.

Esta formalización propone modificar la semántica operacional de *AgentSpeak* y la sintaxis de *Jason* para permitir el desarrollo de agentes *BDI* dotados de características y comportamientos afectivos. Las teorías psicológicas y neurológicas se han centrado tradicionalmente en la descripción de varias características y procesos relacionados con la emoción y con la personalidad. Los procesos relacionados con la emoción, generalmente se estudian desde una perspectiva cognitiva y se pueden agrupar en la generación de la emoción, la experiencia de la emoción y los efectos que produce la emoción. La arquitectura *GenIA*^{3} incluye los procesos centrales en estos tres grupos, así como los procesos de una arquitectura tradicional de agente BDI teniendo en consideración los rasgos de personalidad que pueden influir en los procesos de razonamiento del agente.

La arquitectura *GenIA*^{3} esta basada en la modalidad de emoción cognitiva. Existe un amplio consenso en informática, con una sólida formación teórica, de los principales procesos afectivos que se deben considerar en la construcción de agentes afectivos[5, 3]:

- Evaluación: Es el proceso mediante el cual se derivan un conjunto de variables de evaluación como resultado de una transformación de la situación actual del agente, sus preocupaciones, y su estado cognitivo.

- Generador afectivo: En este proceso las variables de evaluación, obtenidas en el proceso anterior, se transforman en una representación del estado afectivo del agente.
- Regulador del afecto: Este proceso atiende a la necesidad de dar una respuesta emocional, determinando cuales serán los comportamientos emocionales y las respuestas de afrontamiento. Por ejemplo, los gestos o expresiones faciales.
- Modulación afectiva de las creencias: Determina cómo el estado afectivo debe modificar las creencias del agente. Por ejemplo, si el agente se encuentra en un estado afectivo negativo, tendrá la tendencia a confiar menos en sus creencias.
- Dinámica temporal del estado afectivo: Este proceso es independiente del resto, y determina cómo varía la intensidad del estado afectivo en el tiempo. Cuando un ser humano recibe un estimulo negativo, por ejemplo suspender una asignatura, inicialmente el impacto sobre su estado afectivo es mayor. Conforme el tiempo va pasando este impacto disminuye y en ocasiones desaparece. Mediante este procedimiento se pretende simular esta cualidad humana.

Estos cinco procesos se ejecutan en paralelo junto con los procesos propios de un agente *BDI*. Como puede verse en la figura 2.1, la secuencia de los procesos afectivos es independiente de la secuencia de los procesos de BDI permitiendo que se puedan ejecutar en hilos independientes, pudiéndose escoger el nivel de racionalidad o emoción de cada agente modificando la frecuencia de los ciclos. Por ejemplo, se podrían tener diez ciclos emocionales por cada ciclo racional, de esta forma las emociones tomarían mayor peso en la toma de decisiones obteniéndose un agente más emotivo que racional. De la misma forma el proceso "*Dinámica temporal del estado afectivo*" también puede ejecutarse de forma independiente a los otros.

El ciclo afectivo propuesto en la arquitectura *GenIA*^{3} tiene la siguiente secuencia de ejecución: cuando se genera un evento (interno o externo) se evalúa en el proceso de evaluación. Este proceso tiene en cuenta la información cognitiva del agente como por ejemplo, la personalidad, las preocupaciones o las creencias. El resultado de este proceso se emplea por el generador afectivo para modificar el estado afectivo del agente. En función de ese estado afectivo el regulador del afecto se encarga de seleccionar las estrategias de afrontamiento necesarias, así como de tomar las acciones pertinentes para modificar

el estado afectivo a un estado deseado pudiendo modificar para ello las creencias del agente. Esta modificación de las creencias es llevada a cabo por el proceso de modulación afectiva de las creencias, que también puede ser activado por la función de revisión de creencias (*brf*). De esta forma se puede determinar la forma en la que el estado actual influye en el contenido y en el peso de las creencias. La función de modulación afectiva de las creencias tiene acceso a la base de creencias y puede modificarla para añadir o eliminar creencias. Esta función también puede reactivar el proceso de evaluación.

Además de los procesos afectivos, un lenguaje *BDI* necesitará añadir nuevas estructuras para representar los siguientes atributos relativos al razonamiento afectivo: las preocupaciones, las probabilidades de las creencias y la personalidad[3]. Las preocupaciones juegan un papel importante en la forma en la que el agente evalúa los eventos. Por ello las preocupaciones son una de las ideas más recurrentes en las teorías de evaluación (estas incluyen, las motivaciones, las normas, los ideales) y juegan un papel especial en la elección del sentido de las expectativas, es decir, el que una expectativa sea positiva o negativa depende de las preocupaciones actuales del agente. Por ejemplo, si un sujeto tiene la expectativa de que nieve, si el sujeto tiene como preocupación quedarse atrapado por la nieve, la expectativa será negativa, sin embargo si lo que le interesa es esquiar, la expectativa de que nieve tendrá una connotación positiva. La personalidad, según la mayoría de las teorías relacionadas con el afecto, es un conjunto de características individuales que dirigen el comportamiento y diferencian un sujeto de otro. Además, en *GenIA*^{3} la personalidad puede incluir conceptos como estrategias de afrontamiento o nivel de racionalidad.

Para poder modelar el estado afectivo *GenIA*^{3} introduce también nuevos parámetros en el agente además de los de la arquitectura *BDI*. Estos parámetros son: las preocupaciones, la personalidad, el estado afectivo y eventos relevantes para el cálculo del estado afectivo. La gramática *EBNF* propuesta para *GenIA*^{3} soluciona la inclusión de algunos de estos parámetros como puede verse en la figura 2.2. Esta gramática es una extensión de la gramática *EBNF* de *Jason*. Las palabras subrayadas se corresponden con los símbolos no terminales de la gramática *EBNF* de *Jason*. Las palabras y carácteres entrecomillados se corresponden son los símbolos terminales. Dentro de estos símbolos no terminales cabe destacar las nuevas palabras reservadas: *"concerns___:", "personality___:", "copingst___:"* y *"cs___"*. Estos símbolos se usan para que el programador pueda definir claramente las preocupaciones, la personalidad y los comportamientos emocionales respectivamente. Además ayudan al compilador a diferenciar fácilmente las nuevas estructuras.

Se ha modificado la regla gramatical del no terminal *agent* para añadirle las preocupaciones (*concerns*) y la personalidad (*personality*). Se han añadido nuevos no terminales a la gramática que tienen el siguiente significado:

- concerns: Es una regla[2] cuyo lado izquierdo es un literal formado por la palabra reservada "*concerns___*" y una variable sin instanciar que se espera sea de tipo numérico, que indica el grado en que las creencias actuales están alineadas con las preocupaciones del agente. El lado derecho de la regla expresa la forma en que este valor numérico se calcula de acuerdo con las creencias actuales de agente y cuando se cumpla dará valor a la variable de la parte izquierda de la regla.

- personality: La personalidad del agente se ha modelado de forma que se permita la inserción de varios valores numéricos, denominados *traits*. De esta forma el programador puede emplear el modelo de personalidad que desee. Además puede añadir el grado de racionalidad del agente y la lista de estrategias de afrontamiento o reacciones emocionales (como gestos o expresiones faciales).

- traits: Es una lista de valores numéricos, cada uno representando el grado en que la correspondiente dimensión está presente en la personalidad del agente. Puede estar formada por uno o más valores numéricos, dependiendo del modelo emocional que se quiera representar.

- rat_level: Valor numérico que indica el grado de racionalidad del agente. Cuanto mayor sea, el agente será más racional y menos emocional y viceversa.

- coping_strats: Lista de estrategias de afrontamiento o reacciones emocionales.

- cs: Representa una estrategia de afrontamiento o reacción emocional. Tiene una estructura similar a los planes *Jason*[3] cuyo evento disparador tiene como funtor el símbolo reservado "*cs___*" y dentro de sus atributos debe tener el contexto y la categoría afectiva (condiciones para que la estrategia sea ejecutada).

[2]Se ha modelado siguiendo la sintaxis de las reglas en *Jason* como se explicó en la sección 1.3. Las reglas se emplean para deducir creencias de la base de creencias.

[3]Se ha modelado siguiendo la sintaxis de los planes en *Jason* como se explicó en la sección 1.3.

- aff_categ: Las categorías afectivas se definen en el sistema multi-agente. Se emplean para aclarar la programación, permitiendo al programador expresar un estado afectivo con su propio lenguaje. Por ejemplo, una categoría afectiva podría ser "inseguro".

- beliefs: El compilador de *Jason* tiene un símbolo no terminal denominado *belief* que expresa una creencia. Con este nuevo símbolo se pretende diferenciar las creencias de *Jason* (sección 1.3) de las creencias con probabilidad.

- literal_prob: Representa las creencias con probabilidad que serán empleadas para la representación de las expectativas. En *Jason* las creencias son del tipo `funtor(parámetros)`. Las expectativas deberán tener esta estructura `funtor(parámetros)[prob__(número)]<tiempo inicial, tiempo final>`, es decir una probabilidad y un rango temporal (ver la sección 2.3).

- atomic_formula_prob: Se ha empleado la misma estructura que tiene *atomic_formula* en *Jason*[10]. Como se ha mencionado en el punto anterior se ha añadido el rango temporal.

- time_point_range: El rango temporal estará formado por un tiempo inicial y un tiempo final, que serán del tipo *time_point*.

- time_point: Es un valor numérico que indica el tiempo de las creencias en milisegundos. También se puede expresar como una expresión aritmética.

Para llevar a cabo la realización de este trabajo se hace necesaria la incorporación de parte de la sintaxis presentada en la formalización de *GenIA*^{3} para *Jason*. Concretamente es necesario incluir en el analizador léxico-sintáctico de *Jason* la sintaxis correspondiente a los símbolos: *beliefs*, *literal_prob*, *atomic_formula_prob*, *time_point_range* y *time_point* puesto que son necesarios para la implementación de las expectativas en los agentes *Jason* ya que forman parte de su sintaxis. Inicialmente se planteó la introducción únicamente de las modificaciones necesarias para las expectativas pero para que en el resto del desarrollo no fuera necesario tener que modificar el analizador léxico se decidió añadir la sintaxis completa de *GenIA*^{3} a *Jason* como parte del trabajo de fin de grado.

```
agent                → (init_bels|init_goals)*plans concerns personality
concerns             → "concerns__""(" <VAR> ")"":-" log_expr "."
personality          → "personality__:""{" traits [","rat_level]
                       [","coping_strats] "}""."
traits               → "["<NUMBER>(","<NUMBER>)* "]"
rat_level            → <NUMBER>
coping_strats        → "copingst__:" (cs)+
cs                   → "cs__""(" context "," aff_categ ")""->" body "."
aff_categ            → <ATOM>
beliefs              → ( (literal|literal_prob) "." )*
literal_prob         → [ "~" ] atomic_formula_prob
atomic_formula_prob→ ( <ATOM> | <VAR> ) [ "(" list_of_terms ")" ]
                       [ "[" list_of_terms "]" ] [ time_point_range ]
time_point_range     → "<" arithm_expr "," arithm_expr ">"
```

Figura 2.2: Extensión simplificada de la gramática *EBNF* para *Jason*.

2.3. Expectativas en un agente *Jason*

Este trabajo implementa el módulo de las expectativas y forma parte de un trabajo mucho más amplio para el desarrollo de la formalización de la arquitectura *GenIA*^{3} para el intérprete de *AgentSpeak Jason*. Las expectativas representan un componente mental anticipatorio en el proceso de razonamiento[29]. Junto con las percepciones, las expectativas juegan un papel muy importante en la generación de emociones como son la esperanza, el miedo, la satisfacción o la decepción. El proceso de creación y evaluación de expectativas modifica el estado afectivo del agente, y por lo tanto, influye en su proceso de toma de decisiones.

2.3.1. La influencia de las expectativas en el estado afectivo

En psicología las teorías de evaluación sostienen que las emociones son el resultado de las interpretaciones y explicaciones que cada individuo realiza en base a sus circunstancias, por tanto la evaluación es un proceso de interpretación de la relación de una persona con su entorno[24]. Las emociones pueden ir además acompañadas de respuestas fisiológicas como el sudor o el vello erizado. La psicología ha empleado esta teoría para interpretar y predecir los mecanismos y las pautas de la emociones y las estrategias de afrontamiento. Para poder realizar la evaluación es necesario tener un criterio definido sobre ciertas variables, que se denominan **variables de evaluación** (aunque tam-

bién son conocidas como cheques de evaluación o dimensiones de evaluación) y un conjunto de emociones específicas que se asocian con ciertas configuraciones de estos criterios[36].

Existen varios criterios para evaluar las emociones, entre los comúnmente aceptados se encuentra el criterio de las implicaciones para el futuro. Este criterio se sostiene sobre la teoría de que algunas emociones son sobre cosas que están por venir, como esperanzas o temores, o son reacciones a violaciones de expectativas como por ejemplo, sorpresa o decepción. En el terreno de la computación, las teorías de la valoración para las futuras implicaciones es necesario evaluar la probabilidad, el valor de lo inesperado y la posibilidad de cambio. Por tanto, debe representar los objetivos y las expectativas futuras y debe incluir mecanismos para evaluar la probabilidad de eventos y acciones y sus consecuencias, incluyendo las interacciones entre los posibles resultados. Por ejemplo, la no consecución de un objetivo interfiere con el logro de otro[24].

Un ejemplo de un modelo de proceso de la dinámica de evaluación que emplea las expectativas es el modelo *EMA*[24]. *EMA* proporciona un marco para explotar y explicar la dinámica de las emociones. Es compatible no sólo con distintas teorías de evaluación, sino también con otros procesos cognitivos y de percepción del agente. En *EMA* las expectativas son variables de evaluación y se calculan constantemente. *EMA* distingue entre dos tipos de eventos relacionados con el valor de lo esperado. El primero es cuando existe una expectativa previa y esta se confirma o no. El segundo es cuando no existe una expectativa sobre un evento que si se cumple. Por tanto, realmente lo que se mide es el valor de lo que el agente espera y de lo que para el agente es inesperado. Este valor es empleado para el cálculo de la emoción *sorpresa*. En la arquitectura *GenIA*^{3} se pretende realizar una aproximación a este modelo, por tanto, las expectativas se emplean como un valor de certeza sobre la consecución de un evento futuro cuyo cumplimiento tendrá un efecto sobre el estado de ánimo del agente.

