

10 | JASON

Jason [11, 9, 12, 10] es un intérprete programado en Java, para una versión extendida de *AgentSpeak(L)*. Como tal, implementa la semántica operacional del lenguaje y provee una plataforma para el desarrollo de SMA, con muchas opciones configurables por el usuario. Se trata de un código abierto, distribuido bajo una licencia GNU LGPL. Sus extensiones incluyen comunicación basada en actos de habla [107], herramientas para simulación social [14] y un sistema de módulos [80], entre otras.

En este capítulo introduciremos Jason y ejemplificaremos su uso. La organización del capítulo es como sigue: Primero abordaremos la instalación de Jason y sus ambientes de desarrollo: El estándar basado en jEdit; y el basado en Eclipse, gracias a un plug-in desarrollado por Zatelli. Posteriormente, abordamos Jason *à la* Prolog, es decir, haciendo uso de las creencias y las metas verificables que configuran un sistema muy similar a los sistemas de programación lógica. Se revisará el uso de hechos, reglas y metas verificables; la aritmética; las representaciones en primer orden; las anotaciones; y la negación fuerte y débil. A continuación, revisaremos el concepto de los módulos en Jason, puesto que ahora todo programa de agente está modularizado; y terminaremos el capítulo abordando la definición de acciones internas.

10.1 INSTALACIÓN

Jason puede descargarse gratuitamente desde su página en internet ¹, donde además encontrarán una descripción del lenguaje, documentación, ejemplos, proyectos asociados etc. También encontrarán una liga a la página del libro **Programando Sistemas Multi-Agentes en AgentSpeak(L) usando Jason** ², el mejor soporte para este lenguaje de programación orientado a agentes.

Libro

10.1.1 Distribución en sourceforge

La **distribución** de Jason se muestra en la Figura 10.1. Lo importante es que el ejecutable, en este caso `scripts/jason` esté en algún sitio accesible.

Distribución

Ejemplo 10.1. *El folder completo de la distribución, `jason-2`, puede colocarse en el folder de Aplicaciones en MacOS.*

Observen que el código fuente está disponible en `src` y que los ejemplos y demos del sitio web del lenguaje están incluidos en los folders `examples` y `demos`, respectivamente. La documentación en `doc` incluye algunos artículos relevantes y la descripción del API de Jason. La carpeta `scripts` tiene todos los scripts para ejecutar Jason desde consola. Como mencioné, es importante por lo tanto que esté en la ruta de acceso del sistema. Desde la versión 2.0 del lenguaje, no hay una aplicación disponible en la interfaz gráfica de los diferentes sistemas operativos donde Jason puede ejecutarse. En lugar de ello, deben usarse estos scripts. Como se menciona en el README es necesario tener instalado java 1.7 o superior (y de preferencia gradle).

¹ <http://jason.sourceforge.net/wp/>

² <http://jason.sourceforge.net/jBook/jBook/Home.html>

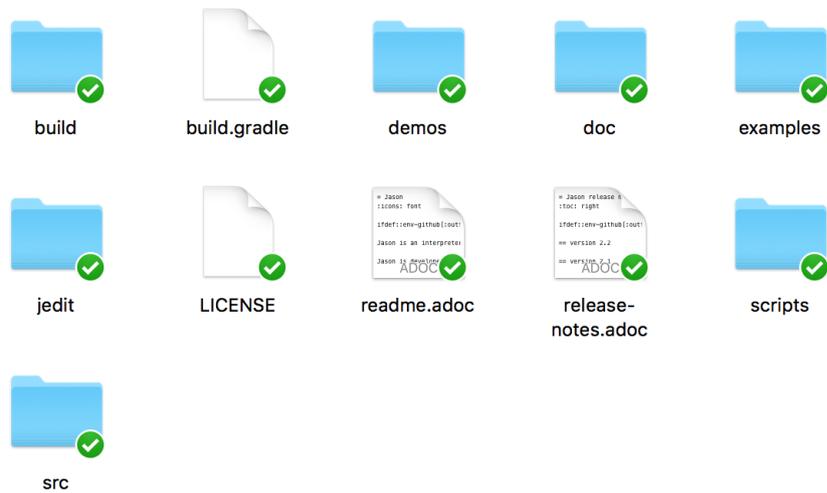


Figura 10.1: La carpeta principal de la distribución de Jason.

10.1.2 Distribución en github

Si bien las versiones del repositorio oficial en sourceforge se actualizan cuando se libera una versión mayor del lenguaje, es posible instalar los avances intermedios si basamos nuestra instalación usando el repositorio para desarrolladores disponible en github³ y compilándolo con ayuda de gradle:

```
1 > git clone https://github.com/jason-lang/jason.git
2 > cd jason
3 > gradle config
```

Otras tareas para gradle se describen en el Cuadro 10.1. Se deduce que config hace en realidad tres cosas: Compila las fuentes para generar los archivos jar correspondientes; configura el archivo de propiedades de jason y coloca todos los jar en la carpeta build/libs; y solicita al usuario su autorización para configurar las variables JASON_HOME y PATH adecuadamente⁴.

Acción	Descripción
jar	Genera un nuevo jason.jar
doc	Genera javadoc y transforma asciidoc en html.
eclipse	Genera una configuración para proyectos eclipse.
config	Ejecuta las tres acciones anteriores.
clean	Borra los archivos generados.
release	Produce un zip en build/distributions.

Cuadro 10.1: Acciones Gradle para instalar Jason.

10.2 AMBIENTES DE DESARROLLO

Jason cuenta con dos ambientes de desarrollo, el oficial basado en jEdit –Un editor de código multi lenguaje, implementado en java; y en eclipse, con todas las ventajas para java que esto conlleva.

³ <https://github.com/jason-lang/jason.git>

⁴ Efectivamente, es mejor usar Jason en un ambiente UNIX-like.

10.2.1 jEdit

Si abren una consola y ejecutan `jason-ide` tendrán acceso a la ventana principal de un ambiente de desarrollo basado en **jEdit** (Figura 10.2). Al estar implementado en java, este IDE no responde exactamente igual que las aplicaciones MacOS nativas, p. ej., su menú está en la ventana de la aplicación y no en la barra de menues tradicional.

jEdit

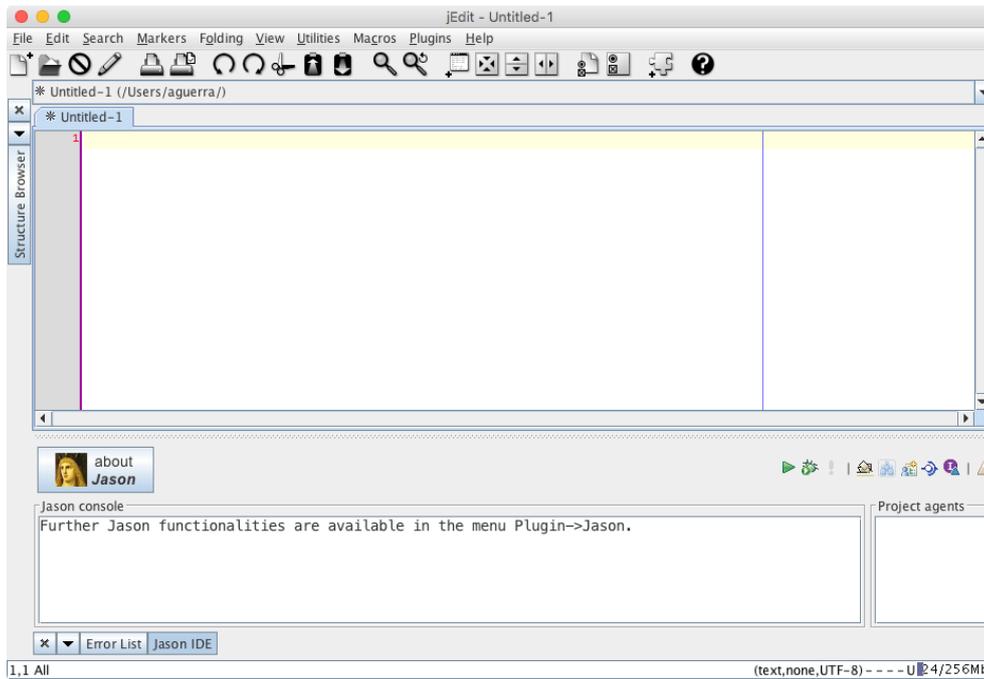


Figura 10.2: La ventana principal del IDE de Jason, basado en jEdit.

En este ambiente pueden definir y editar los componentes del sistema; ver el listado de agentes en el mismo; así como depurar y ejecutar el sistema definido. Es importante señalarle a Jason que versión de Java utilizaremos. Para ello el menú `Plugins:Plugins options` (Ver Figura 10.3) permite configurar éste y otros parámetros de Jason, como que versión de ant se usará para compilar ó las librerías que serán usadas con el sistema (`jade`, `cartago`, etc.). Observen que en mi caso, estoy usando la distribución `github` de `jason`; `java 9`; y la versión de `ant` que viene incluida en la distribución de `jason`, al igual que `jade`.

10.2.2 Eclipse

También es posible instalar un *plug-in*⁵ en eclipse⁶ para trabajar con Jason. Para ello, desde eclipse, vaya al menú `Ayuda/Instalar Nuevo Software`, lo cual abrirá la ventana que se muestra en la Figura 10.4. Dar clic en el botón de agregar **repositorio** (`Add...`), para proveer la siguiente información:

Plug-in Eclipse

Repositorio

- Nombre: `jasonide`
- Localidad: `http://jason.sourceforge.net/eclipseplugin/juno/`

Acepte la licencia del *plug-in* (la misma que Jason) y el ambiente estará instalado.

La organización de los proyectos en jEdit y eclipse **difieren** ligeramente. La diferencia más evidente es que, mientras que en jEdit las fuentes de nuestros programas están en el directorio raíz del proyecto, en Eclipse están en la carpeta `src`,

Diferencias

⁵ <http://jason.sourceforge.net/eclipseplugin/juno/>

⁶ <https://www.eclipse.org>

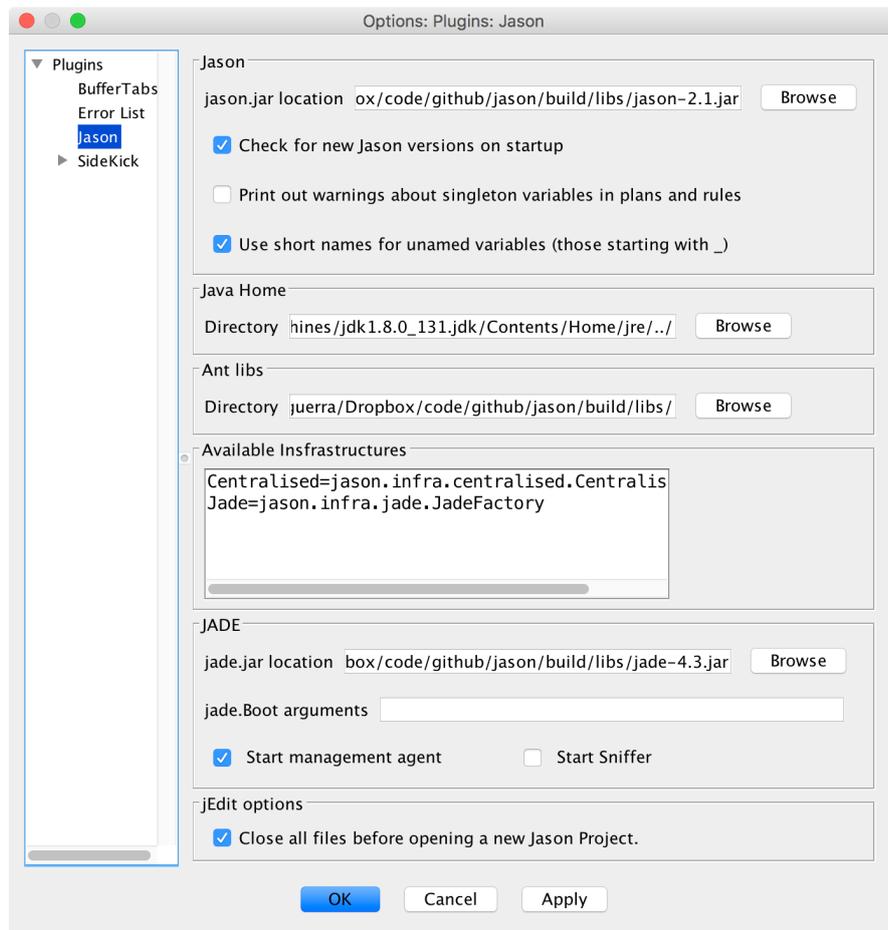


Figura 10.3: Configuración de Jason y Java en jEdit.