Las expectativas tienen un efecto en las experiencias afectivas reales. Como se justificó en la sección 1.4, las expectativas pueden emplearse para el cálculo del valor de la sorpresa ante una situación. Por ejemplo, en un experimento de laboratorio con humanos se realizo la siguiente prueba[39]: se dividió a los sujetos en dos grupos, a cada grupo se le hizo ver la misma película de dibujos animados. Uno de los grupos tenia expectativas infundadas por parte de los investigadores acerca de que la película era muy divertida. El otro grupo a modo de grupo control no tenia ninguna expectativa infunda-

da. Los resultados obtenidos fueron que las personas que no formaban parte del grupo de control, y por tanto tenían expectativas positivas acerca de la película, se rieron más, calificaron la película como significativamente más divertida que los del grupo de control y tardaron muy poco tiempo en realizar dicha calificación. La conclusión obtenida de estos datos es que cuando los seres humanos tienen una expectativa afectiva y las experiencias que están teniendo no tienen una discrepancia muy alta con las expectativas, estas se cumplen incrementando la emoción en el individuo[23].

Diversos estudios demuestran que las expectativas también tienen una influencia en la memoria ya que generan un recuerdo para futuros eventos. Un sujeto que ya ha vivido una experiencia genera unas expectativas que modificarán su voluntad de repetir la experiencia. Siguiendo con el ejemplo anterior, los sujetos que vieron la película con expectativas de divertirse generaron un recuerdo positivo de esa experiencia, que de repetirse en el futuro generará también unas expectativas positivas que influirán en su deseo de volver a repetir dicha experiencia. Pero no siempre tienen este efecto, de hecho puede producirse el efecto contrario, en caso de que las expectativas produzcan una falsa sensación sobre una experiencia futura. Es decir, un individuo pudo tener en el pasado una mala experiencia, pero sus expectativas hacia la repetición de la experiencia pueden ser positivas y el sujeto podría volver a intentar repetir la experiencia. Por tanto, las expectativas también pueden afectar en la toma de decisiones desacertadas en base a recuerdos sesgados.

Actualmente existen algunas investigaciones sobre el control de las expectativas en *Jason*. Por ejemplo, las realizadas en [29] controlan las expectativas mediante el uso de un monitor que se encarga de realizar el control de las expectativas del agente. El modelo propuesto en este trabajo no emplea ningún tipo de monitor externo, sino que se emplea un nuevo tipo de agente. Este agente implementa de forma interna el control de las expectativas. De esta forma no es necesario el uso de un monitor externo, evitando sus inconvenientes como por ejemplo el interbloqueo o inanición y además el empleo de monitores suele llevar un coste alto asociado en gran parte a la sincronización.

Como se mencionó anteriormente las expectativas representan un componente mental anticipatorio en el proceso de razonamiento. Por tanto tienen un tiempo definido en el cual se espera que se cumplan. Aunque este rango temporal no tiene por que estar claramente definido. Por ejemplo, un alumno de una universidad podría tener la expectativa de aprobar el curso en Junio y así no tener que presentarse a las recuperaciones. Esta expectativa tiene

una fecha límite en la que debe de cumplirse, si pasado el mes de Junio no ha aprobado el curso la expectativa no se cumple. Este incumplimiento de la expectativa producirá en el alumno un cambio en su estado emocional.

Como se mencionó en la sección 1.4, el modelo *OCC* tiene una variable que contempla el valor de lo inesperado para el cálculo de la intensidad de las emociones. Esta variable representa el valor de la sorpresa ante una situación. Pero, la sorpresa puede ser positiva o negativa, es decir, existen tanto sorpresas agradables como desagradables. Por tanto, las expectativas no siempre tienen por qué tener una connotación positiva en caso de cumplirse y negativa en caso contrario. También pueden aparecer en un individuo expectativas negativas, es decir que de cumplirse tienen un efecto negativo en el estado emocional[7, 23, 40, 19]. Siguiendo con el ejemplo anterior, cuando el alumno termina su examen, tiene la expectativa de que lo ha suspendido porque no le ha salido bien. Esta expectativa tiene un rango temporal que finalizará el día que se publiquen las notas de la asignatura. En caso de que la expectativa se cumpla tendrá un efecto negativo en su estado emocional.

Otra parte importante de las expectativas es la probabilidad [40, 19, 15] con la que el individuo espera que se cumplan. Cuanto mayor es la probabilidad de cumplimiento menor es la afección que produce en el individuo. Por ejemplo, en el caso anterior, si el alumno ha estudiado muy poco, la probabilidad de que se cumpla la expectativa negativa de suspender es elevada. Por tanto en el momento en el que se publican las notas y se produce el suspenso, la expectativa se cumple. Pero el estado emocional no sufre un gran cambio debido a que ya espera que se cumpliera con una probabilidad muy alta. Sin embargo, si sucede que aprueba, el estado emocional se ve incrementado en gran medida, pues la expectativa de suspender no se cumple. Al tener esta expectativa una probabilidad elevada de ocurrir, el hecho de no producirse genera en el individuo una modificación del estado emocional más grande y en sentido opuesto, que si se hubiera cumplido la expectativa. En este caso, el estado emocional se verá incrementado pues el no cumplimiento de una expectativa negativa genera sentimientos positivos.

2.3.2. Modelo de expectativas desarrollado

Como se ha mencionado anteriormente en la sección 2.2 las expectativas representan el componente mental anticipatorio que permite calcular el valor de lo inesperado ante la consecuencia de un evento. Al ser un componente antipatorio es necesario definir un tiempo futuro en el cual esas expectativas

puedan llegar a cumplirse. Por tanto, es necesario que tengan un rango temporal donde pueda comprobarse su cumplimiento. Además en un lenguaje de agentes ese rango temporal debe poder ser directamente manipulado por el programador. Por ejemplo, un agente puede tener la expectativa de que dentro de una hora lloverá, está claro que en el momento que se inserta la expectativa su rango temporal deberá abarcar una hora completa. Si durante esa hora llueve la expectativa se da por cumplida pero si finaliza la hora y continua sin llover la expectativa no se cumplirá.

Cuando un sujeto genera una expectativa pensando en un evento futuro, siempre existe un cierto grado de incertidumbre que, como se mencionó anteriormente en la sección 1.4, se puede emplear para el cálculo de la variable de evaluación que tiene en cuenta el valor de lo inesperado. Para poder representar el grado de la incertidumbre es necesario definir de alguna forma la probabilidad a priori de que esa expectativa se cumpla. Esta probabilidad indicará el grado de certidumbre que se tiene sobre el cumplimiento de la expectativa mientras el rango temporal esté activo.

Las expectativas son importantes tanto a la hora de calcular el estado afectivo, como a la hora de tomar decisiones. Por tanto en la formalización de la arquitectura *GenIA*^{3} para agentes *Jason* se propone una sintaxis para la representación de las expectativas dentro del *EBNF* que puede verse en la figura 2.2. Esta sintaxis pretende resolver las necesidades propuestas anteriormente. El modelo propuesto para modelar las expectativas es el siguiente:

```
literal(parámetros)[Prob__(número)]<t1, t2>
```

Esta construcción tiene la misma estructura que una creencia, pero es necesario añadir la anotación `[Prob__(número)]` y el rango temporal en el que deberá estar activa la expectativa. Este rango temporal está representado por dos valores numéricos. `t1` representa el tiempo en el que la expectativa debe iniciarse en el agente. `t2` representa el tiempo en el que la expectativa finaliza. Esta nueva sintaxis permitirá que el programador pueda incorporar de manera sencilla expectativas en los agentes.

Para poder evaluar el desarrollo del módulo de las expectativas sin contar con el resto de la implementación finalizada, ha sido necesario modificar esta sintaxis. Como se comentó en la sección 1.4 la emoción de sorpresa puede ser positiva o negativa, es decir, existen tanto sorpresas agradables como desagradables. Por tanto es necesario que las expectativas reflejen esa consideración. Para poder trabajar con las expectativas se ha realizado una modificación

que consiste en permitir al programador añadir si las expectativas son positivas o negativas. Esta modificación es temporal, pues cuando el módulo de las expectativas se incorpore con el resto de la implementación de *GenIA*^{3} no será necesario explicitar el tipo de las expectativas, pues el agente será capaz de deducirlo por él mismo a través de las preocupaciones (*concerns*). La sintaxis queda de la siguiente forma:

```
literal(parámetros)[Prob__(numéro, tipo)]<t1, t2>
```

donde en tipo se deberá indicar si la expectativa es positiva o negativa. De momento no se ha tomado una decisión de si podría no ser necesaria cuando se incorpore con el resto de módulos, puesto que permite a los programadores una mayor expresividad. Para poder introducir las expectativas en *Jason* es necesario modificar su analizador léxico-sintáctico para incorporar esta nueva sintaxis como se mencionó en la sección 2.2.1.

Capítulo 3

Implementando el uso de expectativas en un agente Jason

3.1. Introducción

En este capítulo se introducen las modificaciones realizadas al código *Jason* con el fin de incluir las expectativas, empleando la especificación *EBNF* que se introdujo en el apartado 2.2. Para ello se ha dividido en dos bloques. El primer bloque describe las modificaciones realizadas en la sintaxis de *Jason* para introducir la gramática *EBNF* propuesta en el modelo *GenIA*^{3} (sección 2.2). En el segundo bloque se desarrollan las modificaciones que ha sido necesario realizar en la estructura del código *Jason* con el fin de permitir el uso de expectativas en los agentes.

3.2. Modificando el analizador sintáctico

Como se ha mencionado anteriormente, *Jason* es una extensión de *AgentSpeak*, y está implementado en el lenguaje de programación *Java*. Para poder leer los ficheros de entorno y de agente, se emplea un analizador sintáctico que implementa la gramática *EBNF* de *Jason*[10]. Hay un analizador para el entorno del sistema multi-agente y otro para los agentes. El código del entorno del sistema multi-agente emplea un fichero con la extensión "*.mas2j*" y los ficheros de agente emplean la extensión "*.asl*". Estos analizadores están desarrollados en *javacc* [21] que es una herramienta que permite generar analizadores sintácticos, especificando la gramática *EBNF* mediante funciones. Una vez definida la sintaxis *javacc* genera automáticamente el código java asociado al fichero analizado.

Uno de los retos iniciales de este proyecto fue precisamente aprender *javacc*. No obstante, gracias a que en la asignatura Lenguajes de programación y procesadores de lenguajes se vieron de forma teórica analizadores léxico-sintácticos, no resultó demasiado complicado adaptar dichos conocimientos adquiridos a la sintaxis de *javacc*.

Para introducir la nueva gramática *EBNF* ha sido necesario introducir los siguientes símbolos léxicos en el analizador sintáctico para los ficheros de agente *AS2JavaParser*: `personality__`, `concerns__`, `copingst__`, `cs__`, `prob__`, `positive__` y `negative__`:

```
173     //personality tokens
174     | <TK_PERSONALITY:  "personality__" >
175     | <TK_CONCERNS:  "concerns__" >
176     | <TK_COPINGST:  "copingst__" >
177     | <TK_CS: "cs__" >
178     | <TK_PROB: "prob__" >
179     | <TK_POSITIVE: "positive">
180     | <TK_NEGATIVE: "negative"
```

El objetivo de este trabajo era incorporar las expectativas, de forma que de ahora en adelante se explicarán solo aquellas modificaciones con este fin. El propósito de la modificación del analizador sintáctico es permitir que el programador pueda introducir expectativas en los agentes. Por ejemplo:

```
tiempo(nublado)[prob__(0.9,positive)] <Now : Now+10000>.
```

que representa que el agente tiene la expectativa de que, con una probabilidad del noventa por ciento, desde ahora (representado como `Now`) hasta dentro de diez segundos (expresado como `Now+10000` donde 10000 representan diez segundos en milisegundos) se insertará la creencia `tiempo(nublado)`. La palabra reservada `positive` indica que la expectativa tiene una connotación positiva para el agente, es decir si se cumple la expectativa tendrá un efecto positivo en el estado emocional del agente. Esta modificación que no estaba prevista en la gramática *EBNF* original, ha sido añadida solo a efectos de clarificar la utilidad de las expectativas, pues en el futuro no será necesario explicitar si es positiva o negativa, ya que el agente lo sabrá gracias a los *concerns*.