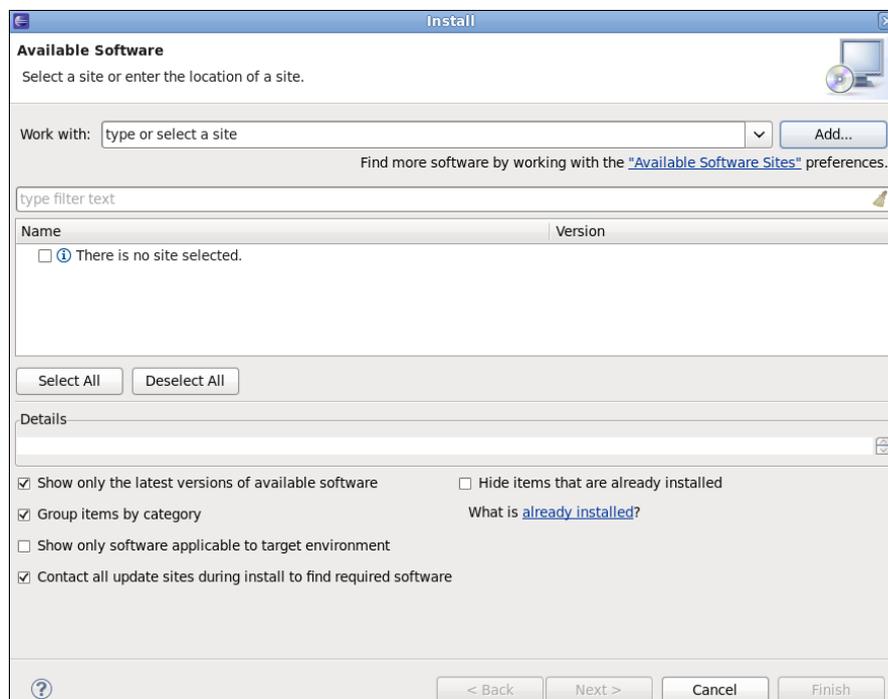


Figura 10.4: La ventana para instalar nuevo software en Eclipse.

que a su vez contiene dos carpetas java y asl. En este caso, es necesario especificar

el lugar donde están los programas de agente, mediante la siguiente instrucción en el archivo principal del proyecto (el que tiene la extensión mas2j): aslSrcPath: "src/asl".

10.3 DEFINIENDO UN SMA

En el ambiente de desarrollo basado en jEdit, el botón  abre una ventana para definir un **proyecto nuevo**. La Figura 10.5 muestra la entrada para crear un SMA, al que llamaremos saludos. La **infraestructura** centralizada define un SMA donde todos los agentes estarán ejecutándose en la misma computadora. Es posible distribuir a los agentes en una red usando como infraestructura Jade [6].

*Proyecto nuevo
Infraestructura*

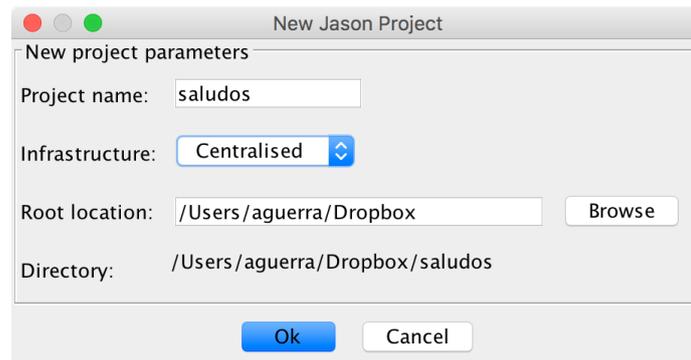


Figura 10.5: Definiendo un nuevo Sistema Multiagente llamado saludos.

El resultado de esta operación es un folder en la ruta indicada como directorio, p. ej. /Users/aguerra/saludos. Este directorio contendrá el archivo saludos.mas2j con la definición del SMA:

```

1  /* Jason Project */
2
3  MAS saludos {
4    infrastructure: Centralised
5    agents:
6  }
```

Observen que no hay agentes en este SMA. El botón  nos permite **añadir agentes**. Añadiremos dos agentes al sistema, enrique y beto. La Figura 10.6 muestra la ventana para la definición del agente enrique, se procede de igual manera para beto.

Añadir agentes

Aunque no usaremos las opciones disponibles de momento, observen que es posible definir una **clase** y una **arquitectura** para los agentes que añadimos al sistema. También es posible clonar un agente el número de veces que queramos. Como el sistema que definimos usa la infraestructura centralizada, nuestros agentes corren en el servidor local. También es posible variar la cantidad de información que un agente despliega al ser ejecutado, variando el nivel de verbose. Después de agregar a ambos agentes, nuestro archivo saludos.mas2j debería lucir así:

Clase y Arquitectura de agentes

```

1  /* Jason Project */
2
3  MAS saludos {
4    infrastructure: Centralised
5    agents:
6      enrique;
7      beto;
8  }
```

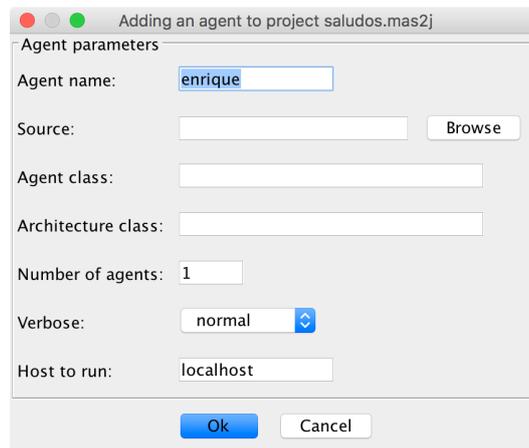


Figura 10.6: Añadiendo al agente enrique al SMA.

Además, en el folder del proyecto encontraremos los archivos fuente de `enrique.asl` y `beto.asl`. Primero, modificaremos al agente `enrique`, de forma que su archivo fuente sea como sigue:

```

1 // Agent enrique in project saludos.mas2j
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <- .send(beto,tell,hola).
```

Luego eliminaremos de `beto` todo lo que no sea comentarios. Observen que no hemos definido creencias para ninguno de los dos agentes.

El botón  lanza el **inspector de mentes**, que nos permite ejecutar el SMA paso a paso; y observar el estado mental de los agentes. Como es de esperar, dadas las definiciones previas, `enrique` saluda a `beto` y éste último registra en sus creencias el saludo por la performativa `tell` del mensaje en cuestión. Eventualmente el inspector de mentes debería mostrar esta información para `beto`, tal y como se muestra en la Figura 10.7.

Inspector de mentes

Podemos hacer que `beto` sea más receptivo, modificando su programa para reaccionar al mensaje de `enrique`.

```

1 // Agent beto in project saludos.mas2j
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 /* Plans */
8
9 +hola[source(Ag)] <- .print("Recibí un saludo de ", Ag).
```

Es más, podemos hacer de `beto` un agente educado:

```

1 // Agent beto in project saludos.mas2j
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 /* Plans */
8
```

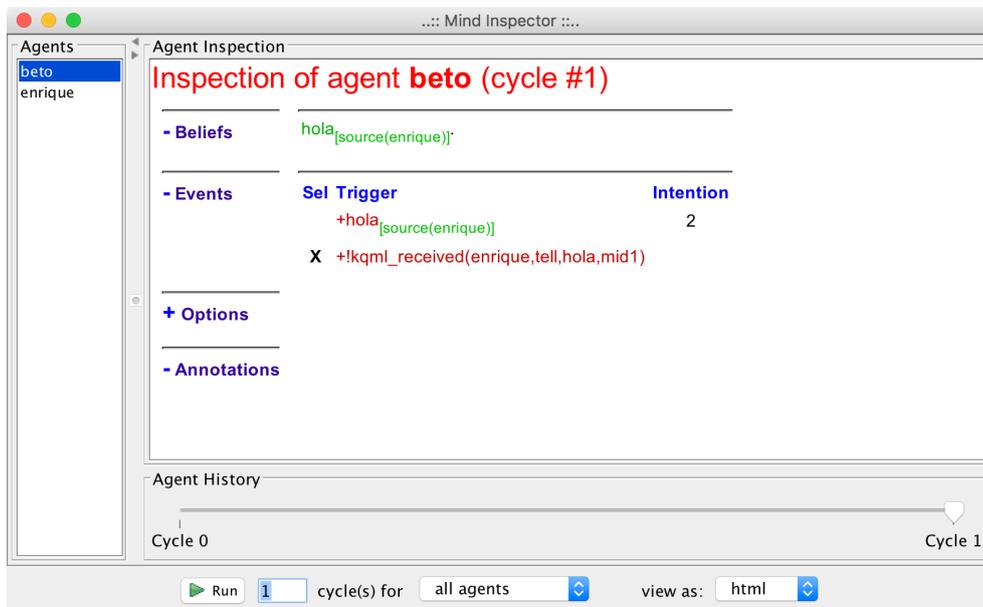


Figura 10.7: El estado mental del agente beto al recibir el mensaje de enrique

```

9 +hola[source(Ag)] <-
10 .print("Recibí un saludo de ", Ag);
11 .send(Ag,tell,hola).

```

Lo mismo para enrique:

```

1 // Agent enrique in project saludos.mas2j
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <- .send(beto,tell,hola).
12
13 +hola[source(Ag)] <-
14 .print("Recibí un saludo de ", Ag);
15 .send(Ag,tell,hola).

```

¿Cual será la salida en consola al ejecutar este SMA? Aunque parezca extraño, los agentes no entran en un ciclo de saludos, debido a que percepción y actualización de creencias no son equivalentes. De forma que la salida en consola al ejecutar este SMA será como se muestra en la Figura 10.8.

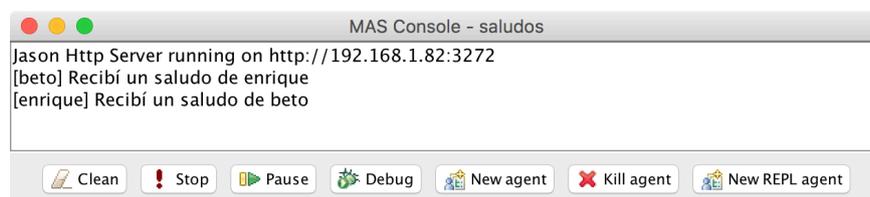


Figura 10.8: La consola de beto durante la ejecución del SMA.