Siguiendo las especificaciones de *javacc*, donde cada parte izquierda de cada regla de la gramática *EBNF* se corresponde con una función y la parte derecha de cada regla se corresponde con la implementación de esas funciones, se han hecho las siguientes modificaciones en el analizador léxico-sintáctico de

Jason. Se ha modificado la función *agent()*, que ya estaba implementada, para añadir las preocupaciones (*concerns*) y la personalidad. Lo mismo ocurre con las creencias, se ha modificado el método existente *belief()* para tener en cuenta los nuevos literales con probabilidad (las expectativas). Por último se han creado las nuevas funciones: *concerns()*, *personality()*, *traits()*, *rat_level()*, *coping_strats()*, *cs()*, *aff_categ*, *literal_prob*, *atomic_formula_prob()* y *time_point_range()*. La implementación de estas funciones tiene un patrón muy similar a las expresiones que se representan en el *EBNF* de la figura 2.2. A pesar de que *javacc* está basado en *java* existen algunas diferencias apreciables al comprar la sintaxis de ambos lenguajes, sobre todo en la forma de implementar las funciones. Para ayudar a comprender la sintaxis de *javacc* a continuación se propone el siguiente código perteneciente a la función del no terminal *cs*:

```
1259 CopingStrategy cs(): { Object t; PlanBody B = null; Token k;
1260                CopingStrategy copingst = new CopingStrategy();}
1261 {
1262    < TK_CS >
1263    "("
1264    t = log_expr() {copingst.setContext((LogicalFormula)t);}
1265    ","
1266    k = <ATOM>
1267              {copingst.setAffectCategory(new Atom(k.image));}
1268    ")"
1269    "->"
1270    B = plan_body() { B=(PlanBody)B.getBodyNext();
1271                      copingst.setBody(B);}
1272    "."             { return copingst; }
1273 }
```

Como se menciona en la sección 2.2.1, la sintaxis del no terminal *cs* es la siguiente:

$$\underline{cs} \rightarrow \text{“}cs\_\_\_\text{”“(”}\underline{context}\text{“,”}\ \underline{aff\_categ}\text{“)”“}\rightarrow\text{”}\ \underline{body}\text{“.”}$$

Como puede verse en la línea 1259 las cabeceras de las funciones llevan un valor de retorno (en este caso devuelve un objeto de tipo *CopingStrategy*) y el nombre de la función igual que en *java*, pero van seguidas de dos puntos. Después de los dos puntos es necesario abrir corchetes para la declaración de variables internas de la función, estás variables solo pueden ser definidas en la cabecera. A continuación, igual que ocurre en *java* se abren corchetes donde se debe de ubicar el cuerpo de la función. El cuerpo de la función es la parte derecha de la regla, es decir un símbolo *cs___* (línea 1262) seguido del símbolo paréntesis abierto, seguido de un no terminal denominado *context*,

línea 1264. Como es un no terminal tiene una función que lo implementa, en este caso es *log_expr()*. Como puede verse el valor de la función se guarda en una de las variables locales previamente definidas ya que este valor es necesario para poder realizar acciones semánticas. Las acciones semánticas se definen dentro de corchetes mediante el uso de código *java*. Por ejemplo, siguiendo con el contexto, el valor devuelto por la función *log_expr()* es de tipo *object* y se almacena en la variable **t**. El valor de la variable **t** es empleada para asignar el contexto a la *copingStrategy* mediante el uso de la función *setContext(LogicalFormula l)*.

Siguiendo con la definición del no terminal *cs*, una vez se ha completado el contexto, viene el símbolo coma (línea 1265), seguido de la categoría afectiva [1], en este caso un átomo, y se realiza la asignación de la categoría afectiva a la estrategia de afrontamiento. Sigue con un paréntesis cerrado, el símbolo menos seguido del símbolo mayor. A continuación viene el cuerpo de la estrategia de afrontamiento, es decir, las acciones que deberá realizar el agente cuando se active dicha estrategia de afrontamiento. La forma de finalizar el cuerpo de una estrategia de afrontamiento es empleando el símbolo punto y por tanto ya se puede realizar el retorno de la función que en este caso devuelve una estrategia de afrontamiento como un objeto de la clase *CopingStrategy*.

Además ha sido necesario incorporar una función `prob()`, ya que es necesaria para poder leer la nueva anotación[2] `[prob__(probabilidad, tipo)]` que afecta directamente a los atributos del agente. Esta ampliación que no estaba propuesta en el *EBNF* se ha tenido que incluir debido a que las anotaciones implementadas en *Jason* no están preparadas para la introducción de este tipo de sintaxis. La solución encontrada a este problema ha sido crear una interfaz denominada *EmTerm* y una clase que la extiende *EmtermImpl*. Esta clase contiene dos atributos, uno de tipo numérico donde se almacena la probabilidad, y el otro de tipo lógico donde se almacena si la expectativa es positiva o negativa. A continuación se muestra el código para la función *prob()*:

[1] La categoría afectiva en el *EBNF* original se correspondía con el no terminal *aff_categ*, pero a la hora de realizar la implementación, viendo que la categoría afectiva solo podía tomar el valor de un átomo y su implementación no aportaba nada, se decidió directamente tomar la categoría afectiva como un átomo. Por eso en lugar de aparecer el no terminal aparece el símbolo reservado en la gramática para los átomos.

[2] Las anotaciones en *Jason* se emplean para tener información adicional de las creencias, como por ejemplo el origen de dicha creencia, si es propio o viene de las percepciones o por parte de otro agente.

```
1174     EmTerm prob():  { Token n; EmTermImpl emTerm; Double d; }
1175 {
1176     <TK_PROB>
1177     "("
1178     n = <NUMBER>    {
1179                       d = Double.parseDouble(n.image);
1180                       emTerm = new EmTermImpl(d); }
1181     [
1182      ","
1183      (<TK_POSITIVE>
1184      |
1185      <TK_NEGATIVE>  {emTerm.setPositive(false); }
1186      )
1187     ]
1188     ")"
1189                     { return emTerm; }
1190 }
```

Como puede verse en el código, esta función devuelve un *EmTerm*, de esta forma se podrá asignar, con posterioridad, el valor de la probabilidad y el tipo a la expectativa que se está analizando. Lo primero que debe leer es el símbolo *prob___* seguido de un paréntesis abierto. Dentro del paréntesis hay un número con la probabilidad de la expectativa, una coma y el tipo, que puede ser positivo o negativo (líneas 1183-1186).

Como se ha mencionado al inicio, *javacc* genera automáticamente el código *java* correspondiente al analizador léxico-sintáctico, por tanto para que se puedan añadir los nuevos atributos es necesario, además de todo lo que se ha comentado anteriormente, modificar las clases del analizador sintáctico. A continuación se va a hacer un recorrido de las clases modificadas justificando la necesidad de ese cambio:

- *Literal*: El formato escogido para la sintaxis de las expectativas es el mismo que para las creencias salvo por el rango temporal, en *Jason* este tipo de sintaxis se denominan literales. Inicialmente se planteó la idea de crear un nuevo tipo de literal, que heredara de *Literal*, pero para poder llevarlo a cabo, dado que *java* no tiene herencia múltiple, era necesario duplicar demasiadas clases para conseguir un correcto funcionamiento del sistema. Es por ello que se decidió añadir los cambios directamente a la clase *Literal*. Por tanto se han creado tres nuevos atributos, *probability* de tipo numérico, que contiene la probabilidad de la expectativa, *initialTime* de tipo *TimePoint* y *finalTime* de tipo *TimePointFinal*, que implementan el rango temporal inicial y final,

en el cual puede cumplirse la expectativa. Inicialmente estos últimos atributos eran de tipo *Long* y contenían el tiempo que se leía directamente desde el analizador, pero, para dotar de una mayor expresividad al programador, se decidió crear dos nuevas clases *TimePoint* y *TimePointFinal*.

- *TimePoint*: Esta clase se emplea para recoger las expresiones aritméticas que se usan para definir los rangos temporales. Esta clase contiene cuatro atributos: *exp* de tipo *ArithExpr* que contendrá la expresión aritmética leída por el analizador léxico, *Units* es un carácter que contiene el tipo de unidad temporal que se emplea en la expresión que por defecto será de milisegundos, un atributo de tipo *Long* denominado *time* donde se almacenará el tiempo cuando se evalúe la expresión aritmética y será el atributo empleado para comprobar el cumplimiento de las expectativas y por último el atributo *userTime* que contiene el tiempo que ha introducido el usuario respetando las unidades que haya empleado, este atributo también es de tipo *Long* y solo se emplea para el método *toString*, de tal forma que, si el usuario quiere mostrar por pantalla las expectativas, se mostrarán tal como el usuario las escribió, en lugar de mostrarlas con el tiempo en milisegundos que no resulta demasiado esclarecedor. Para que el usuario pueda introducir el tiempo actual en el rango temporal sin tener que calcularlo, se ha introducido la palabra reservada *Now*, de forma que si aparece en la expresión aritmética se sustituye por el tiempo del sistema en el momento de la evaluación[3].

- *TimePointFinal*: Esta clase hereda de *TimePoint*, la diferencia es que se permite, aparte de la palabra *Now*, la palabra *Infinite* para representar las expectativas con un futuro no definido. Por ejemplo, una persona puede tener la expectativa de que algún día viajará a Egipto. Pero ese día no esta definido en el futuro. Por eso *Infinite* permite a las expectativas tener un tiempo "infinito"[4].

- *EmTerm*: Como se ha mencionado anteriormente, se ha incluido un nuevo tipo de anotación. En este caso la anotación modifica directamente un atributo del agente, por lo que ha sido necesario crear un

[3]La evaluación de las expresiones aritméticas se realiza en el momento en el que se insertan las expectativas dentro de la lista de expectativas del agente como se explicará en la sección 3.4. De esta forma se garantiza que el rango temporal de las expectativas empezará a contar a partir del momento de su inserción.

[4]Para representar el infinito se ha empleado el *Long.MAX_VALUE* que es el tamaño máximo de *Long*, que es un número finito.

nuevo tipo de término, denominado *EmTerm*. Esta interfaz extiende la interfaz *Term*. *EmTerm* contiene un campo *probability* donde se insertará la probabilidad de la expectativa en el momento en el que se recoja por parte del analizador léxico. Esto permite que cuando se realiza la asignación de las anotaciones a la expectativa, se compruebe si hay alguna que sea del tipo *EmTerm*, en cuyo caso se modifica el campo correspondiente de la clase *Literal*. Además se ha añadido otro atributo denominado *positive* de tipo lógico que indica si la expectativa tiene una connotación positiva o negativa para el agente.

- *EmTermImpl*: Esta clase implementa los métodos de la interfaz *EmTerm*.

- *Range*: Cuando se está leyendo el rango temporal, igual que ocurría con la anotación en las expectativas, es necesario almacenarlo para posteriormente poder asignarlo a la expectativa. Se ha creado la clase *Range* explícitamente para almacenar rangos de tiempo. Por tanto la función que calcula el rango temporal en el analizador devuelve un objeto de tipo *Range*. Esta clase cuenta con cuatro atributos: *min* y *max*, que son de tipo doble, y se emplean para almacenar rangos explicitados como números, mientras que *tpf* y *tp*, son atributos de tipo *TimePoint* y se emplean para almacenar los rangos que se implementan mediante expresiones aritméticas. Por ejemplo, en el analizador léxico-sintáctico de *Jason* se ha creado una función *time_point_range()* que devuelve un objeto de este tipo:

```
1271 Range time_point_range(): { Object tp; Object tpf; }
1272 {
1273         "<"
1274         tp   = arithm_expr()
1275         ":"
1276         tpf  = arithm_expr()
1277         ">"
1278         {return new Range(tp, tpf);}
1279 }
```

- *Atom*: Cuando el analizador lee el fichero de especificación del agente crea los objetos correspondientes. Por ejemplo, si lee la creencia *tiempo(nublado)* crea un objeto de tipo *Literal* con sus atributos correspondientes. El problema es que cuando va a proceder a usar esa creencia añadiéndola a la base de creencias, no se realiza una copia del objeto Literal, si no que se crea un nuevo objeto de tipo *Atom*, y este objeto contiene todos los atributos necesarios para esa creencia. Si la creencia vuelve a repetirse en el futuro se creará otro nuevo objeto de esta clase.

Por tanto para poder emplear el tiempo y la probabilidad en las expectativas, ha sido necesario modificar el constructor de la clase *Atom*, para que, de forma explicita hiciera copia de los atributos de tiempo y probabilidad, necesarios para las expectativas. Esta técnica de replicar objetos nos ha facilitado el cálculo del rango de tiempo de las expectativas, ya que hasta el momento de la inserción en la lista de expectativas, dentro del método *brf* (que se explica más adelante en la sección 3.3), es donde se instancia la palabra reservada *Now* con el tiempo actual del sistema. De esa forma se permite insertar expectativas durante la ejecución indicando el tiempo que deben estar activas, abstrayendo al programador del cálculo del tiempo en el que se insertarán en la lista de expectativas del agente.

- *Structure* ha sido necesario modificar el método *toString()* para añadir la estructura temporal de las expectativas de forma que al imprimirlas por pantalla el usuario pueda ver la misma estructura que escribió en el código del agente.

3.3. Modificando la semántica

Las expectativas tienen un rango de tiempo limitado en el que están activas, por lo que se hace necesario realizar una evaluación periódica de su cumplimiento. Por esta razón se ha decidido insertar un nuevo estado en el ciclo de razonamiento de los agentes. Este nuevo estado tiene la función de evaluar todas las expectativas en cada ciclo, de forma que si una expectativa se cumple en su rango temporal se añade a la lista de expectativas cumplidas del agente, o si por el contrario el rango de tiempo ha pasado se añade como expectativa no cumplida. En ambos casos se realizan las acciones necesarias para modificar el estado emocional con la nueva información.

Este nuevo estado del ciclo de razonamiento recibe el nombre de *EvalExp* y se ha insertado entre *ProcMsg* y *SelEv*. Como se ha mencionado anteriormente es necesario realizar una comprobación del cumplimiento de las expectativas en todos los ciclos de razonamiento, pues de no hacerlo así podrían descarte expectativas que sí se habrían cumplido, pero el rango temporal ya no esta activo en el instante de evaluación. Por tanto es necesario realizarlo en un estado que se procese en cada ciclo. Inicialmente se decidió insertarlo después del estado *StartRC(Start Reasoning Cicle)* ya que este estado, como su nombre indica, es el primero que se ejecuta en cada ciclo de razonamiento, pero durante el desarrollo del trabajo se modificó la versión de *Jason* con la

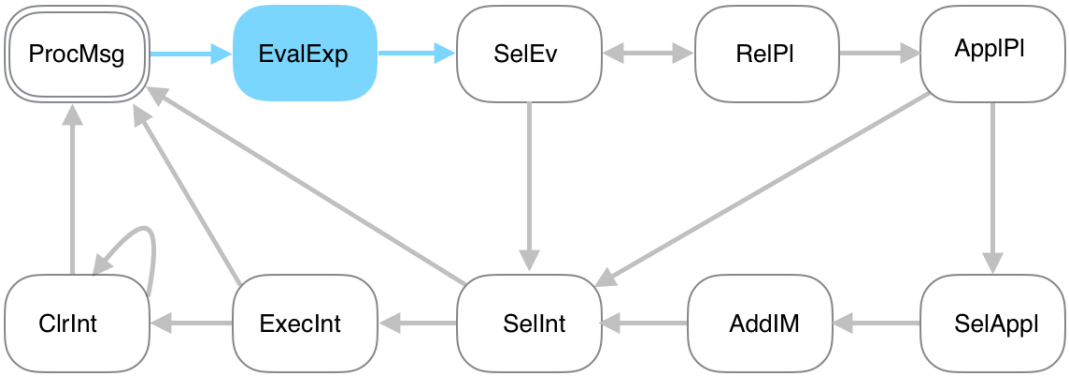

Figura 3.1: Ciclo de razonamiento *Jason* con el nuevo estado.

que se estaba trabajando para actualizarlo a la versión 4.2, en la que el primer estado del ciclo de razonamiento fue sustituido por *ProcMsg*. Por tanto, se ha elegido el estado *ProcMsg* para que cuando finalice se inicie el nuevo estado *EvalExp*.