La percepción, en este caso, está asociada a la recepción del mensaje con performativa `tell`; mientras que la actualización de creencias tiene que ver con qué hace cada agente con el contenido del mensaje. Como `hola` ya está en las creencias de ambos agentes luego del primer mensaje recibido, en los mensajes posteriores no se agrega

nada al estado mental de los agentes y, por tanto, el evento `+hola` nunca se produce, de ahí que no se impriman más mensajes en la consola.

Es posible utilizar Jade como infraestructura, modificando el archivo de definición del sistema `saludos.mas2j` de la siguiente manera:

```

1  /* Jason Project */
2
3  MAS saludos {
4    infrastructure: Jade
5    agents:
6    enrique;
7    beto;
8  }

```

esto permite ejecutar un agente sniffer (Ver Figura 10.9) para observar la comunicación entre `enrique` y `beto`; además de que eventualmente nos permitiría distribuir los agentes en una red de cómputo y otras facilidades. Para que el sniffer se ejecute automáticamente al lanzar el sistema, es necesario indicarlo en las preferencias del *plug-in* de Jason en el ambiente de desarrollo basado en `jEdit`.

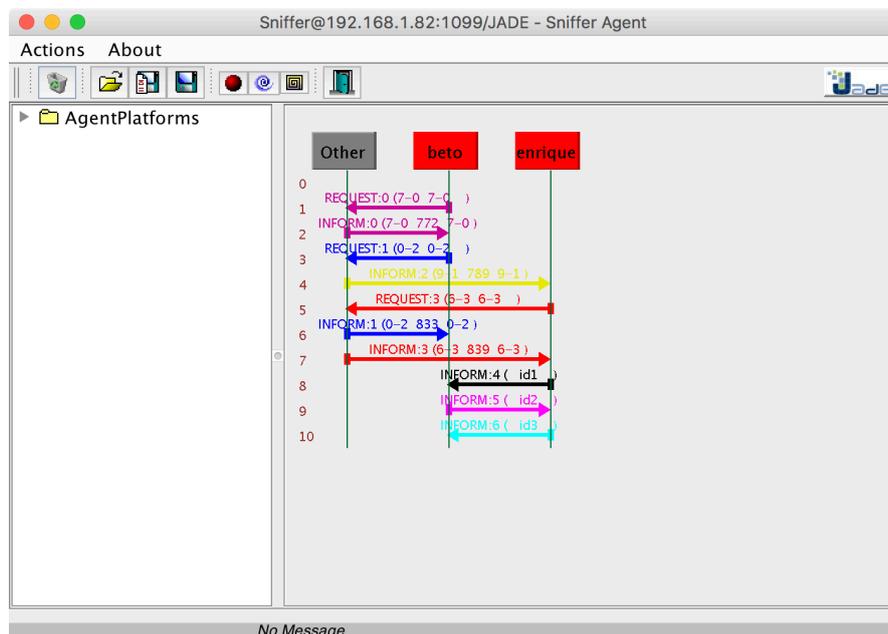


Figura 10.9: Un agente sniffer monitoreando comunicaciones.

10.4 IMPLEMENTACIÓN DE MEDIOS AMBIENTES

Los agentes están situados en su medio ambiente y los lenguajes de programación orientados a agentes deberían proveer una noción explícita de éste. Aunque esto no pareciera ser mandatorio en el caso de los agentes puramente comunicativos, como `enrique` y `beto`, observen que los actos de habla buscan ajustar el medio ambiente a los estados Intencionales del agente; y que las creencias son representaciones del agente ajustadas al medio ambiente.

En el caso de agentes situados en medios ambientes reales, aunque la simulación no es mandatoria, tiene algunas ventajas a saber: Los agentes y los SMA son sistemas distribuidos de un alto grado de complejidad. Aunque existen herramientas formales para la verificación de estos sistemas, la validación mediante simulación sigue siendo una práctica muy extendida.

En todo caso, simulado o real, el acceso de Jason al medio ambiente se define a través de Java. La **arquitectura general** de un agente incluye los métodos Java que

definen la interacción con el ambiente, como se muestra en diagrama de secuencia UML de la Figura 10.10.

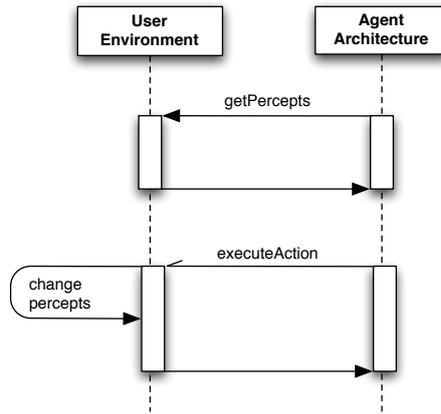


Figura 10.10: Interacción entre la implementación del ambiente y la arquitectura del agente.

La arquitectura de un agente utiliza el método `getPercepts` para obtener las percepciones del ambiente simulado. Estas pueden verse como propiedades del ambiente accesibles al agente, de forma que este método establece un mecanismo de **focus** de atención. A partir de esta información el agente **actualiza** sus creencias, normalmente cuando su ciclo de razonamiento está en el estado *ProcMsg*. Observen que la percepción y la actualización de creencias son dos procesos diferentes.

Focus de atención
Actualización de creencias

Ahora bien, cuando el agente ejecuta una acción, la arquitectura solicita al ambiente la ejecución de la acción y suspende la intención asociada hasta que el ambiente provee **retroalimentación** sobre la ejecución de la acción, normalmente, que la acción ha sido ejecutada. La verificación de si los efectos esperados de la acción se cumplieron o no, está asociada normalmente a la percepción y no a esta retroalimentación.

Retroalimentación

Observen que el ciclo del razonamiento del agente continua mientras la intención asociada a la acción ejecutada está **suspendida**. Esto tiene un efecto similar a si el método `executeAction` fuese invocado de forma asíncrona. Si el ambiente está siendo ejecutado en otra máquina, el lapso de esta suspensión puede ser considerable.

Intenciones suspendidas

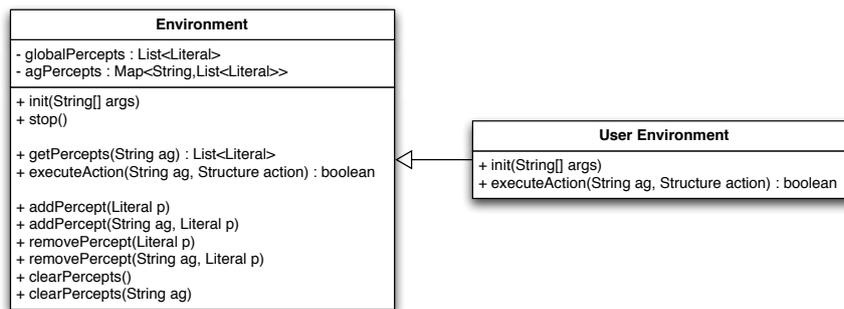


Figura 10.11: Implementación de un ambiente extendiendo la clase Environment.

Para implementar un ambiente en Jason, el programador normalmente extiende la clase `Environment` y redefine (usando *override*) los métodos `executeAction`, `init` y `stop`. La Figura 10.11 muestra un diagrama de clase mostrando esta relación. Una implementación de la clase ambiente extendida suele tener la estructura mostrada en el Cuadro 10.2.

Clase Environment

El Cuadro 10.3 resume los métodos de Java que pueden usarse para programar un ambiente Jason. Solo objetos de la clase `Literal`, que es parte del paquete `jason` pueden agregarse a las listas de percepciones mantenidas por la clase `Environment`.

Clase Literal

```

1 import jason.asSyntax.*;
2 import jason.environment.*;
3
4 public class <EnvironmentName> extends Environment {
5     // Los miembros de la clase...
6
7     @Override
8     public void init(String[] args) {
9         // Qué hacer al iniciar la ejecución...
10    }
11
12    @Override
13    public boolean executeAction(String ag, Structure act) {
14        // Efectos de las acciones...
15    }
16
17    @Override
18    public void stop() {
19        // Qué hacer al detener el sistema...
20    }
21 }

```

Cuadro 10.2: Implementación del ambiente del usuario.

En esta parte no debería considerarse agregar anotaciones a las literales, pues todas son anotadas automáticamente con `source(percept)`.

Método	Semántica
<code>addPercept(L)</code>	Agrega la literal <i>L</i> a la lista global de percepciones.
<code>addPercept(A,L)</code>	Agrega la literal <i>L</i> a las percepciones del agente <i>A</i> .
<code>removePercept(L)</code>	Remueve la literal <i>L</i> de la lista global de percepciones
<code>removePercept(A,L)</code>	Remueve la literal <i>L</i> de las percepciones del agente <i>A</i> .
<code>clearPercepts()</code>	Borra las percepciones de la lista global.
<code>clearPercepts(A)</code>	Borra las percepciones del agente <i>A</i> .

Cuadro 10.3: Métodos Java para programar ambientes Jason.

La mayor parte del código relacionado con la implementación de ambientes está en el método `executeAction`, que debe declararse tal y como se muestra en el Cuadro 10.2. Siempre que un agente trata de ejecutar una acción en el ambiente, el nombre del agente y una estructura representando la acción solicitada son enviadas a este método como parámetros. El código en `executeAction` suele verificar la estructura *à la Prolog* que representa la acción y el agente que intenta ejecutar la acción. Luego, para cada combinación acción/agente que sea relevante, el código hace lo necesario en el **modelo** del ambiente. Normalmente esto incluye cambiar ciertas percepciones. Observen que la ejecución de una acción es booleana y regresa falso si la solicitud de ejecución al ambiente falla. Un plan falla si alguna de sus acciones falla al ser ejecutada.

Recuerden que la percepción y la actualización de creencias no son procesos equivalentes. Esta posible confusión es causa de algunos errores al implementar ambientes y su interacción con los agentes mediante las clases y métodos definidos en Jason. Se suele esperar que los agentes mantengan en su estado mental las percepciones aún cuando estás solo estén presentes durante un ciclo de razonamiento. Esto es falso. Si un agente necesita recordar percepciones pasadas que ya no se dan en el ambiente, es necesario crear **notas mentales** al percibir la propiedad en cuestión a través de sus planes. Las notas mentales se recuerdan hasta que explícitamente son olvidadas. Las creencias asociadas a una percepción son eliminadas en cuanto la percepción se deja de observar en el ambiente. También es posible que

Ejecución de una acción

Notas mentales

una percepción desaparezca como efecto de la ejecución de una acción, antes de que el agente pueda formar una creencia acerca de ella. Aunque consideren que el proceso de actualización de creencias genera eventos asociados a agregar y borrar creencias.

Veamos todo esto con un ejemplo sencillo. Vamos a crear un nuevo SMA llamado *piros* y agregar un agente llamado *piro*. El código del agente será como sigue:

```

1 // Agent piro in project piros.mas2j
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <- incendiar.
12
13 +fuego <- .print("Fuego! Corran").

```

El agente *piro* es un buen piromano que incendia su ambiente y, eso sí, una vez que percibe fuego, avisa que hay que iniciar la huida. La creencia *fuego* deberá ser agregada por el ambiente, en respuesta a la acción externa *incendiar*.

El botón  nos permite definir un ambiente para el SMA que estamos definiendo. Al darle clic una ventana nos pedirá el nombre del ambiente. La única consideración aquí es que el ambiente es una clase de Java, y por tanto, su nombre debe iniciar con mayúscula. En nuestro caso el ambiente se llamará *PirosAmb*. Modifique el código del ambiente como sigue:

```

1 // Environment code for project piros.mas2j
2
3 import jason.asSyntax.*;
4 import jason.environment.*;
5 import java.util.logging.*;
6
7 public class PirosAmb extends Environment {
8
9     private Logger logger = Logger.getLogger("piros.mas2j."+PirosAmb.class.getName());
10
11     /** Se ejecuta al iniciar el SMA con la información en .mas2j */
12     @Override
13     public void init(String[] args) {
14         super.init(args);
15     }
16
17     @Override
18     public boolean executeAction(String agName, Structure action) {
19         if (action.getFunctor().equals("incendiar")) {
20             addPercept(Literal.parseLiteral("fuego"));
21             return true;
22         } else {
23             logger.info("executing: "+action+", but not implemented!");
24             return false;
25         }
26     }
27
28     /** Se ejecuta al cerrar el SMA */
29     @Override
30     public void stop() {
31         super.stop();
32     }
33 }

```

en este caso, la inicialización y cierre del ambiente se heredan de la clase *Environment*. Solo estamos redefiniendo el método *executeAction* para agregar la percepción *fuego* en respuesta a la acción *incendiar*. Observen el uso de *addPercept* para im-

plementar la percepción del fuego. Al ejecutar el SMA, la salida en consola es la siguiente:

```
1 | [piro] Fuego! Corran
```

10.5 JASON A LA PROLOG

De cierta forma, las creencias de Jason y las metas verificables se comportan de manera muy similar a un sistema de **Programación Lógica**, digamos Prolog [102, 17, 25]. Para ilustrar esto vamos a crear un nuevo proyecto en eclipse, llamado creencias, con una infraestructura centralizada y sin un medio ambiente asociado. Si todo va bien, el navegador de Jason, debe mostrar el proyecto que incluye al agente `sample_agent`.

Programación Lógica

10.5.1 Hechos, reglas y metas verificables

Modifiquemos este agente para incluir creencias sobre su familia (clásico ejemplo de Prolog):

```
1 | // Agent agent1 in project creencias
2 |
3 | /* Initial beliefs and rules */
4 |
5 | progenitor(carmelo,alejandro).
6 | progenitor(carmen,alejandro).
7 | progenitor(carmelo,laura).
8 | progenitor(carmen,laura).
9 | progenitor(laura,rafael).
10 | progenitor(isidro,rafael).
11 |
12 | /* Initial goals */
13 |
14 | !start.
15 |
16 | /* Plans */
17 |
18 | +!start <-
19 |   ?progenitor(laura,rafael);
20 |   .print("Laura es progenitor de Rafael");
21 |   ?progenitor(carmelo,Y);
22 |   .print("Caramelo es progenitor de ", Y, ".");
23 |   ?progenitor(X,rafael);
24 |   .print(X," es un progenitor de Rafael").
```

A diferencia de Prolog, es el agente y no el usuario quien hace las preguntas en este caso. Para ello se define el plan de este agente. Su primera acción verifica si un hecho es verdadero (que Laura es progenitor de Rafael); y luego se hacen dos preguntas más para saber de quién es progenitor Carmelo y quién es progenitor de Rafael. La acción interna `.print`, imprime mensajes en consola. En este caso, la salida del programa sería la siguiente:

Preguntas

```
1 | [agent1] Laura es progenitor de Rafael
2 | [agent1] Caramelo es progenitor de alejandro.
3 | [agent1] laura es un progenitor de rafael
```

Si una pregunta **falla**, el plan falla y la intención asociada también. Por ejemplo, si agregamos la meta verificable `?madre(laura,rafael)` al final del plan del agente, tendremos un fallo, ya que tal meta no puede ser resuelta:

Fallo

```
1 | [agent1] Laura es progenitor de Rafael
2 | [agent1] Caramelo es progenitor de alejandro.
3 | [agent1] laura es un progenitor de rafael
4 | [agent1] No failure event was generated for +!start[code(madre(laura,
```

```

5 | rafael)),code_line(25),code_src("../creencias/src/asl/sample_agent.asl"),
6 | error(test_goal_failed), error_msg("Failed to test '?madre(laura,rafael)')"),
7 | source(self)]

```

Observen el uso de las etiquetas para registrar el fallo: La meta `!start` falló, debido a que una meta verificable `?madre(laura,rafael)` ha fallado. Hay varios términos en las etiquetas del evento de fallo que nos puede ser de utilidad en estos casos:

- `code(C)` nos indica que *C* fue el elemento del programa que causó el fallo.
- `code_src(Asl)` nos indica que *Asl* es el programa de agente que falló.
- `code_line(L)` nos indica que el error se produjo en la línea *L*.
- `error(X)` nos indica que el error *X* se produjo.
- `error_msg(Msg)` nos indique que *Msg* es el mensaje que será desplegado en consola para señalar el error.+

Esta información puede ser usada para agregando los planes relevantes (`-!α`), para **contender con el error**:

Planes de fallo

```

1 | // Agent agent2 in project creencias
2 |
3 | /* Initial beliefs and rules */
4 |
5 | progenitor(carmelo,alejandro).
6 | progenitor(carmen,alejandro).
7 | progenitor(carmelo,laura).
8 | progenitor(carmen,laura).
9 | progenitor(laura,rafael).
10 | progenitor(isidro,laura).
11 |
12 | /* Initial goals */
13 |
14 | !start.
15 |
16 | /* Plans */
17 |
18 | +!start <-
19 |   ?progenitor(laura,rafael);
20 |   .print("Laura es progenitor de Rafael");
21 |   ?progenitor(carmelo,Y);
22 |   .print("Carmelo es progenitor de ", Y, ".");
23 |   ?progenitor(X,rafael);
24 |   .print(X," es un progenitor de Rafael");
25 |   ?madre(laura,rafael).
26 |
27 | -!start[error(Error)] <-
28 |   .print("El plan +!start falló por el error ", Error).

```

con lo que el error es procesado adecuadamente:

```

1 | [agent1] Laura es progenitor de Rafael
2 | [agent1] Carmelo es progenitor de alejandro.
3 | [agent1] laura es un progenitor de rafael
4 | [agent1] El plan +!start falló por el error test_goal_failed

```

En realidad, en este caso querríamos agregar conocimiento al agente para contender con la meta problemática, en lugar de procesar el fallo una vez que éste ha sucedido. Agregar conocimiento, significa agregar creencias al agente, incluyendo reglas:

```

1 | // Agent agent3 in project creencias
2 |
3 | /* Initial beliefs and rules */
4 |
5 | progenitor(carmelo,alejandro).

```

```

6 | progenitor(carmen,alejandro).
7 | progenitor(carmelo,laura).
8 | progenitor(carmen,laura).
9 | progenitor(laura,rafael).
10| progenitor(isidro,laura).
11|
12| mujer(laura).
13| mujer(carmen).
14| hombre(carmelo).
15| hombre(alejandro).
16| hombre(isidro).
17|
18| madre(X,Y) :- mujer(X) & progenitor(X,Y).
19| padre(X,Y) :- hombre(X) & progenitor(X,Y).
20|
21| /* Initial goals */
22|
23| !start.
24|
25| /* Plans */
26|
27| +!start <-
28|   ?progenitor(laura,rafael);
29|   .print("Laura es progenitor de Rafael");
30|   ?progenitor(carmelo,Y);
31|   .print("Carmelo es progenitor de ", Y, ".");
32|   ?progenitor(X,rafael);
33|   .print(X, " es un progenitor de Rafael");
34|   ?madre(laura,rafael);
35|   .print("Laura es madre de Rafael");
36|   ?madre(Z,alejandro);
37|   .print(Z, " es madre de Alejandro").
38|
39| -!start[error(Error)] <-
40|   .print("El plan +!start falló por el error ", Error).

```

La salida es la siguiente:

```

1 | [agente3] Laura es progenitor de Rafael
2 | [agente3] Carmelo es progenitor de alejandro.
3 | [agente3] laura es un progenitor de Rafael
4 | [agente3] Laura es madre de Rafael
5 | [agente3] carmen es madre de Alejandro

```

Por supuesto que las reglas pueden ser recursivas, por ejemplo:

```

1 | // Agent agent4 in project creencias
2 |
3 | /* Initial beliefs and rules */
4 |
5 | progenitor(carmelo,alejandro).
6 | progenitor(carmen,alejandro).
7 | progenitor(carmelo,laura).
8 | progenitor(carmen,laura).
9 | progenitor(laura,rafael).
10| progenitor(isidro,rafael).
11|
12| mujer(laura).
13| mujer(carmen).
14| hombre(carmelo).
15| hombre(alejandro).
16| hombre(isidro).
17|
18| madre(X,Y) :- mujer(X) & progenitor(X,Y).
19| padre(X,Y) :- hombre(X) & progenitor(X,Y).
20|
21| ancestro(X,Y) :- progenitor(X,Y).
22| ancestro(X,Y) :- progenitor(X,Z) & progenitor(Z,Y).
23|
24| /* Initial goals */
25|
26| !start.

```

```

27
28 /* Plans */
29
30 +!start <-
31   ?ancestro(carmelo,rafael);
32   .print("Carmelo es un ancestro de Rafael");
33   ?ancestro(X,rafael);
34   .print(X, " es un ancestro de Rafael");
35   .findall(Xs, ancestro(Xs,rafael),L);
36   .print("Los ancestros de Rafael son ",L).

```

Con la siguiente salida:

```

1 [agente4] Carmelo es un ancestro de Rafael
2 [agente4] laura es un ancestro de Rafael
3 [agente4] Los ancestros de Rafael son [laura,isidro,carmelo,carmen]

```

Observen la acción interna `.findall`, que se usa al igual que en Prolog, para coleccionar todas las respuestas posibles a una meta dada. La acción interna `.setof` hace lo mismo, pero sin incluir soluciones repetidas, construyendo el conjunto solución de manera incremental. El primer argumento de estas acciones es un patrón que representa la forma en que los resultados serán recolectados. En este caso, como solo coleccionamos la variable `X`, solo los nombres de quien sea ancestro serán coleccionados. Si sustituimos `X` por `ancestro(X)`, obtendríamos una lista de estos.

```

1 [agente4] Los ancestros de Rafael son [ancestro(laura),ancestro(isidro),
2 ancestro(carmelo),ancestro(carmen)]

```

El segundo argumento de estas acciones es la meta a resolver. Su tercer argumento es una **lista**, donde los resultados son recolectados.

Listas

10.5.2 Listas

Las listas se representan igual que en Prolog. La lista **vacía** puede denotarse por `[]` y la lista que tiene una **cabeza** `X` y una **cola** `[Xs]` de denota como `[X|Xs]`. Veamos un ejemplo de búsqueda en una lista.

Lista vacía
Cabeza
Cola

```

1 // Agent agente5 in project creencias
2
3 /* Initial beliefs and rules */
4
5 busqueda(X,[X|_]).
6 busqueda(X,[Y|Ys]) :- busqueda(X,Ys).
7
8
9 /* Initial goals */
10
11 !start.
12
13 /* Plans */
14
15 +!start : true <-
16   Lista = [1,2,3,4,5];
17   ?busqueda(3,Lista);
18   .print("3 es miembro de la lista ",Lista);
19   .findall(X,busqueda(X,Lista),L);
20   .print("Los miembros de la Lista son ",L).