Una vez decidida la inclusión del nuevo estado, se ha creado una función denominada *applyEvalExp()*:

```
314 private void applyEvalExp() throws JasonException{
315   confP.stepSense = State.SelEv;
316   synchronized (getAg().getBB().getLock()) {
317     Iterator<Literal> expectations =
318             getAg().getBB().getExpectations();
319     BeliefBase beliefs = getAg().getBB();
320     while(expectations.hasNext()){
321        Literal exp = expectations.next();
322        if(exp.getFinalTime().getTime() <
323             System.currentTimeMillis()){
324          expectations.remove();
325          logger.fine("TS -> the expectation "
326            + exp.getFunctor().toString()
327            + " not is fulfilled");
328          if(exp.isPositiveExpectation())
329            getC().addNotFulfilledExpectationsP(exp);
330          else getC().addNotFulfilledExpectationsN(exp);
331        }
332        else if(exp.getInitialTime().getTime() <=
333             System.currentTimeMillis() &&
334             beliefs.contains(exp) != null){
335          expectations.remove();
336          logger.fine("TS -> the expectation "
337            + exp.getFunctor().toString()
338            + " is fulfilled");
```

```
339             if(exp.isPositiveExpectation())
340                   getC().addFulfilledExpectationsP(exp);
341             else getC().addFulfilledExpectationsN(exp);
```

Lo primero que se realiza en esta función, siguiendo la filosofía de *Jason*, es transitar al estado *SelEv*, como puede verse en la línea 315, para volver a restablecer el ciclo de razonamiento. Para realizar la comprobación de todas las expectativas, lo primero es obtener las expectativas de la base de creencias, como puede verse en la línea 317, se obtiene como un iterador. Una vez obtenida la lista de expectativas se recorre para comprobar si alguna de las expectativas se han cumplido. Para ello, primero se comprueba si su rango temporal ha finalizado (línea 322) sin que se haya cumplido, en caso afirmativo se elimina de la lista de expectativas y se comprueba si es positiva o negativa y se añade a la lista de expectativas no cumplidas correspondiente (líneas 328-330). En caso de que el rango temporal esté activo, es decir, el tiempo inicial sea menor que el actual y el tiempo final sea mayor que el actual, se comprueba si hay alguna creencia que haga que se cumpla la expectativa (que exista la misma creencia con el mismo nombre y los mismos parámetros dentro de la base de creencias) en cuyo caso se elimina de la lista de expectativas y se añade a la lista de expectativas cumplidas correspondiente.

Estas listas de expectativas cumplidas y no cumplidas, se incorporan a la clase *Circumstance*, mediante la inclusión de cuatro listas de literales: *notFulfilledExpectationsP, notFulfilledExpectationsN, fulfilledExpectationsP y fulfilledExpectationsN*. Como puede verse hay dos listas para las expectativas que no se satisfacen, y dos para las que sí. La letra final "P" indica que son expectativas positivas mientras que la "N" indica que son negativas. Además se ha añadido un nuevo atributo *AS* de tipo *List<Double>* que contendrá los valores que representan el estado emocional. Actualmente solo se utiliza el primer elemento de la lista, ya que por el momento el estado emocional se representa con un número entero. Cuando se llama al constructor de *Circumstance* se inicializa el primer valor de la lista a un valor neutro[5].

La clase *EmAgent* extiende a la clase *Agent* creando tres nuevos atributos *affTs* de tipo *AffectiveTransitionSystem* que como su nombre indica es el sistema de transición afectivo, *affTempDyn* de tipo *AffectTemporalDynamic* que serán empleados en desarrollos posteriores, por lo que no afectan al tra-

[5]Conforme el estado se acerca al mínimo aumenta la tristeza y cuando se acerca al máximo aumenta la felicidad, mientras esté en el estado neutral no estará ni triste ni contento

bajo actual. El tercer y último atributo es un *Map<String,AffectCategories>* de *Java* que contendrá las categorías afectivas denominado *affectiveCategory*. También se han sobrescrito los métodos *selectOption* y *brf*. El primero se ha reescrito para permitir el uso de la anotación para etiquetas de planes `[affect__(State)]` dentro del código del agente, para que los planes puedan ser seleccionados solo si el agente se encuentra en ese estado emocional. En *Jason* todos los planes tienen asociado un nombre por defecto, pero puede ser asignado por el usuario escribiendo `@Nombre_Plan` directamente antes del inicio del plan. Esto permite que el usuario pueda hacer referencia directa a un plan mientras el agente se está ejecutando. Pero una etiqueta también puede llevar anotaciones y es en las anotaciones donde se puede incluir información que permita al intérprete elegir un plan antes que cualquier otro. Por eso, al incluir la etiqueta `[affect__(State)]` dentro del nombre del plan, se permite seleccionar, de los planes que hayan sido activados por un evento disparador, aquellos que tengan como estado emocional el mismo que el agente. Por ejemplo:

```
1 @Triste[affect__(1)]
2 +!mostrarEstadoAfectivo
3 ->
4 .print("triste").
5
6 @Feliz[affect__(5)]
7 +!mostrarEstadoAfectivo
8 ->
9 .print("feliz").
```

El primer plan tiene como nombre **Triste** y solo será aplicable si el estado afectivo del agente tiene el valor de uno. Mientras que el segundo, tiene la etiqueta **Feliz** y solo será aplicable si el estado afectivo del agente tiene valor cinco. Como puede verse ambos planes tienen el mismo evento disparador y gracias a las etiquetas se pueden diferenciar. Esto permite una programación más sencilla, puesto que solo hay un nombre para el evento disparador de *mostrarEstadoAfectivo*. Si no se pusieran las etiquetas sería necesario realizar la comprobación del estado afectivo dentro del propio plan, o tener mas de un plan con distinto nombre para cada estado afectivo. Ambas soluciones complican el código y lo alargan de forma innecesaria. Por eso es importante la creación de esta etiqueta y la sobre sobrescritura del método *selectOption()*.

Aunque no era objetivo de este trabajo, para abordar la influencia del estado afectivo en el comportamiento se ha decidido mostrar el uso de las expectativas empleando un ejemplo que demostrase visualmente la influencia de las expectativas en los agentes afectivos. Para ello se ha decidido imple-

mentar el estado emocional mediante una sola dimensión, con cinco estados emociones posibles repartidos de la siguiente forma: uno representará el estado triste, cinco el estado feliz, tres el estado de equilibrio o neutro y los estados dos y cuatro representarán emociones intermedias. El método *selectOption* deberá ser revisado en el futuro para incorporar la sintaxis del estado emocional en la selección de planes.

En cuanto a la justificación de modificación del método *brf* de la clase *EmAgent*, para dotar de una mayor robustez a la comprobación del cumplimiento de las expectativas dentro del rango temporal, partiendo de la premisa de que el ciclo de un agente no tiene un tiempo fijo, se decidió en un primer momento incluir en el sistema de transiciones dos nuevos atributos: `getPrevCLockReading` y `clockReading`. Para controlar el tiempo de duración de cada ciclo de razonamiento y así comprobar si las expectativas que se salieron de rango se habrían cumplido si el ciclo hubiera sido más corto, pero añadía al problema una mayor complejidad, ya que se hacía necesario llevar una lista de las expectativas no cumplidas durante el ciclo, e introducía cierta incertidumbre. Por esta razón se buscó una manera alternativa. Finalmente se decidió, además de comprobar las expectativas en el nuevo estado del ciclo de razonamiento *EvalExp*, comprobarlas cuando se van a insertar o a eliminar creencias, ya que cabe la posibilidad de que se activen expectativas después de insertar o eliminar una creencia y antes de realizar el paso *EvalExp*. El método que realiza la inclusión o eliminación de creencias en un agente *Jason* se denomina *brf* (*Belief revision function*) y se encarga de actualizar la base de creencias del agente a partir de las creencias que recibe y de las actuales. Entre los parámetros que recibe el método *brf* se encuentra la creencia o la expectativa que se va a añadir o eliminar.

La primera acción que realiza este método es recoger de la base de creencias la lista de expectativas:

```
151 public List<Literal>[] brf(Literal beliefToAdd, Literal
    beliefToDel,
152        Intention i, boolean addEnd) throws
    RevisionFailedException {
153     Iterator<Literal>  expectations = getBB().getExpectations
    ();
```

La manera de saber si la llamada a este procedimiento es para insertar o eliminar, simplemente consiste en comprobar cual de los dos parámetros, *beliefToAdd* o *beliefToDel*, es distinto de `null`. En el caso de la inserción de una creencia, es decir, que el campo *beliefToAdd* no es `null`, se comprueba si es una expectativa:

```
167     if (beliefToAdd.getInitialTime() != null ||
168         beliefToAdd.getProbability() != 1){
169       beliefToAdd.getInitialTime().eval();
170       beliefToAdd.getFinalTime().eval();
171       getBB().addExpectation(beliefToAdd);
```

Para distinguir las creencias de las expectativas es necesario que las expectativas tengan una probabilidad que sea distinta de uno, puesto que al ser una predicción de un evento futuro no pueden tener una certeza absoluta, ya que de tenerla sería una creencia. También es necesario que tengan un rango temporal, pues como se ha dicho antes, son predicciones que deben cumplirse en el futuro. Esta comprobación se realiza en la línea 167, en caso de que sea una expectativa se realiza en ese momento la evaluación, mediante la función *eval()* (líneas 169-170), del tiempo inicial y final. Esto es importante realizarlo justo en este momento, pues es cuando el rango temporal de la expectativa va a definirse con el tiempo actual actual del sistema. Por tanto en este momento es cuando se le debe de asignar a la palabra reservada *Now* el tiempo actual del sistema y evaluar las expresiones aritméticas que introdujo el usuario en la definición del rango temporal de la expectativa. A continuación puede verse el código de la función *eval()* de la clase *TimePointFinal*[6]:

```
37 public void eval() throws NoValueException{
38     if(exp != null) {
39         Unifier u = new Unifier();
40         u.bind(new VarTerm("Now"),
41             new NumberTermImpl(System.currentTimeMillis()));
42         u.bind(new VarTerm("Infinite"),
43             new NumberTermImpl(Long.MAX_VALUE));
44         Term t = exp.capply(u);
45         time = (long) ((NumberTerm)t).solve();
46     }
47 }
```

Cuando se invoca ese método se evalúa la expresión aritmética que determina el valor del intervalo del rango temporal. Como puede verse en las líneas 39-43, se crea un nuevo objeto de tipo *Unifier* que se emplea para unificar los lexemas *Now* e *Infinite* con cualquier aparición en la expresión aritmética. Para ello se utiliza el procedimiento *bind(VarTerm v, NumberTermImpl n)*, este método crea una relación entre el lexema y el valor numérico que, posteriormente será empleada para sustituir los lexemas por ese valor. Esta

[6]El método *eval()* de la clase *TimePoint* es idéntico a este salvo por las líneas 42 y 43 que hacen referencia a la palabra reservada *infinite* que, por coherencia, solo se permite en la definición del tiempo final de la expectativa.

unificación se hace efectiva mediante el método *capply()* (línea 44) que intercambia el valor de los lexemas que están contenidos en el unificador por su valor numérico asociado. Por último se calcula el resultado de la expresión aritmética y se asocia a la variable global *time*.

Tras el inciso para la explicación de la función *eval()* se procede a continuar con el método *brf*. Anteriormente se explicó lo que sucedía cuando llegaba una expectativa para añadir. Sin embargo también hay que considerar que el método también puede recibir una creencia para insertar. La inserción de las creencias se ha dejado igual que en el código original de *Jason* salvo por que, al insertar una creencia se comprueba si la nueva creencia hace que alguna expectativa se cumpla. Para ello, se realiza un recorrido de la lista de expectativas del mismo modo que se hace en el método *applyEvalExp()* de la clase *TransitionSystem* mencionado arriba.

En caso de que *beliefToAdd* sea `null`, quiere decir que *beliefToDel* es distinto de `null`. Aunque las expectativas se eliminan de forma automática cuando se cumplen o cuando se acaba su rango temporal, también se permite la eliminación explicita. Esta forma de eliminar las expectativas se corresponde con las teorías de evaluación, ya que las expectativas pueden variar a lo largo del tiempo conforme se acerca su cumplimiento. Por ejemplo, un alumno que empieza la carrera puede tener la expectativa inicial de aprobar todas las asignaturas del primer semestre. Conforme avanza el curso esa expectativa puede desaparecer, o modificarse antes de que entre en rango temporal. Si el alumno suspende un examen, su expectativa de aprobar todas seguramente varíe a aprobar algunas, o simplemente disminuya la probabilidad de que esa expectativa se cumpla. Para tener en cuenta esta consideración, se hace la misma comprobación que se hacia antes para comprobar si es una expectativa, es decir si tiene probabilidad y rango temporal, en caso afirmativo se elimina de la lista de expectativas. En caso de que sea una creencia la que se va a eliminar, igual que ocurría en la inserción, se comprueba si esa creencia hace que se cumpla alguna expectativa antes de eliminarla. De esta forma se asegura que a la hora de insertar y eliminar creencias, se evalúa el cumplimiento de las expectativas.

3.4. Añadiendo las expectativas a la base de creencias

El objetivo principal de este trabajo es la gestión de expectativas. Hasta ahora se ha hablado de su tratamiento pero no de donde se almacenan mientras están activas. Los agentes *Jason* disponen de una base de creencias, donde se encuentran todas las creencias que tiene el agente, tanto las iniciales como las que ha adquirido de otro agente, de las percepciones o mediante su propio razonamiento. Inicialmente, y dado que las expectativas tienen el mismo formato que una creencia, se decidió añadirlas directamente a la base de creencias. Pero al realizar las primeras pruebas apareció un problema a la hora de realizar las búsquedas para comprobar el cumplimiento de las creencias tanto en el método *brf* como en el nuevo paso del ciclo de razonamiento.

El problema venía a la hora de comprobar el cumplimiento de las expectativas. Para comprobar si una expectativa se cumple es necesario buscar una creencia que tenga la misma estructura, y los métodos que existen para comparar creencias no tienen en cuenta las anotaciones. Por tanto con la estructura sintáctica planteada en este trabajo para representar las expectativas no era posible que las búsquedas produjeran la confirmación o no de la expectativa pues la propia expectativa se empleaba para la comprobación. Para poder solventar este problema se valoraron diferentes alternativas, entre las que estaba por ejemplo modificar el código de la búsqueda para que tuviera en cuenta las anotaciones, pero esto producía un mayor coste temporal puesto que era necesario aumentar el número de iteraciones en cada búsqueda al tener que recorrer para cada creencia su lista de anotaciones. Finalmente se decidió crear una lista aparte de la lista de creencias, donde se almacenan las expectativas. De esa manera comprobar si una expectativa se cumple es tan fácil como preguntar si está en la lista de creencias haciendo uso de la función *contains(Litreal)*. Esta función devuelve el valor lógico correspondiente a si la base de creencias contiene o no el literal.

La lista de expectativas se ha incluido directamente en la clase abstracta *BeliefBase*. De esta forma se permite que las expectativas puedan ser empleadas desde cualquiera de las implementaciones de la base de creencias. En este caso la implementación se ha realizado en la clase *DefaultBeliefBase* puesto que, como su nombre indica, es la clase por defecto para las creencias.

Capítulo 4

Ejemplo del uso de un agente emocional

4.1. Introducción

A efectos de demostrar el correcto funcionamiento de las expectativas en los agentes *Jason*, se ha desarrollado un experimento basado en el juego de las siete y media. Este tipo de juegos son muy útiles a la hora de realizar experimentos sobre las emociones, pues juegan un papel muy claro en la toma de decisiones. Además, los juegos permiten comprobar las variaciones en el estado emocional, pues los jugadores normalmente se alegran cuando ganan y se enfadan cuando pierden. Para poder llevar a cabo el experimento se ha empleado una baraja de naipes española de cuarenta cartas. El juego consiste en obtener siete puntos y medio, o acercarse a ello lo más posible. Las cartas valen tantos puntos como el valor indicado en la carta, excepto las figuras, que valen medio punto [20]. Uno de los jugadores será la banca, que será el que repartirá las cartas a los jugadores. Cada jugador puede tener solo una carta tapada. En cada turno el jugador decide si quiere pedir otra carta (boca abajo o boca arriba) o plantarse. Si consigue siete y medio lo muestra, si se pasa debe descartarse y perderá inmediatamente esa ronda.

Partiendo de este juego se ha diseñado una versión reducida realizando una serie de asunciones: el jugador siempre tendrá tapada la primera carta que recibe, el AS se considera siempre como un punto y solo habrá un jugador que jugará contra la casa. Este juego fue desarrollado como parte de las prácticas de la asignatura Técnicas de Inteligencia Artificial, mediante un sistema experto implementado en el lenguaje de programación *Jess Rules*[22]. *Jess Rules* es una herramienta que provee un entorno para el desarrollo de

sistemas de reglas basado en *Java*. El jugador humano jugaba contra el sistema experto y durante el turno del jugador se le preguntaba si quería más cartas o bien si quería plantarse. En caso de que se plantara jugaba el turno la casa, que estaba programada para pedir carta mientras tenia menos de cinco y medio, y cuando llegaba a cinco y medio si tenía los mismos puntos o menos que los visibles del jugador seguía pidiendo.Se ha realizado una adaptación de ese juego para *Jason*, pero en esta ocasión el jugador será un agente emocional (del tipo *EmAgent*) en lugar de un humano y la banca será un agente *Jason* normal. A continuación se explicará el código de la banca.

4.2. Código de la banca

La banca[1] tiene cinco tipos de creencias iniciales, *playerPoints(0)* contiene los puntos reales del jugador y se emplea para distinguir los puntos totales, *visiblePPoints(0)* contiene los puntos que puede ver la banca, es decir sin incluir la carta que el jugador tiene tapada. La creencia *housePoints(0)* sirve para almacenar los puntos de la banca en cada ronda. La creencia *cards(L)* donde "L" es una lista del uno al cuarenta, representa la baraja, y por último *card(16,6,"seis de espadas")*, donde el primer dígito es el identificador de esa carta que se corresponde con uno de los cuarenta de la lista anterior, el segundo dígito son los puntos de la carta y el tercer atributo es la descripción de la carta.

Al inicio de la partida la banca queda a la espera de que algún jugador pida una carta. Para ello el jugador envía un mensaje del tipo *tell* con la creencia *+playerAskCard* que dispara el plan cuyo código es el siguiente:

```
50  +playerAskCard[source(player)] : cards(L)
51  <-
52    -playerAskCard[source(player)];
53    .random(X);
54    N = math.ceil(X*.length(L));
55    .nth(N-1,L,NewCard);
56    .delete(N-1,L,NewList);
57    -+cards(NewList);
58    !updatePlayerPoints(NewCard);
59    printCard(NewCard);
60    if(card(NewCard,Val,Str)){
61      .send(player,tell,card(Val));
62    }.
63
```

[1]El código completo puede verse en el apéndice A

```
64
65      +!updatePlayerPoints(N) : card(N,P,S) & playerPoints(
        Points) & visiblePPoints(VPP)
66      <-
67          -+playerPoints(Points + P);
68          if(Points > 0){
69              -+visiblePPoints(VPP + P);
70          }else{
71              +hidden(N);
72          }.
```

Lo primero que se hace es eliminar la creencia (como puede verse en la línea 52), lo que permitirá que se vuelva a lanzar el evento cuando se reciba otro mensaje igual y además se evita almacenamiento innecesario, puesto que una vez procesada la acción no aporta nada al agente. Lo siguiente es calcular la carta que se va a repartir al jugador, de forma aleatoria empleando la función interna *.random(X)*, que devuelve en la variable "X" un valor entre cero y uno que posteriormente, en la linea cincuenta y cuatro, se transforma a un valor entre uno y la longitud de la lista de cartas que quedan en el mazo. De esta forma se obtiene un indice que apunta a un valor de dicha lista, y así se extrae un identificador de carta. Se elimina ese identificador de la lista de cartas (línea 57), se actualizan los puntos del jugador mediante la función *updatePlayerPoints(N)* y se llama al entorno para que imprima la carta mediante la función *printCard(NewCard)* que se explicará en la sección 4.5. Por último se envía al jugador un mensaje con el valor de la carta.

A continuación se muestran el resto de planes que pueden ser activados. Cuando la banca recibe un mensaje del jugador, si añade la creencia *bust*, indica que el jugador se ha pasado de siete y medio y por lo tanto se debe de dar por concluida la ronda. Si es una creencia *playerWin*, es porque el jugador ha obtenido siete y medio y por lo tanto ha ganado. Estos dos primeros mensajes producen un reinicio del juego mediante la llamada a la acción *!restartGame*. Y por último si es del tipo *stand* significa que el jugador se ha plantado por lo que el turno pasa a la banca (en este caso *House* empleando la nomenclatura inglesa) mediante la acción *!houseAskCard*.

```
74      +bust[source(player)]
75      <-
76          -bust[source(player)];
77          playerBust(0);
78          .send(player,tell,lose);
79          !restartGame.
```

Igual que ocurría en el plan asociado a la creencia `playerAskCard`, en la acción `houseAskCard` se selecciona una carta, y se actualiza la puntuación

de la banca con la acción `!updateHousePoints(NewCard)`, se selecciona la siguiente acción. Si tiene más de siete y medio comunica al jugador que ha ganado mediante un mensaje de tipo `tell` con la creencia `win`. Si no, mientras tenga menos de cinco coma cinco o menos puntos que los visibles del jugador, pide carta. En caso contrario se planta lanzando la acción `!houseStand`, que simplemente compara los puntos del jugador con los de la banca y envía un mensaje al jugador con el resultado. Por último la acción `!restartGame` reinicia todas las creencias iniciales del agente que se comentaron al inicio de esta sección.

```
135     +!restartGame
136     <-
137         -+cards([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
138                  17,18,19,20,21,22,23,24,25,26,27,28,29,
139                  30,31,32,33,34,35,36,37,38,39,40]);
140         -+playerPoints(0);
141         -+visiblePPoints(0);
142         -+housePoints(0);
143         -hidden(_);
144         .send(player,achieve,restart).
```

4.3. Código del jugador

El jugador[2] tiene dos creencias iniciales: *myPoints(0)* que equivale a los puntos actuales del jugador y *ganadasSeguidas(0)* que indica, cuantas partidas seguidas ha ganado. A diferencia de la banca, el jugador es un agente emocional y tiene las siguientes expectativas iniciales:

```
ganar(ronda)[prob__(0.5,positive)] <Now : Now + 15000>.
conseguir(7.5)[prob__(0.1,positive)] <Now : Infinite>.
ganadasSeguidas(2)[prob__(0.2,positive)] <Now : Infinite>.
```

El jugador está esperando ganar esta ronda con una probabilidad de cero coma cinco, conseguir siete y medio con una probabilidad de cero coma uno y ganar dos rondas seguidas con una probabilidad de cero coma dos. El rango de tiempo de estas dos últimas se ha dejado hasta el infinito porque puede cumplirse en cualquier ronda. Al ser quien inicia la ronda comunicándose con la banca, tiene un objetivo inicial denominado `!init` que se emplea para iniciar la primera ronda. Las siguientes rondas empezarán cuando se inserte el objetivo `!start`. Esta distinción es necesaria ya que al inicio del juego se necesita un tiempo de espera extra para que se inicie la interfaz gráfica como

[2]El código completo puede verse en el apéndice B

se explica en la sección 4.5. Cada vez que se añade el objetivo `!start` se envía un mensaje a la banca de tipo `tell` con la creencia `playerAskCard`, que, como se comentó antes dispara la acción que calcula la carta y se la envía al jugador. Cada vez que se inserta en el jugador la creencia `card(valor de carta)` se activan una serie de posibles planes. Cada uno lleva asociada una etiqueta del tipo `[affect__(número)]` que, como se explicó en la sección 3.3, permite que se seleccione el plan que se corresponde con el estado emocional actual. Por ejemplo, si el estado emocional fuera dos, se seleccionaría el siguiente plan:

```
36  @p2[affect__(2)]
37  +card(Val)[source(house)] : myPoints(N)
38  <-
39      .wait(500);
40      -card(Val)[source(house)];
41      -+myPoints(N+Val);
42      !siguienteJugada(5.0,N+Val).
```

Este plan modifica los puntos actuales y añade un nuevo objetivo en el agente del tipo `!siguienteJugada` donde el primer parámetro indica con cuantos puntos debe plantarse, y el segundo parámetro son los puntos actuales. Todos los planes que se disparan cuando se inserta la creencia `card` realizan exactamente las mismas acciones, la única variación son los puntos con los que debe plantarse. De esta forma las decisiones se ven afectadas por el estado emocional. Cuanto más positivo sea el estado emocional más arriesgará el jugador para intentar alcanzar siete y medio.

Cuando se inserta el objetivo `!siguienteJugada` se comprueba el estado emocional mediante la acción interna `ia.readEmState(State)` que deja en la variable `State` el estado emocional actual (la explicación de esta función puede verse en la sección 4.4) y se imprime ese estado emocional haciendo una llamada al entorno mediante la acción interna `emEstate(State)` (esta función esta definida en la sección 4.5). Igual que ocurría con la banca, se hace una selección de la siguiente acción dependiendo de los puntos actuales y se notifica la decisión tomada mediante un mensaje a la banca.

```
68  +!siguienteJugada(Plantarse, N)
69  <-
70      ia.readEmState(State);
71      emEstate(State);
72      if((N) > 7.5){
73          .send(house,tell,bust);
74      }
75      else{
76          if((N) < Plantarse){
77              .send(house,tell,playerAskCard);
```

```
78              }else{
79                  if((N) == 7.5){
80                      .send(house,tell,playerWin);
81                      +conseguir(7.5);
82                  }else{
83                      .send(house,tell,stand);
84                  }
85              }
86          }.
```

Se han definido dos tipos de creencias que cuando se añaden hacen que se cumplan las dos expectativas que se definieron con un rango temporal infinito: `conseguir(7.5)` y `ganarSeguidas(2)`. Cuando se añaden a la lista de creencias del jugador activan un plan que elimina dichas creencias y se vuelven a insertar las expectativas que se han cumplido. De esa forma se permite que el agente siempre tenga estas dos expectativas en su lista de expectativas. Con este ejemplo se hace visible como es posible la insertar expectativas en cualquier momento de la ejecución.

```
88      +conseguir(7.5)
89      <-
90          -conseguir(_);
91          +conseguir(7.5)[prob__(0.9,positive)] <Now : Infinite
    >.
```

Cuando el jugador recibe por parte de la banca la creencia `win` quiere decir que el jugador ha ganado la ronda y por tanto se activa un plan que añade la creencia `ganar(ronda)` que permite que se cumpla la expectativa del mismo nombre. Además, se comprueba el estado emocional y se envía la orden al entorno para que lo muestre en la interfaz gráfica. De la misma manera existe un plan similar para cuando recibe la creencia `lose`, con la diferencia de que inserta la creencia `perder(ronda)`, por tanto la expectativa `ganar(ronda)` no se cumplirá.

```
94      +lose[source(house)]
95      <-
96          -+ganadasSeguidas(0);
97          -lose[source(house)];
98          ia.readEmState(State);
99          emEstate(State);
100         .print("PLAYER LOSE");
101         +perder(ronda);
102         playerLose(0).
```

Por último cuando se recibe el objetivo `!restart` de parte de la banca, se comprueba el estado emocional y se imprime por pantalla, se eliminan las creencias de ganar y perder, y se espera durante nueve segundos. Esta espera sirve para controlar el no cumplimiento de las expectativas, ya que

hasta que no acaba su rango temporal no se pueden dar por no cumplidas. Una vez finaliza la espera se vuelve a insertar la expectativa de que ganará esta ronda, y además se añade una expectativa nueva:

card(0.5)[prob___(0.3, positive)]<Now : Now + 10000>;

El jugador espera, con una probabilidad de cero coma tres, obtener en la ronda que va a empezar alguna carta que tenga un valor de medio punto con una probabilidad de cero coma tres. La diferencia temporal que se observa con las expectativas iniciales es debida a que ahora ya no es necesario hacer una espera para que se inicie la interfaz gráfica.