```

Cuya salida en consola es:

```

1 [agente5] 3 es miembro de la lista [1,2,3,4,5]
2 [agente5] Los miembros de la Lista son [1,2,3,4,5]

```

Agreguemos la regla *elimina/3* que elimina un elemento de una lista y regresa la lista modificada:

```

1 // Agent agente6 in project creencias
2

```

```

3  /* Initial beliefs and rules */
4
5  busqueda(X,[X|_]).
6  busqueda(X,[Y|Ys]) :- busqueda(X,Ys).
7
8  elimina(X,[X|Xs],Xs).
9  elimina(X,[Y|Ys],[Y|Zs]) :- elimina(X,Ys,Zs).
10
11 /* Initial goals */
12
13 !start.
14
15 /* Plans */
16
17 +!start : true <-
18   Lista = [1,2,3,4,5];
19   .print("La lista original es ",Lista);
20   ?elimina(3,Lista,Resultado);
21   .print("Eliminar 3 de la lista resulta en ",Resultado).

```

Cuya salida es:

```

1  [agente6] La lista original es [1,2,3,4,5]
2  [agente6] Eliminar 3 de la lista resulta en [1,2,4,5]

```

Muchas de estos predicados se pueden implementar como **acciones internas** predefinidas y definidas por el usuario en Java. El cuadro 10.4 resume las acciones predefinidas para el manejo de listas y conjuntos.

Acciones internas

Acción interna	Descripción
<i>.member(X, Xs)</i>	X es miembro de Xs.
<i>.length(X, L)</i>	La longitud de X es L.
<i>.empty(X)</i>	X es una lista vacía.
<i>.concat(L₁, ..., L_n)</i>	Concatena todas las listas en L _n .
<i>.delete(X, L, R)</i>	Elimina X de L resultando la lista R.
<i>.reverse(L, R)</i>	La lista R es el reverso de L.
<i>.shuffle(L, R)</i>	R es la lista L revuelta.
<i>.nth(N, L, R)</i>	R es en N-ésimo elemento de la lista L.
<i>.max(L, R)</i>	R es el máximo elemento de la lista L.
<i>.min(L, R)</i>	R es el mínimo elemento de la lista L.
<i>.sort(L, R)</i>	R es la lista resultante de ordenar L.
<i>.list(L)</i>	Verifica si L es una lista.
<i>.suffix(R, L)</i>	R es un sufijo de la lista L.
<i>.prefix(R, L)</i>	R es un prefijo de la lista L.
<i>.sublist(R, L)</i>	R es una sub-lista de la lista L.
<i>.difference(L₁, L₂, R)</i>	R es la diferencia entre L ₁ y L ₂ .
<i>.intersection(L₁, L₂, R)</i>	R es la intersección de L ₁ y L ₂ .
<i>.union(L₁, L₂, R)</i>	R es la unión de L ₁ y L ₂ .

Cuadro 10.4: Acciones internas predefinidas para listas y conjuntos.

El siguiente agente prueba muchas de las acciones para listas:

```

1  // Agent agente7 in project creencias
2
3  /* Initial beliefs and rules */
4
5  /* Initial goals */
6

```

```

7 | !start.
8 |
9 | /* Plans */
10 |
11 | +!start : true <-
12 |   Lista1 = [1,2,3,4,5];
13 |   Lista2 = [a,b,c,d,e];
14 |   .print("La lista 1 es ",Lista1);
15 |   .print("La lista 2 es ",Lista2);
16 |   .member(X,Lista1);
17 |   .print(X, " es miembro de la lista 1");
18 |   .length(Lista1,Long);
19 |   .print("La longitud de la lista 1 es ",Long);
20 |   .concat(Lista1,Lista2,L3);
21 |   .print("Pegar la lista 1 y 2 nos da ",L3);
22 |   .delete(X,Lista1,L4);
23 |   .print("Borrar ",X," de la lista 1, nos da ",L4," Oops!");
24 |   .delete(c,Lista2,L5);
25 |   .print("Borrar c de la lista 2 no es problema ",L5);
26 |   .shuffle(Lista1,L6);
27 |   .print("Revolver la lista 1 produce ",L6);
28 |   .reverse(Lista2,L7);
29 |   .print("Invertir la lista 2 ",L7);
30 |   .nth(Long-1,Lista1,Last);
31 |   .print("El último elemento de la lista 1 es ",Last);
32 |   .max(Lista1,MaxL1);
33 |   .print("El máximo elemento en la lista 1 es ",MaxL1);
34 |   .min(Lista2,MinL2);
35 |   .print("El mínimo elemento de la lista 2 es ",MinL2);
36 |   .sort(L6,L8);
37 |   .print("Ordenar la lista 1 revuelta resulta en ",L8).

```

Cuya salida se muestra a continuación:

```

1 | [agente7] La lista 1 es [1,2,3,4,5]
2 | [agente7] La lista 2 es [a,b,c,d,e]
3 | [agente7] 1 es miembro de la lista 1
4 | [agente7] La longitud de la lista 1 es 5
5 | [agente7] Pegar la lista 1 y 2 nos da [1,2,3,4,5,a,b,c,d,e]
6 | [agente7] Borrar 1 de la lista 1, nos da [1,3,4,5] Oops!
7 | [agente7] Borrar c de la lista 2 no es problema [a,b,d,e]
8 | [agente7] Revolver la lista 1 produce [3,5,4,1,2]
9 | [agente7] Invertir la lista 2 [e,d,c,b,a]
10 | [agente7] El último elemento de la lista 1 es 5
11 | [agente7] El máximo elemento en la lista 1 es 5
12 | [agente7] El mínimo elemento de la lista 2 es a
13 | [agente7] Ordenar la lista 1 revuelta resulta en [1,2,3,4,5]

```

Algunas **observaciones** son pertinentes. Como su nombre lo indica, estos constructores no son creencias del agente, como si lo son las reglas y los hechos ejemplificados anteriormente. Las acciones internas son operaciones implementadas en Java, que no afectan el medio ambiente del agente. En principio, deberían ser más **eficientes** que sus contrapartes implementadas a la Prolog, pero como veremos luego, no son explotables al usar **actos de habla**.

Además, al no ser cláusulas, la **semántica** de estas operaciones no se sigue de la Programación Lógica, sino de su implementación en Java (La cual está muy bien documentada en el caso de las acciones internas predefinidas por Jason). Consideren *.delete* como ejemplo: El primer argumento de esta operación puede ser un término, una cadena de texto, o un número; y su comportamiento depende del tipo de argumento recibido de forma poco afortunada: Si queremos borrar las ocurrencias de 1 en una lista de números, esta acción no nos sirve, pues en realidad borrará el segundo elemento de la lista al ser su primer argumento un número (Ver salida anterior). El siguiente agente define una cláusula *del* que borra todas las ocurrencias de un término, número o no, en una lista.

```

1 | // Agent agente8 in project creencias
2 |
3 | /* Initial beliefs and rules */

```

Cláusulas vs
Acciones Internas

Eficiencia

Comunicación

Semántica

```

4 |
5 | del(_,[],[]).
6 | del(X,[X|L1],L2) :- del(X,L1,L2).
7 | del(X,[H|L1],[H|L2]) :- X\==H & del(X,L1,L2).
8 |
9 | /* Initial goals */
10 |
11 | !start.
12 |
13 | /* Plans */
14 |
15 | +!start : true <-
16 |   Lista = [1,2,3,2,4,2,5];
17 |   ?del(2,Lista,R);
18 |   .print("Eliminar 2 de la lista ",Lista," resulta en ",R).

```

Su salida en consola es:

```
1 | [agente8] Eliminar 2 de la lista [1,2,3,2,4,2,5] resulta en [1,3,4,5]
```

10.5.3 Artimética

Jason provee una serie de operadores aritméticos predefinidos (Ver cuadro 10.5) como acciones internas. Al igual que con las acciones para listas, algunos de ellos se pueden reprogramar à la Prolog, si su semántica no es la esperada.

<i>math.abs(N)</i>	<i>math.acos(N)</i>	<i>math.asin(N)</i>	<i>math.atan(N)</i>
<i>math.average(L)</i>	<i>math.cell(N)</i>	<i>math.cos(N)</i>	<i>.count(B)</i>
<i>math.e</i>	<i>math.floor(N)</i>	<i>.length(L)</i>	<i>math.log(N)</i>
<i>math.max(N₁, N₂)</i>	<i>math.min(N₁, N₂)</i>	<i>math.pi</i>	<i>math.random(N)</i>
<i>math.round(N)</i>	<i>math.sin(N)</i>	<i>math.sqrt(N)</i>	<i>math.std_dev(L)</i>
<i>math.sum(L)</i>	<i>math.tan(N)</i>	<i>system.time</i>	

Cuadro 10.5: Acciones internas aritméticas.

El siguiente agente hace uso de algunas funciones aritméticas:

```

1 | // Agent agente9 in project creencias
2 |
3 | /* Initial beliefs and rules */
4 |
5 | /* Initial goals */
6 |
7 | !start.
8 |
9 | /* Plans */
10 |
11 | +!start : true <-
12 |   Lista1 = [1,2,3,4,5];
13 |   .print("La lista 1 es ",Lista1);
14 |   .print("La longitud de la lista 1 ", .length(Lista1));
15 |   .print("La sumatoria de la lista 1 es ", math.sum(Lista1));
16 |   .print("El promedio de la lista 1 es ", math.average(Lista1)).

```

Su salida en consola es:

```

1 | [agente9] La lista 1 es [1,2,3,4,5]
2 | [agente9] La longitud de la lista 1 es 5
3 | [agente9] La sumatoria de la lista 1 es 15
4 | [agente9] El promedio de la lista 1 es 3

```

10.5.4 Otras estructuras

El siguiente agente incluye una serie de cláusulas para trabajar con árboles binarios.

```

1 // Agent agente10 in project creencias
2
3 /* Initial beliefs and rules */
4
5 insertaArbol(X,vacio,arbol(X,vacio,vacio)).
6
7 insertaArbol(X,arbol(X,A1,A2),arbol(X,A1,A2)).
8
9 insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1N,A2)) :-
10   X<Y & insertaArbol(X,A1,A1N).
11
12 insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1,A2N)) :-
13   X>Y & insertaArbol(X,A2,A2N).
14
15 creaArbol([],A,A).
16
17 creaArbol([X|Xs],AAux,A) :-
18   insertaArbol(X,AAux,A2) &
19   creaArbol(Xs,A2,A).
20
21 lista2arbol(Xs,A) :- creaArbol(Xs,vacio,A).
22
23 nodos(vacio,[]).
24
25 nodos(arbol(X,A1,A2),Xs) :-
26   nodos(A1,Xs1) &
27   nodos(A2,Xs2) &
28   .concat(Xs1,[X|Xs2],Xs).
29
30 ordenaLista(L1,L2) :-
31   lista2arbol(L1,A) &
32   nodos(A,L2).
33
34 /* Initial goals */
35
36 !start.
37
38 /* Initial plans */
39
40 +!start <-
41   Lista1 = [5,3,4,1,2];
42   ?lista2arbol(Lista1,Arbol1);
43   .print("La lista 1 es ", Lista1);
44   .print("El árbol creado de la lista es ",Arbol1);
45   ?nodos(Arbol1,Nodos1);
46   .print("Cuyos nodos en orden son ",Nodos1).