4.4. Acciones Internas

Se han creado dos acciones internas para mostrar las expectativas y obtener el estado emocional del agente durante la ejecución. En primer lugar, la acción `printCurrentExpectations` (apéndice D) tiene un parámetro de tipo *String* para que el usuario pueda mostrar un mensaje antes de que se impriman las expectativas de forma que sea más sencilla la depuración. Se imprime el mensaje por pantalla y posteriormente se accede a la lista de expectativas del agente y se imprime por consola.

En segundo lugar, la acción interna que permite obtener el estado afectivo actual es `readEmState`. Esta acción inicialmente realizaba un cálculo del estado emocional mediante la diferencia de las listas de expectativas cumplidas y no cumplidas de la clase *Circumstance*. Como aproximación simple tiene sentido que las expectativas anteriores afecten al estado actual (era una forma de representar la memoria emocional). El problema es que cuando las listas crecen se produce una convergencia, ya que el cumplimiento de una expectativa no afecta tanto cuando las listas adquieren cierto tamaño. Por tanto cuando se realizó el primer experimento se comprobó que en todas las ejecuciones, tras varias rondas, no producía ningún efecto en el estado emocional que se cumplieran las expectativas o no, por lo que el usuario no recibía información visual y además resultaba poco realista. Para solucionar este problema se decidió dar un paso más en la implementación, usando la lista *AS* (vista en la sección 3.2). Se realizaron las modificaciones oportunas en los métodos de inserción de expectativas cumplidas y no cumplidas, para que el estado emocional se incrementara o disminuyera dependiendo de la probabilidad de la expectativa, limitándolo entre cinco y uno. Por ejemplo, si se añade una expectativa positiva porque se ha cumplido, se modifica el

estado emocional al mínimo entre cinco y el estado emocional actual más el complementario de la probabilidad[3] de esa expectativa. En la línea 320 puede verse la modificación del estado afectivo:

```
317 public void addFulfilledExpectationsP(Literal l){
318         System.out.println("se ha cumplido la expactiva
    positiva " + l.toString());
319         fulfilledExpectationsP.add(l);
320         AS.set(0, Math.min(AS.get(0) + (1-l.getProbability())
    ,5));
321     }
```

4.5. Entorno

Para poder usar el entorno en *Jason* es necesario emplear la clase *Environment* que implementa métodos que se ejecutan desde dentro del propio código del agente, como por ejemplo `executeAction` que permite ejecutar las acciones internas de comunicación con el entorno que se vieron en las secciones 4.2 y 4.3. *Jason* tiene implementadas varias soluciones para el uso de interfaces gráficas. Pero después de realizar un análisis de los requisitos de la interfaz se llegó a la conclusión de que las alternativas que ofrecía *Jason* no cubrían las necesidades, por ello se decidió crear una interfaz gráfica y un entorno propios, la interfaz se ha desarrollado con *Processing*[32] y la comunicación entre *Processing* y *Jason* se ha hecho mediante el uso de comunicaciones empleando los *sockets* de *Java* como se verá más adelante en la sección 4.5.1.

Para poder crear un entorno propio con acciones internas propias es necesario rescribir los métodos `init` y `executeAction`. En `init` se añaden todas las acciones necesarias para que se inicie el entorno, en este caso simplemente se activa el servidor creando un objeto de la clase *Server*. En el método `executeAction` que tiene como parámetros un *String* con el nombre del agente que ejecuta la acción, y la propia acción se hace un comprobación de cual es la acción que debe ejecutarse, como puede verse en el siguiente fragmento de código:

```
18  if (action.getFunctor().equals("printCardH")) {
19         int x = (int)((NumberTerm)action.getTerm(0)).solve();
```

[3]Se usa el complementario por la consideración de que si una expectativa tiene una probabilidad alta de que se cumpla, en caso de cumplirse no afectará tanto al estado emocional ya que se esperaba que se cumpliera, sin embargo si tiene la probabilidad baja, es decir no se espera que se cumpla y se cumple debe de afectar en mayor grado al estado emocional

```
20          view.drawHCard(x);
21  } else {
```

Se comprueba si el funtor[4] es equivalente a `printCardH`, en caso afirmativo se extrae el primer parámetro que contiene el número de la carta que se va a imprimir y se llama a la función correspondiente de la clase *Server*.

4.5.1. Comunicación del entorno con la interfaz gráfica

Para poder comunicar *Processing* con *Jason* mediante red se ha creado una nueva clase dentro del ejemplo denominada *Server*. La clase *Server* simplemente crea un nuevo servidor en el puerto 5204 y se queda a la espera de que se conecte un cliente, en este caso el cliente es la interfaz de *Processing*. Como se comentó en la sección 4.3 se hace una espera por parte del jugador para dar tiempo a que se inicie la interfaz y se conecte al servidor. Para poder enviar todas las ordenes de dibujo a *Processing* se ha creado una codificación numérica que permite que cada orden se empaquete en un solo byte. Así se consigue minimizar el coste de las comunicaciones. Cuando *Processing* recibe un byte lo descodifica mediante la función `tratarDatos(int datos)` y así sabe en cada momento qué tiene que dibujar. Por ejemplo:

```
55 //Server Code
56     public void playerBust() throws IOException{
57             salida.writeInt(255);
58     }

21 //Processing Code
22 void draw(){
23     if (myClient.available() > 0) {
24         dataIn = myClient.read();
25         tratarDatos(dataIn);
26     }
27     [...]

57     if(pBust) text("Player Bust ", 30, 200);

65 void tratarDatos(int d){
66     [...]

82     if(d == 255) pBust = true;
```

Cuando *Processing* recibe el byte "255" realiza la impresión por pantalla de la cadena `"Player Bust"` mediante la función `text(Cadena_de_texto, posición_x, posición_y)` es decir que el jugador se ha pasado. Lo mismo se ha hecho para cada uno de los mensajes que debe mostrar en pantalla. Así

[4]En *Jason* el funtor equivale al nombre o identificador de los literales

pues para cada acción que emita un agente, la clase *Server* tiene un método que envía un byte a *Processing*.

4.5.2. Interfaz gráfica con Processing

Processing es un software de código abierto basado en *Java* para la creación de diseños multimedia e interactivos[32]. Contiene varias funciones que permiten crear un entorno gráfico de forma muy sencilla. Por ejemplo, para dibujar un cuadrado basta con usar la función `rect(posición_en_x, posición_en_y, ancho, alto)`. Hay dos métodos principales que deben ser reescritos por el usuario. En el método `setup()` se deben insertar las funciones que el usuario quiere que solo se hagan una vez al inicio. Por ejemplo, definir el tamaño de la ventana con `size(ancho,alto)`. El método `draw()` se ejecuta en un bucle infinito, mientras la aplicación esté en funcionamiento, y se encarga del refresco de los objetos en la pantalla y en el que se deben insertar todas las primitivas gráficas que se quieren mostrar en pantalla constantemente.

Se ha usado una baraja de libre dominio con licéncia *GNU* (recortando las cartas una por una) numerando del uno al cuarenta para usar la misma codificación que se emplea en los agentes (como puede verse en la definición de creencias de la banca sección 4.2 y A). De esta forma se facilita la carga de las imágenes. También se han recortado *emoticonos* para ponerle cara al jugador y que pueda verse en su expresión el estado de ánimo en el que se encuentra. Nuevamente se han almacenado con la misma codificación que los agentes: del uno al cinco. Como puede verse en la figura 4.1 hay cinco *emoticonos* uno por cada estado de ánimo del agente emocional. Están ordenados de izquierda a derecha de forma que el que está en el extremo izquierdo es el estado uno que se corresponde con el estado triste, el que está en el extremo derecho es el estado cinco, que se corresponde con el estado feliz, el central es el estado de equilibrio, y los restantes son los estados de transición (se podrían representar como un poco triste en el caso del segundo y alegre en el el caso del cuarto).

El resultado de la interfaz puede verse en la figura 4.2. En la parte superior está el jugador con sus cartas. La cara del jugador se muestra en la esquina superior izquierda de la pantalla e inicialmente está en el estado neutral o de equilibrio, que se corresponde con el estado tres. La primera carta que se reparte al jugador (como se explicó en la sección 4) está tapada, pero para que se sepa que hay debajo se ha decidido tapar solo la mitad de la carta, permitiendo mostrar el resto. Pero ha de quedar claro que esta carta no puede ser

Figura 4.1: Caras que expresan los distintos estados de ánimo del agente.

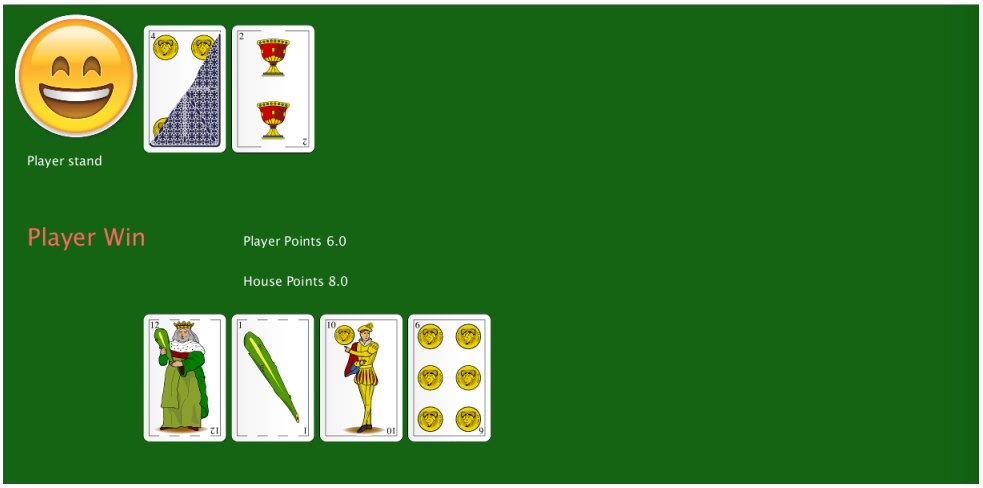

Figura 4.2: Interfaz gráfica. Situación del tablero para usuario (parte superior) y banca (parte inferior)

vista por la banca, por tanto cuando sea su turno no la tendrá en cuenta a la hora de realizar su jugada. En la parte inferior aparecen las cartas de la banca cuando llega su turno. En la parte derecha aparecen una serie de mensajes que describen el desarrollo de la parida. Debajo de la imagen del jugador aparecerán los mensajes *Player Stand* es decir el jugador se ha plantado y *Player Bust* que significa que se ha pasado de siete y media. Debajo de este texto aparece si el jugador gana o pierde. Cuando la casa se planta aparece también un mensaje de *House Stand* a la derecha de sus cartas. En la parte central de la pantalla aparecen los puntos tanto de jugador como de la banca.

En cuanto al código de la interfaz gráfica, se han creado dos objetos de tipo *PImage* denominados "*img*"y "tapada". El primero se emplea para la carga de las imágenes que se van a imprimir. El segundo para almacenar la carta partida que se emplea para la carta oculta. Para conectarse con el servidor se ha creado un objeto de la clase *Client*. También se han creado

dos listas *cardP* y *cardH* que contendrán los identificadores de las cartas del jugador y de la banca respectivamente. Se ha creado una variable de tipo entero que almacena el estado emocional del uno al cinco, para poder mostrar las imágenes convenientemente guardadas con los mismos números. Se han creado dos variables de tipo entero, *pPoints* y *hPoints* para almacenar la puntuación tanto del jugador como de la banca. Y por último se han creado cinco variables booleanas *hStand, pStand, pWin, pLose* y *pBust* que se emplean en la impresión de los mensajes *"House Stand", "Player Stand", "Player Win", "Player Lose"* y *"Player Bust"* respectivamente.

En el método `setup()` se define el tamaño de la ventana, se conecta al servidor y se define la tasa de muestreo a treinta fotogramas por segundo (*FPS*), ya que la aplicación no requiere una tasa mayor de refresco de pantalla. Por último se carga la imagen de la carta tapada como se mencionó anteriormente.

```
11 void setup() {
12   size(1220, 600);
13   myClient = new Client(this, "localhost", 5204);
14   frameRate(30);
15   tapada = loadImage("cartas/50.png");
16 }
```

En el método `draw()` se comprueba si se han recibido datos desde el servidor. En caso afirmativo se llama a la función `tratarDatos`. Se pinta el fondo de color verde simulando un tapete de cartas mediante la función `background`, que recibe tres valores al color verde en formato *RGB*.

```
21 void draw() {
22     if (myClient.available() > 0) {
23         dataIn = myClient.read();
24         tratarDatos(dataIn);
25     }
26     background(20,100,20);
```

Para imprimir las imágenes se cargan desde el fichero y se muestran por pantalla empleando la función `image(PImage img,int x, int y)`, que dibuja la imagen que se le pasa como primer argumento en las coordenadas "x" e "y". Para las cartas se hace un recorrido por las listas de cartas de la banca y del jugador y se van cargando y mostrando por pantalla. Por último se muestra la puntuación y se imprimen los mensajes que estén activos, es decir aquellos cuya variable lógica esta a `true`.