```

Su salida en consola es la siguiente:

```

1 [agente10] La lista 1 es [5,3,4,1,2]
2 [agente10] El árbol creado de la lista es arbol(5,arbol(3,arbol(1,
3 vacio,arbol(2,vacio,vacio)),arbol(4,vacio,vacio)),vacio)
4 [agente10] Cuyos nodos en orden son [1,2,3,4,5]

```

10.5.5 Anotaciones

Todas las creencias de Jason tienen al menos una **anotación** asociada, su fuente. En el inspector de mentes del sistema (Ver figura 10.12), podrán ver que todas las creencias usadas hasta ahora están adornadas con la etiqueta `source(self)`, que significa que se trata de creencias añadidas por el mismo agente. De hecho, como los agentes comunican y perciben creencias, etiquetar la fuente de éstas era una necesidad. Posteriormente, se generalizó el uso de las etiquetas para incluir meta-información asociada a las literales de Jason.

Anotación

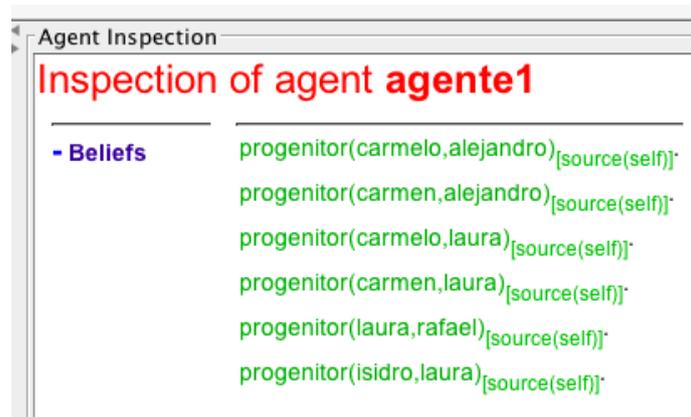


Figura 10.12: Las creencias de un agente siempre están anotadas, al menos con `source(self)`.

Las anotaciones no cambian el poder expresivo del lenguaje de programación, pero mejoran su legibilidad. Su sintaxis es la de una lista de términos (cuidado, no de literales). Por ejemplo:

```
1 | p(t)[source(self),costo(10),prioritario]
```

puede representar que la literal $p(t)$ ha sido agregada a las creencias por el agente mismo, tiene un costo de 10 unidades y se trata de algo prioritario. Observen que todo ello es meta-información sobre la creencia.

Aunque la sintaxis de las anotaciones se corresponde con la de una lista de términos, en realidad su semántica es la de un conjunto y así es como son consideradas por Jason.

Unificación con anotaciones

El uso de las anotaciones introduce una restricción al computar el unificador más general entre dos **literales**. L_1 unifica con L_2 si y sólo si las anotaciones de L_1 son un subconjunto de las de L_2 . Por ejemplo

*Unificador más general
Unificación entre literales*

```
1 | p(t) = p(t)[a1];           // Unifica
2 | p(t)[a1] = p(t);         // No unifica
3 | p(t)[a2] = p(t)[a1,a2,a3] // Unifica
```

Como las anotaciones son listas que representan conjuntos, la notación de **acceso a listas** para cabeza y cola pueden usarse con ellas:

Acceso a listas

```
1 | p(t)[a2|As] = p(t)[a1,a2,a3] // As unifica con [a1,a3]
2 | p(t)[a1,a2,a3] = p(t)[a1,a4|As] // As unifica con [a2,a3]
```

La unificación con **variables** es un poco más complicada, ya que debe considerar los diversos casos de unificación para $X[As] = Y[Bs]$; y si las variables en cuestión son de base o no. Cuando X e Y son de base:

Unificación con variables

```
1 | X = p[Cs] // unifica X con p[Cs]
2 | Y = p[Ds] // unifica Y con p[Ds]
3 | X[As] = Y[Bs] // unifica si (Cs ∪ As) ⊂ (Ds ∪ Bs)
```

El siguiente ejemplo ilustra este caso:

```
1 | X = p[a1,a2];
2 | Y = p[a1,a3];
3 | X[a4] = Y[a2,a4,a5]; // unifica
```

donde $Cs = \{a_1, a_2\}$, $Ds = \{a_1, a_3\}$, $As = \{a_4\}$ y $Bs = \{a_2, a_4, a_5\}$. De lo que se sigue que $Cs \cup As = \{a_1, a_2, a_4\}$ y $Ds \cup Bs = \{a_1, a_2, a_3, a_4, a_5\}$ y por tanto, se satisface la restricción definida previamente.

Cuando solo X es de base, la unificación se resuelve de la siguiente forma:

```

1 | X      = p[Cs]
2 | X[As] = Y[Bs] // unifica si (Cs ∪ As) ⊂ Bs
3 |           // e Y unifica con p

```

Cuando solo Y es de base, la unificación se resuelve de la siguiente forma:

```

1 | Y      = p[Ds]
2 | X[As] = Y[Bs] // unifica si As ⊂ (Ds ∪ Bs)
3 |           // y X unifica con p

```

10.5.6 Negación fuerte y débil

A diferencia de Prolog, donde el **principio del mundo cerrado** (*Closed World Assumption*, *CWA*) se adopta automáticamente, Jason puede contender también con una representación fuerte de la **negación**. Recuerden que el *CWA* expresa que todo lo que no se sabe cierto, o no es derivable de lo que se sabe cierto siguiendo las reglas del programa, es falso. En este sentido, Jason provee el operador `not`, donde la negación de una fórmula es cierta, si el intérprete falla al derivar dicha fórmula.

El operador de **negación fuerte** es utilizado para representar que el agente explícitamente cree que cierta fórmula no es el caso. La semántica de las negaciones, cuando se aplican a literales, se muestra en el cuadro 10.6.

CWA

Negación

Negación fuerte

Sintaxis	Semántica
l	El agente cree que l es verdadera
$\sim l$	El agente cree que l es falsa
$not\ l$	El agente no cree que l es verdadera
$not\ \sim l$	El agente no cree que l es falsa.

Cuadro 10.6: Semántica de la negación de literales.

El siguiente ejemplo ilustra todo lo que conocemos de las creencias. El agente cree que la *caja1* es *roja*, pero según *beto* la *caja1* es verde. Para complicar más la historia, según *enrique* la *caja1* no es verde. La meta principal del agente es reportar de que color es la caja.

```

1 | // Agent agente11 in project creencias
2 |
3 | /* Initial beliefs and rules */
4 |
5 | color(caja1,verde)[source(beto)].
6 | ~color(caja1,verde)[source(enrique)]. // azul no causa contradicción
7 | color(caja1,rojo). // verde hace que enrique sea el mentiroso
8 |
9 | colorSegunYo(Caja,Color) :-
10 |   color(Caja,Color)[source(Src)] &
11 |   (Src == self | Src == percept).
12 |
13 | descr(Ag,mentiroso) :-
14 |   mentiroso(Ag)[cert(C1)] &
15 |   daltonico(Ag)[cert(C2)] &
16 |   C1 > C2.
17 | descr(Ag,daltonico) :- daltonico(Ag).
18 | descr(Ag,confiable).
19 |
20 | /* Initial goals */
21 |
22 | !start.
23 |
24 | /* Plans */
25 |
26 | @contradiccion
27 | +!start : color(caja1,Color) & ~color(caja1,Color)[source(S2)] <-

```

```

28 | .print("Contradicción detectada");
29 | ?color(cajal,Color1)[source(S1)];
30 | .print("La cajal es de color ",Color1,", según ",S1);
31 | ?colorSegunYo(cajal,Color2);
32 | .print("Aparentemente el color de la cajal es ",Color2,", según yo");
33 | if (Color1 \== Color2) {
34 |     +mentiroso(S1)[cert(0.7)]; // Invertir y beto será mentiroso
35 |     +daltonico(S1)[cert(0.3)];
36 | } else {
37 |     +mentiroso(S2)[cert(0.3)];
38 |     +daltonico(S2)[cert(0.7)];
39 | };
40 | ?descr(S1,Des1);
41 | .print(S1, " es un agente ", Des1);
42 | ?descr(S2,Des2);
43 | .print(S2, " es un agente ", Des2).
44 |
45 | @sinContradiccion
46 | +!start <-
47 | .print("No hay contradicciones detectadas");
48 | ?colorSegunYo(cajal,Color);
49 | .print("La cajal es de color ",Color,", según yo").

```

Hay dos planes para contender con la meta principal del agente. El primero detecta contradicciones y el segundo no. En el segundo plan, el agente se pregunta por el color de la caja desde su propia perspectiva (la fuente es `self` o `percept`) y reporta el color encontrado.

Cuando la contradicción es detectada el agente confronta la situación. Reporta el color según su perspectiva y ajusta cuentas con los otros agentes. Si hay otro agente reportando un color diferente, nuestro agente creerá que tal agente es mentiroso o daltónico, con cierto grado de certidumbre. En caso contrario, hay un tercer agente causando la contradicción y éste es el mentiroso/daltónico. La salida en consola para este caso es:

```

1 | [agente11] Contradicción detectada
2 | [agente11] La cajal es de color verde, segun beto
3 | [agente11] Aparentemente el color de la cajal es rojo, según yo
4 | [agente11] beto es un agente daltonico
5 | [agente11] enrique es un agente confiable

```

Si cambiamos la información sobre el color de la `cajal` provista por `enrique` a `azul` (línea 6), tendremos que ya no hay contradicción detectable y la salida del programa es la siguiente:

```

1 | [agente11] No hay contradicciones detectadas
2 | [agente11] La cajal es de color rojo, según yo

```

En cambio si nuestro agente creyera que la `cajal` es de color `verde` (línea 7), entonces el daltónico resultaría `enrique`:

```

1 | [agente11] Contradicción detectada
2 | [agente11] La cajal es de color verde, segun beto
3 | [agente11] Aparentemente el color de la cajal es verde, según yo
4 | [agente11] beto es un agente confiable
5 | [agente11] enrique es un agente daltonico

```

Si se invierten los grados de certeza (líneas 34 y 35), resultará que `beto` es `mentiroso` en lugar de `daltonico`.

```

1 | [agente11] Contradicción detectada
2 | [agente11] La cajal es de color verde, según beto
3 | [agente11] Aparentemente el color de la cajal es roja, según yo
4 | [agente11] beto es un agente mentiroso
5 | [agente11] enrique es un agente confiable

```

10.6 ACCIONES INTERNAS

Es posible definir acciones internas personalizadas, similares a las que hemos introducido en la sección anterior, por ejemplo `math.abs`, etc. Como el ejemplo sugiere, las acciones deben organizarse en **librerías**, que son paquetes de Java; mientras que las acciones propiamente dichas, son clases de Java que implementan la interfaz `InternalAction`. Jason provee una implementación por defecto de esta interfaz, conocida como `DefaultInternalAction`. Las acciones internas se denotan como `librería.acción`.