La función `tratarDatos` se encarga de la decodificación del byte que se recibe. Esto se realiza comprobando de qué tipo es: si es menor que cien se

trata de una carta del jugador; si está entre cien y doscientos se trata de un estado de ánimo; si esta entre doscientos y doscientos cincuenta se trata de una carta de la banca; y de docientos cincuenta a doscientos cincuenta y cinco se trata de impresión de mensajes (como por ejemplo si el jugador ha ganado sería el doscientos cincuenta y tres).

Capítulo 5

Conclusiones y futuras ampliaciones

Las emociones juegan un papel muy importante en el proceso de toma de decisiones en los seres humanos. Actualmente no existen demasiadas formalizaciones que permitan la implementación de agentes afectivos. La mayoría de las investigaciones realizadas en este ámbito suelen quedarse en el terreno teórico y muy pocas son las que se ponen en práctica. En este proyecto se ha realizado una extensión del intérprete de *AgentSpeak Jason* para introducir las expectativas en los agentes como parte del desarrollo de *GenIA*^{3}, una arquitectura de propósito general para agentes inteligentes afectivos. Las expectativas representan un componente mental anticipatorio en el proceso de razonamiento y juegan un papel muy importante en la generación de emociones como son la esperanza, el miedo, la satisfacción o la decepción. Las expectativas modifican el estado afectivo del agente, y éste a su vez, influye en su proceso de toma de decisiones.

La extensión de *Jason* realizada en este trabajo permite al programador introducir expectativas en el propio código de definición del agente, posibilitando la definición del rango temporal, la probabilidad y su connotación (positiva o negativa). Las expectativas pueden ser añadidas de forma similar a las creencias iniciales o las notas mentales dentro del cuerpo de una acción. Se han creado los procesos necesarios para la correcta evaluación de las expectativas llevando un control de aquellas que se cumplen y de las que no se cumplen permitiendo modificar el estado afectivo en función del modelo de afecto empleado. Por ejemplo, para la realización del experimento se ha definido un modelo afectivo unidimensional con cinco posibles estados y la transición entre estados se lleva a cabo cuando se evalúa el cumplimiento de las expectativas.

Aun así, de los experimentos se han extraído conclusiones relevantes para el desarrollo del resto del proyecto. Como se explica en el capítulo 4, la primera aproximación que se propuso para evaluar la forma en la que las expectativas influían en el estado de ánimo no era del todo correcta, pues la consideración de calcular el estado de ánimo a partir del histórico almacenado de expectativas cumplidas y no cumplidas producía que el agente alcanzara el estado de equilibrio emocional en pocas iteraciones. Este resultado demuestra que, a la hora de calcular el estado emocional basándose en los eventos pasados, si se emplea todo el histórico de eventos no se produce la respuesta deseada en el agente. La alternativa fue probar qué ocurriría si no existiera ninguna historia, es decir, el estado emocional variaría de forma lineal, incrementándose o decrementándose.

Esta conclusión ratifica hallazgos previos en la teoría de emociones, donde se demuestra la utilidad de las emociones como ayuda en el acotado de posibilidades e información a almacenar. Estos resultados coinciden con las teorías de evaluación pues, como se mencionó en la sección 2.3, las expectativas tienen influencia en la memoria ya que generan un recuerdo para futuros eventos. Por tanto será necesario investigar la mejor forma de emplear los datos almacenados en el histórico de expectativas pero teniendo en cuenta solo ciertas expectativas que puedan ser relevantes para el cálculo del estado de ánimo en un momento determinado. Por otra parte, los resultados obtenidos hacen patente la necesidad de refinar el proceso de actualización del estado afectivo, pues el agente es demasiado volátil, es decir, cambia muy rápidamente de un estado de ánimo a otro en función del cumplimiento de las expectativas.

Para poder probar el ejemplo era necesario decidir el grado en el que las expectativas modificarían el estado de ánimo. Se probaron dos alternativas y se evaluaron los resultados obtenidos. La primera consistía en realizar incrementos o decrementos de una unidad. De esta forma todas las expectativas tenían el mismo peso. Pero esta forma de modificar el estado de ánimo no está acorde con la forma en la que sucede en el ser humano, donde pueden aparecer expectativas con mayor o menor grado de certeza y el estado de ánimo se modifica de acuerdo con esa incertidumbre. Por tanto, se llevó a cabo un proceso de investigación para averiguar la mejor forma de modificar el estado de ánimo en base al grado de incertidumbre de las expectativas.

Finalmente se decidió realizar un incremento y decremento en base a la probabilidad de una expectativa, es decir, aquellas expectativas que tienen una probabilidad alta de cumplirse, tienen menos efecto en el estado de ánimo, que aquellas que tienen una probabilidad baja. De esta forma las expec-

tativas más probables tienen menor grado de incertidumbre para el cálculo del estado de ánimo. Por ejemplo, si un alumno tiene una expectativa de suspender un exámen con un noventa por ciento de probabilidad, si aprueba, su estado de ánimo se incrementará en un alto grado, pues es algo que no esperaba. Sin embargo si se confirma la expectativa su estado de ánimo empeorará poco, pues se puede decir que tenía el suspenso asumido. A pesar de ser un experimento realizado solo con el módulo de las expectativas los resultados y las conclusiones obtenidas han sido bastante útiles para el desarrollo del resto del proyecto.

Por lo que respecta al desarrollo de las expectativas se ha creado una jerarquía de clases siguiendo la filosofía empleada en *Jason* lo que facilitará la continuidad y el mantenimiento del proyecto. Durante el desarrollo con *Jason* aparecieron bastantes problemas, algunos de ellos produjeron modificaciones importantes en el diagrama de clases inicial ya que el código no sigue una estructura clásica en capas. Por ejemplo, se decidió renunciar a la implementación de la clase *Literal_prop*, propuesta como alternativa a la clase *Literal* para creencias con probabilidades y rangos temporales, porque la herencia y las interdependencias que tenía la clase *Literal* complicaba demasiado su desarrollo ya que se hacía necesario modificar gran parte del código original de *Jason*.

A pesar de que *Jason* tiene herramientas para la generación de interfaces y de que el paquete de instalación del propio *Jason* incorpora varios ejemplos de uso, estas herramientas ofrecen bastantes limitaciones a la hora de mostrar imágenes, sobretodo debido al volumen de imágenes necesario para el experimento. Además al empezar a trabajar con imágenes el código empezó a hacerse cada vez más complejo. La solución fue emplear *Processing* que se estudió en la asignatura Diseño e Interconexión de Periféricos. Al incorporar *Processing* los problemas para mostrar las imágenes en la interfaz se solucionaron aunque fue necesario establecer un protocolo de combinaciones entre la interfaz y el código *Jason*.

Dentro de las futuras ampliaciones la más inmediata es la incorporación del módulo de expectativas a la plataforma *GenIA*^{3}. Gracias a que el diseño del módulo de las expectativas se concibió desde un principio para esta integración, será una tarea trivial en el momento que *GenIA*^{3} esté preparado. En cuanto a la gestión de las expectativas, deberán realizarse las modificaciones necesarias para que el agente sea capaz de deducir cuando una expectativa es positiva o negativa empleando sus preocupaciones. Además del uso de la probabilidad, también sería interesante incorporar una forma de introducir

la relevancia de cada expectativa. Para finalizar, aunque de momento los resultados que se han obtenido son provisionales, y solo tienen sentido si se enmarcan en la demostración del uso de las expectativas, cuando este módulo se una al resto del proyecto *GenIA*^{3} permitirá extraer mejores resultados y las expectativas tendrán una mayor utilidad en los procesos afectivos del agente.

Apéndice A

Agente Banca “house.asl”

```
1   // Agent house in project ejemplo
2
3   /* Initial beliefs and rules */
4   cards([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
5          17,18,19,20,21,22,23,24,25,26,27,28,29,
6          30,31,32,33,34,35,36,37,38,39,40]).
7   card(1,1,"uno de oros").
8   card(2,2,"dos de oros").
9   card(3,3,"tres de oros").
10  card(4,4,"cuatro de oros").
11  card(5,5,"cinco de oros").
12  card(6,6,"seis de oros").
13  card(7,7,"siete de oros").
14  card(8,0.5,"sota de oros").
15  card(9,0.5,"caballo de oros").
16  card(10,0.5,"rey de oros").
17  card(11,1,"uno de espadas").
18  card(12,2,"dos de espadas").
19  card(13,3,"tres de espadas").
20  card(14,4,"cuatro de espadas").
21  card(15,5,"cinco de espadas").
22  card(16,6,"seis de espadas").
23  card(17,7,"siete de espadas").
24  card(18,0.5,"sota de espadas").
25  card(19,0.5,"caballo de espadas").
26  card(20,0.5,"rey de espadas").
27  card(21,1,"uno de bastos").
28  card(22,2,"dos de bastos").
29  card(23,3,"tres de bastos").
30  card(24,4,"cuatro de bastos").
31  card(25,5,"cinco de bastos").
32  card(26,6,"seis de bastos").
33  card(27,7,"siete de bastos").
```

```
34      card(28,0.5,"sota de bastos").
35      card(29,0.5,"caballo de bastos").
36      card(30,0.5,"rey de bastos").
37      card(31,1,"uno de copas").
38      card(32,2,"dos de copas").
39      card(33,3,"tres de copas").
40      card(34,4,"cuatro de copas").
41      card(35,5,"cinco de copas").
42      card(36,6,"seis de copas").
43      card(37,7,"siete de copas").
44      card(38,0.5,"sota de copas").
45      card(39,0.5,"caballo de copas").
46      card(40,0.5,"rey de copas").
47      playerPoints(0).
48      visiblePPoints(0).
49      housePoints(0)./* Initial goals */
50
51      /* Plans */
52      +playerAskCard[source(player)] : cards(L)
53      <-
54          -playerAskCard[source(player)];
55          .random(X);
56          N = math.ceil(X*.length(L));
57          .nth(N-1,L,NewCard);
58          .delete(N-1,L,NewList);
59          -+cards(NewList);
60          !updatePlayerPoints(NewCard);
61          printCard(NewCard);
62          if(card(NewCard,Val,Str)){
63              .send(player,tell,card(Val));
64          }
65          .
66
67      +!updatePlayerPoints(N) : card(N,P,S) & playerPoints(
    Points) & visiblePPoints(VPP)
68      <-
69          -+playerPoints(Points + P);
70          if(Points > 0){
71              -+visiblePPoints(VPP + P);
72          }else{
73              +hidden(N);
74          }.
75
76      +bust[source(player)]
77      <-
78          -bust[source(player)];
79          playerBust(0);
80          .send(player,tell,lose);
81          !restartGame.
```

```
82
83      +playerWin[source(player)]
84      <-
85          -playerWin[source(player)];
86          .send(player,tell,win);
87          !restartGame.
88
89      +stand[source(player)]
90      <-
91          -stand[source(player)];
92          playerStand(0);
93          !houseAskCard.
94
95      +!houseAskCard : cards(L) & visiblePPoints(VPP)
96      <-
97          .random(X);
98          N = math.ceil(X*.length(L));
99          .nth(N-1,L,NewCard);
100         .delete(N-1,L,NewList);
101         -+cards(NewList);
102         if(card(NewCard,Val,Str)){
103             printCardH(NewCard);
104         }
105         !updateHousePoints(NewCard);
106         .wait(500);
107         if(housePoints(Points) & Points > 7.5){
108             .send(player,tell,win);
109             !restartGame;
110         }else{
111             if(housePoints(Points) & Points >= 5.5){
112                 if(Points > VPP){
113                     !houseStand;
114                 }
115                 else{
116                     !houseAskCard;
117                 }
118             }else{
119                 !houseAskCard;
120             }
121         }.
122
123     +!updateHousePoints(N) : card(N,P,S) & housePoints(Points
    )
124     <-
125         -+housePoints(Points + P).
126
127     +!houseStand : playerPoints(P) & housePoints(H)
128     <-
129         houseStand(0);
```

```
130         if(P > H){
131             .send(player,tell,win);
132         }else{
133             .send(player,tell,lose);
134         }
135         !restartGame.
136 
137     +!restartGame
138     <-
139         -+cards([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
140                  17,18,19,20,21,22,23,24,25,26,27,28,29,
141                  30,31,32,33,34,35,36,37,38,39,40]);
142         -+playerPoints(0);
143         -+visiblePPoints(0);
144         -+housePoints(0);
145         -hidden(_);
146         .send(player,achieve,restart).
```

Apéndice B

Agente jugador “player.asl”

```
1     // Agent sample_agent in project ejemplo
2
3     /* Initial beliefs and rules */
4     myPoints(0).
5     ganadasSeguidas(0).
6
7     /* expectations */
8
9     ganar(ronda)[prob__(0.5,positive)] <Now : Now + 15000>.
10    conseguir(7.5)[prob__(0.1,positive)] <Now : Infinite>.
11    ganadasSeguidas(2)[prob__(0.2,positive)] <Now : Infinite
      >.
12
13    /* Initial goals */
14
15    !init.
16
17    /* Plans */
18    +!init
19    <-
20        ia.printCurrentExpectations("LAS EXPECTATIVAS
      ACTUALES SON:");
21        .wait(5000);
22        !start.
23
24    +!start
25    <-
26        .send(house,tell,playerAskCard).
27
28    @p1[affect__(1)]
29    +card(Val)[source(house)] : myPoints(N)
30    <-
31        .wait(500);
```

```
32          -card(Val)[source(house)];
33          -+myPoints(N+Val);
34          !siguienteJugada(4.5,N+Val).
35
36      @p2[affect__(2)]
37      +card(Val)[source(house)] : myPoints(N)
38      <-
39          .wait(500);
40          -card(Val)[source(house)];
41          -+myPoints(N+Val);
42          !siguienteJugada(5.0,N+Val).
43
44      @p3[affect__(3)]
45      +card(Val)[source(house)] : myPoints(N)
46      <-
47          .wait(500);
48          -card(Val)[source(house)];
49          -+myPoints(N+Val);
50          !siguienteJugada(5.5,N+Val).
51
52      @p4[affect__(4)]
53      +card(Val)[source(house)] : myPoints(N)
54      <-
55          .wait(500);
56          -card(Val)[source(house)];
57          -+myPoints(N+Val);
58          !siguienteJugada(6.0,N+Val).
59
60      @p5[affect__(5)]
61      +card(Val)[source(house)] : myPoints(N)
62      <-
63          .wait(500);
64          -card(Val)[source(house)];
65          -+myPoints(N+Val);
66          !siguienteJugada(6.5,N+Val).
67
68      +!siguienteJugada(Plantarse, N)
69      <-
70          ia.readEmState(State);
71          emEstate(State);
72          if((N) > 7.5){
73              .send(house,tell,bust);
74          }
75          else{
76              if((N) < Plantarse){
77                  .send(house,tell,playerAskCard);
78              }else{
79                  if((N) == 7.5){
80                      .send(house,tell,playerWin);
```

```
81                      +conseguir(7.5);
82                  }else{
83                      .send(house,tell,stand);
84                  }
85              }
86          }.
87
88      +conseguir(7.5)
89      <-
90          -conseguir(_);
91          +conseguir(7.5)[prob__(0.9,positive)] <Now : Infinite
    >.
92
93
94      +lose[source(house)]
95      <-
96          -+ganadasSeguidas(0);
97          -lose[source(house)];
98          .print("PLAYER LOSE");
99          ia.readEmState(State);
100         emEstate(State);
101         +perder(ronda);
102         playerLose(0).
103
104     +win[source(house)] : ganadasSeguidas(N)
105     <-
106         -+ganadasSeguidas(N+1);
107         -win[source(house)];
108         .print("PLAYER WIN");
109         ia.readEmState(State);
110         emEstate(State);
111         +ganar(ronda);
112         playerWin(0).
113
114     +ganadasSeguidas(2)
115     <-
116         -+ganadasSeguidas(0);
117         +ganadasSeguidas(2)[prob__(0.8,positive)] <Now :
    Infinite>.
118
119     +!restart[source(house)] : myPoints(N)
120     <-
121         ia.readEmState(State);
122         emEstate(State);
123         -ganar(_);
124         -perder(_);
125         .wait(9000);
126         .print("");
127         .print("-------------------------");
```

```
128         .print("NUEVA RONDA");
129         -+myPoints(0);
130         restart(0);
131         ia.readEmState(EState);
132         emEstate(EState);
133         +ganar(ronda)[prob__(0.5,positive)] <Now : Now +
    10000>;
134         +card(0.5)[prob__(0.3)]<Now : Now + 10000>;
135         ia.printCurrentExpectations("LAS EXPECTATIVAS
    ACTUALES SON:");
136         !start.
```

Apéndice C

Sistema multi-agente “ejemplo.mas2j”

```
1     MAS ejemplo {
2
3         infrastructure: Centralised
4         environment: env.MarsEnv
5         agents:
6             house [verbose=1];
7             player [verbose=1] agentClass jason.Em.
    asSemantics.EmAgent;
8
9         aslSourcePath:
10            "src/asl";
11
12    }
```

Apéndice D

Código de la acción interna *printCurrentExpectations*

```
1  // Internal action code for project ejemplo
2
3  package ia;
4
5  import java.util.Iterator;
6
7  import jason.*;
8  import jason.asSemantics.*;
9  import jason.asSyntax.*;
10
11 public class printCurrentExpectations extends
       DefaultInternalAction {
12
13     @Override public int getMinArgs() { return 1; }
14     @Override public int getMaxArgs() { return 1; }
15     //private int maxEmValue = 5;
16     //private int minEmValue = 1;
17     @Override protected void checkArguments(Term[] args)
       throws JasonException {
18         super.checkArguments(args); // check number of
       arguments
19         if ( args.length != 1)
20             throw JasonException.createWrongArgument(this,"
       This function takes one argument.");
21     }
22
23     @Override
24     public Object execute(TransitionSystem ts, Unifier un,
       Term[] args) throws Exception {
25
26         checkArguments(args);
```

```
27          Iterator<Literal> expectations = ts.getAg().getBB().
    getExpectations();
28          String msg = args[0].toString();
29          System.out.println(msg.substring(1,msg.length()-1));
30          while(expectations.hasNext()){
31              Literal exp = expectations.next();
32              System.out.println(exp.toString());
33          }
34          System.out.println("");
35          return true;
36      }
37  }
```

Apéndice E

Código de la acción interna *readEmState*

```
1  // Internal action code for project ejemplo
2
3  package ia;
4
5  import jason.*;
6  import jason.asSemantics.*;
7  import jason.asSyntax.*;
8
9  public class readEmState extends DefaultInternalAction {
10     @Override public int getMinArgs() { return 1; }
11     @Override public int getMaxArgs() { return 1; }
12     //private int maxEmValue = 5;
13     //private int minEmValue = 1;
14     @Override protected void checkArguments(Term[] args)
       throws JasonException {
15         super.checkArguments(args); // check number of
       arguments
16         if ( !args[0].isVar())
17             throw JasonException.createWrongArgument(this,"
       first argument must be a variable.");
18     }
19
20     @Override
21     public Object execute(TransitionSystem ts, Unifier un,
       Term[] args) throws Exception {
22
23         checkArguments(args);
24         Circumstance c = ts.getC();
25         Long L = Math.round(c.getAS().get(0));
26         int i = Integer.valueOf(L.intValue());
27         NumberTermImpl result = new NumberTermImpl(i);
```

```
28          Term stateTerm = args[0];
29          return un.unifies(stateTerm, result);
30      }
31  }
```

Apéndice F

Código de la acción interna *readEmState* con cálculo acumulativo

```
1  // Internal action code for project ejemplo
2
3  package ia;
4
5  import jason.*;
6  import jason.asSemantics.*;
7  import jason.asSyntax.*;
8
9  public class readEmState extends DefaultInternalAction {
10     @Override public int getMinArgs() { return 1; }
11     @Override public int getMaxArgs() { return 1; }
12     //private int maxEmValue = 5;
13     //private int minEmValue = 1;
14     @Override protected void checkArguments(Term[] args)
       throws JasonException {
15         super.checkArguments(args); // check number of
       arguments
16         if ( !args[0].isVar())
17             throw JasonException.createWrongArgument(this,"
       first argument must be a variable.");
18     }
19
20     @Override
21     public Object execute(TransitionSystem ts, Unifier un,
       Term[] args) throws Exception {
22
23         checkArguments(args);
24         Circumstance c = ts.getC();
```

```
25          double positive = (c.getFulfilledExpectationsP().size
    () + c.getnotFulfilledExpectationsN().size());
26          double negative = (c.getnotFulfilledExpectationsP().
    size() + c.getFulfilledExpectationsN().size());
27          double emState =  positive - negative;
28          NumberTermImpl result;
29          if((positive - negative) == 0) result = new
    NumberTermImpl((minEmValue+maxEmValue)/2);
30          else {
31              double m = (maxEmValue - minEmValue) / (positive
    - (-negative));
32              double b = minEmValue - (-negative * m);
33              result = new NumberTermImpl((int)emState * m + b)
    ;
34          }
35          Term stateTerm = args[0];
36          return un.unifies(stateTerm, result);
37      }
38  }
```

Bibliografía

[1] Mehrabian A. "A general framework for describing and measuring individual differences in temperament". En: *Current Psychology* 14 (1996), págs. 261-292.

[2] Ortony A., Clore G. L. y Collins A. "The Cognitive Structure of Emotions". En: *Cambridge University Press, Cambridge, MA* (1988).

[3] Bexy Alfonso, Emilio Vivancos y Vicente Botti. "An Open Arquitecture for Affective Traits in a BDI Agent". En: *Proceedings of the 6th ECTA 2014. Part of the 6th IJCCI 2014* (2014), págs. 320-325.

[4] Bexy Alfonso, Emilio Vivancos y Vicente Botti. "From Affect Theoretical Foundations to Models of Intelligent Affective Agents". Awaiting publication.

[5] Bexy Alfonso, Emilio Vivancos y Vicente Botti. "Toward a Systematic Development of Affective Intelligent Agents". En: *Department of Informatic Systems and Computing, Universidad Politécnica de Valencia, Spain. http://hdl.handle.net/10251/62436* (2016).

[6] Bexy Alfonso, Emilio Vivancos y Vicente Botti. "Towards Formal Modeling of Affective Agents in a BDI Architecture". Awaiting publication. 2016.

[7] Albert Bandura. "Fearful Expectations and Avoidant Actions as Effects of Perceived Self-Inefficacy". En: *American Psychologist* (dic. de 1986), págs. 1389-1391.

[8] Christoph Bartneck. "Integrating the OCC Model of Emotions in Embodied Characters". En: *Workshop on Virtual Conversational Characters: Applications, Methods, and Research Challanges* (2002).

[9] J. Enrique Bigné y Luisa Andreu. "Modelo cognitivo-afectivo de la satisfacción en servicios de ocio y turismo". En: *Cuadernos de Economıa y Dirección de la Empresa* 21 (2004), págs. 89-120.

[10] Rafael H. Bordini, Jomi Fred Hübner y Michael Wooldridge. "Programming Multi-Agent Systems in AgentSpeak using Jason". En: *John Wiley & Sons* (2007).

[11] Michael Bratman. "Intention and Means-End Reasoning". En: *The Philosophical Review* 90.2 (1981), págs. 252-265.

[12] John Broome. "Practical reasoning". En: *Reason and Nature: Essays in the Theory of Rationality* (2002), págs. 85-111.

[13] John Broome y Christian Piller. "Normative Practical Reasoning". En: *Proceedings of the Aristotelian Society* 75 (2001).

[14] Becker-Asano C. "WASABI: Affect simulation for agents with believable interactivity". En: *IOS Press* 319 (2008).

[15] Cristiano Castelfranchi y Emiliano Lorini. "Cognitive Anatomy and Functions of Expectations". En: *Proceedings of IJCAI'03 Workshop on Cognitive Modeling of Agents and Multi-Agent iterations* (2003).

[16] Weiss G. "Multiagent systems: a modern approach to distributed artificial intelligence". En: *MIT press* (1999).

[17] Weiss G. "Multiagent systems: a modern approach to distributed artificial intelligence". En: *MIT Press, Cambridge, MA, USA* (1999).

[18] Patrick Gebhard. "ALMA – A Layered Model of Affect". En: *In: AAMAS '05: Proc. of the 4th Inter. J. Conf. on Autonomous Agents and MAS* (2005), págs. 29-36.

[19] Sarit A. Golub, Daniel T. Gilbert y Timothy D. Wilson. "Anticipating one's troubles: the costs and benefits of negative expectations". En: *Emotion* 9.2 (2009), págs. 277-281.

[20] *https://es.wikipedia.org/wiki/Siete_y_media.* (Visitado 14-04-2016).

[21] *https://javacc.java.net.* (Visitado 28-05-2016).

[22] *http://www.jessrules.com.* (Visitado 03-10-2016).

[23] Kristen J. Klaaren, Sara D. Hodges y Timothy Wilson. "The Role of Affective Expectations in Subjective Experience and Decision-Making". En: *Social Cognition* 12.2 (1994), págs. 77-101.

[24] Stacy C Marsella y Jonathan Gratch. "EMA: A process model of appraisal dynamics". En: *Cognitive Systems Research* 10 (2009), págs. 70-90.

[25] Morris W. N. Mood. "The frame of mind." En: *Springer Science & Business Media* (2012).

[26] Andrew Ortony, Gerald L. Clore y Allan Collins. "The Cognitive Structure of Emotions". En: *Cambridge University Press* (1990).

[27] Becker P. "Structural and Relational Analyses of Emotion and Personality Traits". En: *Zeitschrift für Differentielle und Diagnostische Psychologie* 22.3 (2001), págs. 155-172.

[28] Greenspan P. "Practical Reasoning and Emotion". En: *The Oxford hand-book of rationality* (2004), págs. 206-21.

[29] Surangika Ranathunga, Martin Purvis y Stephen Cranefield. "Integrating expectation handling into Jason". En: *The Information Science* 2011/02 (2011).

[30] Anand S. Rao y Michael P. George. "Modeling Rational Agents within a BDI-Architecture". En: *Second International Conference on Principles of Knowledge Representation and Reasoning* (1991).

[31] Anand S. Rao y Michael P. Georgeff. *BDI Agent: From Theory to Practice*. Proceedings of the First International Conference on Multi-Agent Systems ICMAS, San Francisco, 1995.

[32] Casey Reas y Ben Fry. *Processing: a programming handbook for visual designers and artists*. MIT Press, 2014.

[33] McCrae R.R. y John O.P. "An introduction to the five-factor model and its implications". En: *Journal of Personality* 60 (1992), págs. 171-215.

[34] Picard RW. "Affective computing". En: *MIT Press, Cambridge, MA, USA* (1997).

[35] Michael Schumacher. *Objective Coordination in Multi-Agent System Engineering*. Springer-Verlag, 2001.

[36] C. A. Smith y R. Lazarus. "Emotion and Adaptation". En: *Contemporary Sociology* 21.4 (ene. de 1990), págs. 609-637.

[37] Bastiaan Reinier Steunebrink. "The Logical Structure of Emotions". En: *SIKS Dissertation Series* 23 (2010).

[38] Robert E. Thayer. "The Origin of Everyday Moods: Managing Energy, Tension and Stress". En: *NY: Oxford University Press* (1996).

[39] Timothy D. Wilson y col. "Preferences as expectation-driven inferences: Effects of affective expectations on affective experience". En: *Journal of Personality and Social Psychology* 56.4 (1989), págs. 519-530.

[40] Piotr Winkielman, Robert B. Zajonc y Norbert Schwarz. "Subliminal Affective Priming Resists Attributional Interventions". En: *Cognition and Emotion* 11.4 (1997), págs. 433-465.

[41] Michael Wooldridge. "Intelligent Agents in Multiagent Systems." En: *Cambridge, Massachusetts, MIT Press* (1999), págs. 27-77.

[42] Michael Wooldridge y Nicholas Robert Jennings. “Intelligent Agents: Theory and Practice”. En: *The Knowledge Engineering Review* 10.2 (1995), págs. 115-152.