Librerías

Vamos a crear un SMA centralizado con un solo agente:

```

1 MAS distancia {
2
3   infrastructure: Centralised
4
5   agents:
6     agent1 sample_agent;
7
8   aslSourcePath:
9     "src/asl";
10 }
```

En donde el agente `agent1` haga uso de una acción interna para calcular la distancia euclidiana entre dos puntos:

```

1 // Agent sample_agent in project distancia
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <-
12   ia.distancia(10,10,20,50,D);
13   .println("La distancia euclidiana entre (10,10) y (20,50) es ",D).
```

El código del agente `agent1` ilustra claramente el uso que queremos dar a la acción interna `distancia`. Los ambientes de desarrollo de Jason, permiten agregar acciones internas al sistema. Detalles más, detalles menos, lo importante es decirle al ambiente de desarrollo en que paquete será incluida la acción interna. La implementación de la acción es como sigue (generada, al igual que el SMA en el ambiente de desarrollo basado en eclipse):

```

1 // Internal action code for project distancia
2
3 package ia;
4
5 import jason.*;
6 import jason.asSemantics.*;
7 import jason.asSyntax.*;
8
9 /**
10  * @author aguerra
11  * ia.distancia: Computa la distancia euclidiana entre dos puntos.
12  */
13
14 public class distancia extends DefaultInternalAction {
15
16   private static final long serialVersionUID = 1L;
17
18   @Override
19   public Object execute(TransitionSystem ts, Unifier un, Term[] args)
20     throws Exception {
21     ts.getAg().getLogger().info("executing internal action 'distancia'");
```

```

22     try{
23         NumberTerm x1 = (NumberTerm) args[0];
24         NumberTerm y1 = (NumberTerm) args[1];
25         NumberTerm x2 = (NumberTerm) args[2];
26         NumberTerm y2 = (NumberTerm) args[3];
27
28         double distance = Math.abs(x1.solve()-x2.solve()) +
29             Math.abs(y1.solve()-y2.solve());
30
31         NumberTerm result = new NumberTermImpl(distance);
32         return un.unifies(result,args[4]);
33     } catch (ArrayIndexOutOfBoundsException e) {
34         throw new JSONException("La acción interna 'distancia'+
35             "no ha recibido cinco argumentos!");
36     } catch (ClassCastException e) {
37         throw new JSONException("La acción interna 'distancia'+
38             "ha recibido argumentos no numéricos!");
39     } catch (Exception e) {
40         throw new JSONException("Error en 'distancia'");
41     }
42 }
43 }

```

El diagrama de clases de esta acción se muestra en la Figura 10.13. Su lectura es como sigue: La acción interna `distancia` en el paquete `ia`, extiende la clase `DefaultInternalAction` que implementa la interfaz `InternalAction`.

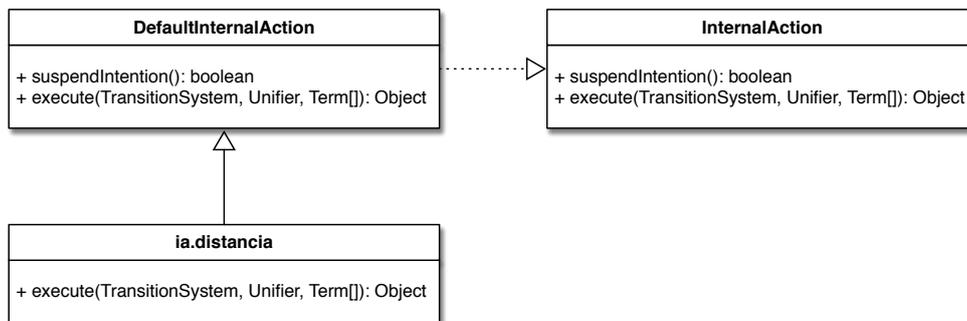


Figura 10.13: El diagrama de clases de la acción interna `distancia`.

La definición de la acción se hace en el paquete `ia` (línea 3). Es necesario importar la sintaxis y la semántica de Jason para poder procesar los argumentos de la acción que son términos numéricos (líneas 22-25).

Como los argumentos de `distancia` son términos numéricos, es posible usar el método `solve` para computarlos, de manera que es posible evaluar algo como `ia.distancia(1/2,1/2,3/4,3/4,D)`. El parámetro de salida de esta acción es el último, por lo que el valor computado `distance`, debe ser convertido a un término numérico (línea 30), antes de unificarlo con el argumento `args[4]`.

La definición de `ia.distancia` también ilustra como procesar dos errores comunes: pasarle el número incorrecto de parámetros y pasarle parámetros no numéricos (líneas 32-39).

Si todo va bien, la salida en consola es la siguiente:

```

1 | [agent1] executing internal action 'ia.distancia'
2 | [agent1] La distancia euclidiana entre (10,10) y (20,50) es 50

```

10.7 MÓDULOS

10.8 ACTOS DE HABLA

La semántica operacional descrita en el capítulo anterior se implementa como una librería de planes que todos los agentes cargan al ser creados. La librería se encuentra en el directorio `/src/main/resources/asl/kqmlPlans.asl` 1 de la distribución de *Jason*. Los planes que contienen con enunciados performativos *tell* son:

```

11  /* ---- tell performatives ---- */
12
13  @kqmlReceivedTellStructure
14  +!kqml_received(Sender, tell, NS::Content, _)
15    : .literal(Content) &
16      .ground(Content) &
17      not .list(Content) &
18      .add_nested_source(Content, Sender, CA)
19    <- ++NS::CA. // add with new focus (as external event)
20  @kqmlReceivedTellList
21  +!kqml_received(Sender, tell, Content, _)
22    : .list(Content)
23    <- !add_all_kqml_received(Sender, Content).
24
25  @kqmlReceivedTellList1
26  +!add_all_kqml_received(_, []).
27
28  @kqmlReceivedTellList2
29  +!add_all_kqml_received(Sender, [NS::H|T])
30    : .literal(H) &
31      .ground(H)
32    <- .add_nested_source(H, Sender, CA);
33    ++NS::CA;
34    !add_all_kqml_received(Sender, T).
35
36  @kqmlReceivedTellList3
37  +!add_all_kqml_received(Sender, [_|T])
38    <- !add_all_kqml_received(Sender, T).
39
40  @kqmlReceivedUnTell
41  +!kqml_received(Sender, untell, NS::Content, _)
42    <- .add_nested_source(Content, Sender, CA);
43    --NS::CA.

```

Observen que el primer caso a considerar es cuando el contenido (*Content*) del mensaje es una creencia (se trata de una sola literal de base, como verifican las acciones internas en el contexto del plan). En ese caso, se agrega la fuente al contenido con la acción interna `.add_nested_source` y se agrega la creencia con su nueva anotación a las creencias del agente. El segundo caso a considerar es cuando se recibe una lista de creencias que serán procesadas recursivamente. El último caso contiene con los enunciados performativos *untell*: se procede a anotar la creencia con su fuente y una vez etiquetada de esa forma, a eliminarla de las creencias del agente.

Los planes que contienen con enunciados performativos *achieve* son:

```

46  /* ---- achieve performatives ---- */
47
48  @kqmlReceivedAchieve
49  +!kqml_received(Sender, achieve, NS::Content, _)
50    : not .list(Content) & .add_nested_source(Content, Sender, CA)
51    <- !!NS::CA.
52  @kqmlReceivedAchieveList
53  +!kqml_received(Sender, achieve, Content, _)
54    : .list(Content)
55    <- !add_all_kqml_achieve(Sender, Content).
56
57
58  @kqmlReceivedAchieveList1
59  +!add_all_kqml_achieve(_, []).

```

```

60
61 @kqmlReceivedAchieveList2
62 +!add_all_kqml_achieve(Sender,[NS::H|T])
63   <- .add_nested_source(H, Sender, CA);
64     !NS::CA;
65     !add_all_kqml_achieve(Sender,T).
66
67
68 @kqmlReceivedUnAchieve[atomic]
69 +!kqml_received(_, unachieve, NS::Content, _)
70   <- .drop_desire(NS::Content).

```

Los planes que contienen con enunciados performativos *ask* son:

```

87 /* ---- ask performatives ---- */
88
89 @kqmlReceivedAskOne1
90 +!kqml_received(Sender, askOne, NS::Content, MsgId)
91   : NS::Content
92   <- .send(Sender, tell, NS::Content, MsgId).
93
94 @kqmlReceivedAskOne1b
95 +!kqml_received(Sender, askOne, NS::Content, MsgId)
96   <- .add_nested_source(Content, Sender, CA);
97     ?NS::CA;
98     // remove source annot from CA
99     CA =.. [-,F,Ts,_];
100    CA2 =.. [-,F,Ts,[]];
101    .send(Sender, tell, NS::CA2, MsgId).

```

Los planes que contienen con enunciados performativos *knowHow* son:

```

90 -!kqml_received(Sender, askOne, NS::Content, MsgId)
91   <- .send(Sender, untell, NS::Content, MsgId).
92
93 @kqmlReceivedAskAll2
94 +!kqml_received(Sender, askAll, NS::Content, MsgId)
95   <- .findall(NS::Content, NS::Content, List);
96     .send(Sender, tell, List, MsgId).
97
98
99 /* ---- know-how performatives ---- */
100
101 // In tellHow, content must be a string representation
102 // of the plan (or a list of such strings)
103
104 @kqmlReceivedTellHow
105 +!kqml_received(Sender, tellHow, Content, _)
106   <- .add_plan(Content, Sender).
107
108 // In untellHow, content must be a plan's
109 // label (or a list of labels)
110 @kqmlReceivedUnTellHow

```

Finalmente, se incluye un plan para contender con los errores generales de comunicación:

```

112   <- .remove_plan(Content, Sender).
113
114 // In askHow, content must be a string representing
115 // the triggering event
116 @kqmlReceivedAskHow

```

10.8.1 Caso de estudio

Definiremos un SMA de dos agentes, *enrique* y *beto* que se comunican utilizando Actos de habla y las acciones internas provistas por Jason. El archivo principal es como sigue:

```

1  /*
2  Demo de comunicación
3
4  Un agente (enrique) se comunica con otro (beto) usando actos de
5  habla implementados en KQML y la acción interna .send
6  */
7
8  MAS comunicacion {
9
10     infrastructure: Centralised
11     agents:
12         enrique [beliefs="receptor(beto)"];
13         beto [verbose=1]; // verbose=2 para ver más detalles
14
15     aslSourcePath: "src/asl";
16 }

```

Este archivo `mas2j` no tiene novedades, salvo que las creencias de *ale* han sido inicializadas con la opción `beliefs`, de forma que *ale* crea que el receptor es *ana*. Esto se utilizará, como veremos a continuación, para dirigir los mensajes. Por otra parte, se puede incrementar el nivel de detalle de la salida en consola con la opción `verbose`. De los dos agentes, *beto* responderá a los mensajes de *enrique*, como tiene un código más corto, lo revisaremos primero:

```

1  // Agente ana en el proyecto comunicacion.mas2j
2
3  vl(1).
4  vl(2).
5
6  /* El siguiente plan se dispara cuando se recibe un mensaje
7  tell. El plan agrega una creencia cuya fuente es el agente
8  emisor del tell, enrique en este caso. */
9  +vl(X)[source(Ag)]
10     : Ag \== self
11     <- .print("Recibió un tell ",vl(X)," de ", Ag).
12
13  /* Igual que el caso anterior pero con una performativa achieve
14  en lugar de tell. */
15  +!ir(X,Y)[source(Ag)] : true
16     <- .println("Recibió un achieve ",ir(X,Y)," de ", Ag).
17
18  /* Cuando bob pregunta t2(X), la respuesta no está en mis
19  creencias. Por tanto el evento "+?t2(X)" se crea y es
20  manejado por el siguiente plan. */
21  +?t2(X) : vl(Y) <- X = 10 + Y.
22
23  /* El siguiente plan es usado para reconfigurar la respuesta a
24  un mensaje con performativa askOne. El plan solo es usado
25  si el contenido de askOne es "nombreComp". Se puede usar
26  un evento de tipo +? para esto, se trata solo de un ejemplo
27  de sobrecarga de directivas de comunicación KQML. */
28  +!kqml_received(Sender, askOne, nombreComp, ReplyWith) : true
29     <- .send(Sender,tell,"Beto Guerra", ReplyWith). // respuesta

```

Lo único a resaltar es el uso de los planes para reconfigurar los actos de habla. Recuerden que la acción interna `.send` se encarga del transporte de los mensajes, pero que estos se ven traducidos a eventos para ser procesados. En este ejemplo, estamos especializando el uso de la performativa `askOne` para que la solicitud no sea procesada por *enrique* como una meta verificable. La variable `ReplyWith` unifica como el identificador *mid* tanto en la solicitud, como en la respuesta.

La definición del agente *enrique* es más larga, así que la analizaremos por partes. Primero, éste agente tiene una sola meta alcanzable `!inicio` que se resuelve intercambiando mensajes con *beto*. Recuerden que la creencia `receptor(beto)` fue inicializada vía el archivo principal `mas2j`. Veamos los primeros casos:

```

1  // Agente enrique en el proyecto comunicacion.mas2j
2
3  !inicio.

```

```

4 |
5 | +!inicio : receptor(A) // Esta creencia viene del mas2j
6 |   <- .println("Enviando tell vl(10)");
7 |     .send(A, tell, vl(10));
8 |
9 |     .println("Enviando achieve ir(10,2)");
10 |    .send(A, achieve, ir(10,2));

```

Iremos intercalando la salida en consola para explicar la interacción entre estos dos agentes. Los mensajes de salida correspondientes a los dos Actos de habla anteriores son:

```

1 | [ale] Enviando tell vl(10)
2 | [ale] Enviando achieve ir(10,2)

```

Además de la salida, el estado mental del agente *beto* está cambiando. En este caso se ha agregado $vl(10)_{[source(enrique)]}$ a sus creencias. También ha añadido $\langle +!ir(10,2)_{[source(enrique)]}, \top \rangle$ a su cola de eventos. Continuemos con el resto de los mensajes en el plan inicial del agente *enrique*:

```

12 |     .println("Enviando solicitud síncrona ");
13 |     .send(A, askOne, vl(X), vl(X));
14 |     .println("La respuesta a la solicitud es: ", X, " (debe ser 10)");
15 |
16 |     .println("Enviando solicitud asíncrona ");
17 |     .send(A, askOne, vl(_)); // como es asíncrona no tiene 4o argumento
18 |     // la respuesta se recibe vía un evento +vl(X)
19 |
20 |     .println("Preguntando algo que Ana no cree, pero puede responder con +? ");
21 |     .send(A, askOne, t2(_), Ans2);
22 |     .println("La respuesta a la solicitud es: ", Ans2, " (debe ser t2(20))");
23 |
24 |     .println("Preguntando por algo que ",A," no sabe.");
25 |     .send(A, askOne, t1(_), Ans1);
26 |     .println("La respuesta es: ", Ans1, " (debe ser false)");
27 |
28 |     .println("Solicitando valores con askall");
29 |     .send(A, askAll, vl(Y), List1);
30 |     .println("La respuesta es: ", List1, " (debe ser [vl(10),vl(1),vl(2)])");
31 |
32 |     .println("Solicitando un askall de t1(X).");
33 |     .send(A, askAll, t1(Y), List2);
34 |     .println("La respuesta es: ", List2, " (debe ser [])");

```

Estos actos de habla hacen diferentes solicitudes a *ana* sobre sus creencias. Algunas solicitudes pueden ser resueltas como metas verificables; otras por medio de planes; y otras no pueden responderse. La salida en consola para estos mensajes es como sigue:

```

1 | [ale] Enviando solicitud síncrona
2 | [ana] Recibió un tell vl(10) de ale
3 | [ale] La respuesta a la solicitud es: 10 (debe ser 10)
4 | [ale] Enviando solicitud asíncrona
5 | [ana] Recibió un achieve ir(10,2) de ale
6 | [ale] Preguntando algo que Ana no cree, pero puede responder con +?
7 | [ale] Valor recibido 10 de ana
8 | [ale] La respuesta a la solicitud es: t2(20)[source(ana)] (debe ser t2(20))
9 | [ale] Preguntando por algo que ana no sabe.
10 | [ale] La respuesta es: false (debe ser false)
11 | [ale] Solicitando valores con askall
12 | [ale] La respuesta es: [vl(10)[source(ana)],vl(1)[source(ana)],vl(2)[source(ana)]] (debe ser [vl(10),vl(1),vl(2)])
13 | [ale] Solicitando un askall de t1(X).
14 | [ale] La respuesta es: [] (debe ser []).

```

Observen que la salida incluye algunas de las respuestas de *beto* a los mensajes iniciales de *enrique*. Por ejemplo, la segunda línea es una respuesta al evento generado por la primera solicitud de *enrique*. El retardo en la respuesta se debe a que *beto* debe procesar el evento, seleccionando un plan aplicable, formando la intención corres-

pondiente y ejecutándola. Lo mismo sucede para la respuesta a la solicitud `!ir(20,2)` por parte de `enrique`.

Observen también el comportamiento diferente entre las solicitudes síncronas y asíncronas. Las primeras instancian la respuesta en el mensaje mismo; las segundas generan eventos.

Finalmente, el código del agente `enrique` se completa como sigue:

```

36     .println("Preguntado el nombre completo de Beto.");
37     .send(A, askOne, nombreCompl, FN);
38     .println("El nombre completo de ",A," es ",FN);
39
40     // Preguntare a Ana el plan para ir a algún sitio
41     .send(A, askHow, {+!ir(_)[source(-)]});
42     .wait(500); // esperar la respuesta 500 ms
43     .print("Planes recibidos:");
44     .list_plans( {+!ir(_)[source(-)]} );
45     .print;
46
47     // Otra implementación (no agrega el plan automáticamente a la
48     // librería de planes)
49     .send(A, askHow, {+!ir(_)[source(-)]}, ListOfPlans);
50     .print("Planes recibidos: ", ListOfPlans);
51
52     // Enviándole a beto un plan para !hello
53     .plan_label(Plan, hp); // obtiene un Plan a partir de su etiqueta (hp)
54     .println("Enviando un tellhow de: ", Plan);
55     .send(A, tellHow, Plan);
56
57     .println("Pidiéndole a ",A," satisfacer !hola(ale).");
58     .send(A, achieve, hola(ale));
59     .wait(2000);
60
61     .println("Pidiéndole a ",A," satisfacer -!hola(ale).");
62     .send(A, unachieve, hola(ale));
63
64     // Enviar un untellHow a beto
65     .send(A, untellHow, hp).
66
67
68 +vl(X)[source(A)]
69     <- .print("Valor recibido ",X," de ",A).
70
71 @hp // El plan que será enviado a beto
72 +!hola(Quien)
73     <- .println("Hola ", Quien);
74     .wait(100);
75     !hola(Quien).

```

Esta parte del código incluye ejemplos más elaborados del uso de los Actos de habla en Jason. Primero, la línea 37 explota la especialización de `askOne` que `beto` implementa para poder contestar directamente su nombre. El resto de las solicitudes tienen que ver con intercambio de planes y la solicitud de iniciar una meta alcanzable y dejar de hacerlo. La salida en consola es como sigue:

```

1 [ale] Preguntado el nombre completo de Beto.
2 [ale] El nombre completo de ana es Beto Guerra
3 [ale] Planes recibidos:
4 [ale] @L_4[source(ana)] +!ir(_41X,_42Y)[source(_40Ag)] <- .println("Recibí un achieve ",ir(_41X,_42Y)," de ",_40Ag)
5 [ale]
6 [ale] Planes recibidos: [{ @L_4 +!ir(_44X,_45Y)[source(_43Ag)] <- .println("Recibió un achieve ",ir(_44X,_45Y)," de
7 [ale] Enviando un tellhow de: { @hp +!hola(_43Quien) <- .println("Hola ",_43Quien); .wait(100); !hola(_43Quien) }
8 [ale] Pidiéndole a ana satisfacer !hola(enrique).
9 [ana] Hola enrique
10 ...
11 [ana] Hola enrique
12 [ale] Pidiéndole a ana satisfacer -!hola(enrique).

```

10.9 LECTURAS Y EJERCICIOS SUGERIDOS

Definitivamente, la mejor referencia para Jason es el libro de Bordini, Hübner y Wooldridge [12], en el cual se basa la presentación sobre las creencias de este capítulo. La introducción a Jason se basa en los mini tutoriales incluidos en su distribución⁷. Como se mencionó, la distribución de Jason también incluye una serie de ejemplos y demos que cubren todos los aspectos a revisar de este lenguaje de programación orientado a agentes.

Existen muchas referencias a la Programación Lógica y Prolog que pueden usarse para profundizar este paradigma y explotarlo mejor en Jason. Clocksin y Melish [25] nos ofrecen una introducción concisa y breve a Prolog. Bratko [17] escribió un clásico en IA, que ejemplifica el uso de Prolog en problemas propios del área. Este texto es mucho más extenso que el anterior, pero puede consultarse por temas de interés, por ejemplo búsquedas, heurísticas, sistemas expertos, planeación, aprendizaje, etc. El texto propuesto por Sterling y Shapiro [102] forma parte de una excelente serie dedicada a la Programación Lógica del MIT Press, va en el mismo estilo que el texto anterior. Algunos de los programas *AgentSpeak(L)* de este capítulo, son adaptaciones de algunos programas lógicos de mi curso de Programación para la IA⁸.

La referencia obligada para entender los problemas de la negación y el principio del mundo cerrado, es el artículo de Reiter [91], aunque esa presentación se da en el contexto de la Programación Lógica y las Bases de Datos.

El capítulo 3 del libro compilado por Weiß [109] ofrece una introducción a la comunicación entre agentes. Labrou, Finin y Peng [60]; y Dignum y Greaves [32] ofrecen una revisión de diversos trabajos en comunicación entre agentes.

Ejercicios sugeridos

Ejercicio 10.1. *Modifique que agente piro para que el mismo apague el fuego que ha iniciado... con cierta probabilidad de éxito.*

Ejercicio 10.2. *Agregue otro agente al proyecto piro, que sea sensible a las acciones de piro.*

Ejercicio 10.3. *¿Cómo se puede lograr que enrique y beto se saluden por siempre?*

Ejercicio 10.4. *Convierta el proyecto *cleaning-robots* de la distribución de Jason, en un proyecto compatible con el plug-in de Eclipse.*

Ejercicio 10.5. *Modifique al agente *r1* del proyecto *cleaning-robots*, para que una vez que tenga la basura, avise al agente *r2*.*

Ejercicio 10.6. *Ilustre con ejemplos los casos de unificación entre variables anotadas, cuando una de ellas es de base y la otra no.*

Ejercicio 10.7. *En el caso del agente 11, cambie el orden de las creencias asociadas a beto, enrique y self como fuentes. ¿Qué sucede con la salida del programa?*

Ejercicio 10.8. *En el caso del agente 11, invierta el orden de los planes ¿Qué sucede en la salida del programa?*

Ejercicio 10.9. *Revisen este material identificando alguna fuerza ilocutoria no definida en Jason, por ejemplo *promise* ¿Qué tan complicado es agregarla a Jason?*

Ejercicio 10.10. *Intenten correr el ejemplo de ana y ale en una arquitectura distribuida bajo Jade.*

⁷ /doc/mini-tutorial

⁸ <http://www.uv.mx/personal/aguerra/pia>